Computer Science: Abstraction to Implementation

Robert M. Keller
Harvey Mudd College
keller@cs.hmc.edu

September 2001

Table of Contents

1. Introduction

This book is intended for a broad second course in computer science, one emphasizing principles wherever it seems possible at this level. While this course builds and amplifies what the student already knows about programming, it is not limited to programming. Instead, it attempts to use various programming models to explicate principles of computational systems. Before taking this course, the student should have had a solid one-semester course in computer programming and problem-solving, ideally using the Java™ language, since some of the presentation here uses Java. The philosophy taken in this course is that computer science topics, at an introductory level, are best approached in an integrated fashion (software, theory, and hardware) rather than as a series of individual isolated topics. Thus several threads are intertwined in this text.

1.1 The Purpose of Abstraction

This text touches, at least loosely, upon many of the most important *levels of abstraction* in computer systems. The term *abstract* may be most familiar to the student in the form of an adjective, as in *abstract art.* That association may, unfortunately, conjure a picture of being difficult to understand. In fact, the use we make of the term of *abstract* is to *simplify*, or *eliminate irrelevant detail*, as in the abstract of a published paper, which states the key ideas without details. In computer science, an abstraction is an intellectual device to simplify by eliminating factors that are irrelevant to the key idea. Much of the activity of computer science is concerned with inventing abstractions that simplify thought processes and system development.

The idea of levels of abstraction is central to managing complexity of computer systems, both software and hardware. Such systems typically consist of thousands to millions of very small components (words of memory, program statements, logic gates, etc.). To design all components as a single monolith is virtually impossible intellectually. Therefore, it is common instead to view a system as being comprised of a few interacting components, each of which can be understood in terms of *its* components, and so forth, until the most basic level is reached.

Thus we have the idea of *implementing* components on one level using components on the level below. The level below forms a set of abstractions used by the level being implemented. In turn, the components at this level may form a set of abstractions for the next level. For example, proposition logic (also called switching logic, or sometimes Boolean algebra) is an abstraction of what goes on at the gate-level of a computer. This logic is typically implemented using electronics, although other media are possible, for example mechanical logic or fluid logic. Switching logic, with the addition of memory components such as flip-flops, is the basis for implementing finite-state machines. Components such as registers and adders are built upon both logic and finite-state machines. These components implement the instruction set of the computer, another abstraction. The instruction set of the computer implements the programs that run on the

computer. A compiler or assembler is a program that translates a program written in a more user-friendly language into commands in the instruction set. The program might actually be an interpreter or "virtual machine" for a still higher level language, such as Java™.

We never can be sure how far these levels may go; what were once complete systems are now being fashioned into networks of computers, and those networks into networks, which themselves are replete with layers of abstractions (called "protocol stacks"). The same phenomenon occurs for software: compilers and interpreters for new languages may be built atop existing languages. Figure 1 is meant to summarize some of these important levels of abstraction.

The benefits of using abstraction are not unique to computer science; they occur in many disciplines and across disciplines. For example:

> *Chemistry is an abstraction of physics*: The purpose of chemistry is to understand molecular interactions without resorting to particle physics to explain every phenomenon.
>
> *Biology is an abstraction of chemistry*: The purpose of biology is to understand the growth and behavior of living things without resorting to molecular explanations for every aspect.
>
> *Genetics is an abstraction of biology*: The purpose of genetics is to understand the evolution of traits of living organisms. Genetics develops its own abstractions based on genes which don't require appealing to cells in every instance.

We could go on with this list. Note that we are saying *an abstraction* rather than *the abstraction*. It is not implied that the abstraction in question covers all aspects of the field being abstracted, nor that it is the only possible abstraction of that field.

Some of the specific advantages in treating systems by levels of abstraction are:

- Each level has its own definition and specification. This means that development using this level can proceed concurrently with development at the next level.

- A system can be developed by more than one individual, each a specialist in a particular aspect of construction. This is important since some systems are sufficiently ambitious that they would be impossible to develop by a single person in his or her lifetime.

- A system can evolve by evolving components separately; it is not necessary to re-implement the entire system when one component changes.

- A non-working system can be diagnosed by diagnosing components at their interfaces, rather than by exhaustively tracing the functions of all components.

Historically, implementations tended to come before abstractions. That is, a system got built, then abstractions were used to simplify its description. However, increasingly, we need to think of the abstractions first then go about implementing them. The abstractions provide a kind of specification of what is to be implemented.

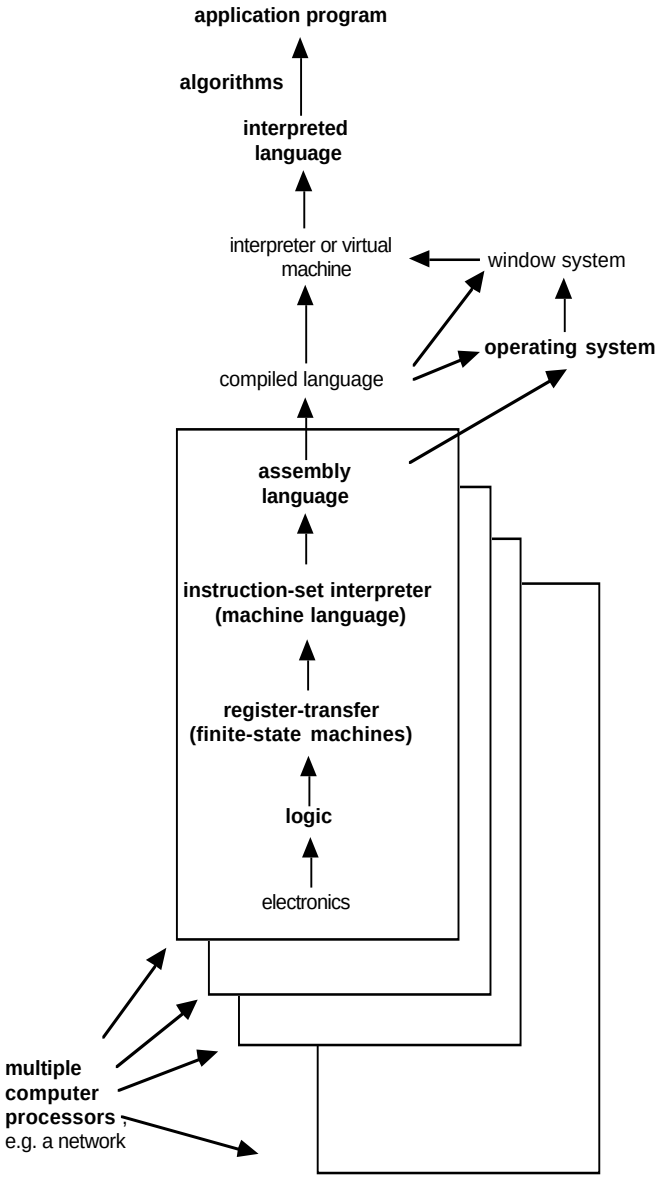


Figure 1: Typical levels of abstraction in computer science. The bolder items are ones given emphasis in this book.

Constructing abstractions is somewhat of an art, with or without a prior implementation. But understanding abstractions is rapidly becoming a way of life for those with any more than a casual relationship to computers. As new languages emerge, they are increasingly expressed using abstractions. A prime example is the object-oriented language Java™, which we use in part of this text to exemplify object-oriented concepts. The idea of inheritance is used heavily in the description of Java libraries. Inheritance hierarchies are miniature abstraction hierarchies in their own right.

At the same time, we are interested in how certain key abstractions are actually realized in hardware or in code, hence the word *implementation* in the title of the text. Having an understanding of implementation issues is important, to avoid making unreasonable assumptions or demands upon the implementor.

1.2 Principles

One might say that a science can be identified with its set of principles. Computer Science is relatively young as a discipline, and many of its principles are concerned with concepts lying at a depth which is beyond an introductory course such as this. Nevertheless, a conscious attempt is made to identify ideas as *named principles* wherever possible. Many of these principles are used routinely by computer scientists, programmers, and designers, but do not necessarily have standard names. By giving them names, we highlight the techniques and also provide more common threads for connecting the ideas. Although a modicum of theory is presented throughout the text, we are interested in imparting the ideas, rather than devoting attention to rigorous proofs.

While many of the points emphasized are most easily driven home by programming exercises, it is important to understand that the course is not *just* about programming, but rather about underlying conceptual continua that programming can best help illustrate.

1.3 Languages

The text is not oriented to a particular language, although a fair amount of time is spent on some language specifics. The educational "industry" has emerged from a point where Pascal was the most widely-taught introductory language. It was about ready to move on to C++ when Java™ appeared on the horizon. Java is a derivative of C++, which offers most of the object-oriented features of the latter, but omits some of the more confusing features. (As is usually the case, it introduces some new confusing features of its own.) For that reason, we start our discussion of object-orientation with Java. Another strong feature of Java, not heavily exploited in this text, is that working application programs, or “applets” as they are called, can be made available readily on the Internet. This text is not, in any way, to be regarded as a replacement for a handbook on any particular language, especially Java or C++. It is strongly advised that language handbooks be available for reference to specific language details that this text does not cover.

One purpose of a language is to give easy expression to a set of concepts. Thus, this book starts not with Java but rather a functional language *rex* of our own design. An interpreter for rex (implemented in C++) is provided. The rationale here is that there are many important concepts that, while they can be applied in many languages, are most cleanly illustrated using a functional language. To attempt to introduce them in Java would be to obscure the concepts with syntactic rubric. We later show how to transcribe the thinking and ideas into other languages. The current object-oriented bandwagon has much to recommend it, but it tends to overlook some of the important ideas in functional programming. In particular, functional programs are generally much easier to show *correct* than are object-oriented programs; there is no widely-accepted mathematical theory for the latter.

1.4 Learning Goals

The expected level of entry to this course is that students know basics of control-flow (*for* and *while* statements), are comfortable with procedures, know how and when to use arrays and structures, and understand the purposes of a type system. The student has probably been exposed to recursion, but might not be proficient at using it. The same is true for pointers. There may have been brief exposure to considerations behind choices of data structures, the analysis of program run-time, and the relationship between language constructs and their execution on physical processors. These things, as well as the structure of processors and relation to logic design, are likely to be gray areas and so we cover them from the beginning.

The student at this point is thus ready to tackle concepts addressed by this book, such as:

- information structures (lists, trees, directed graphs) from an abstract viewpoint, independent of particular data structure implementations
- recursion as a natural problem-solving technique
- functional programming as an elegant and succinct way to express certain specifications in an executable form
- objects and classes for expressing abstract data types
- underlying theoretical models that form the basis for computation
- inductive definitions and grammars for expressing language syntax and properties of sequences, and their application to the construction of simple parsers and interpreters from grammatical specifications
- proposition logic in specifying and implementing hardware systems and in program optimization
- predicate logic in specifying and verifying systems, and directly in programming

- advanced computing paradigms such as backtracking, caching, and breadth-first search
- use of techniques for analysis of program run-time complexity and the relationship to data-structure selection
- structure of finite-state machines how they extend to full processors
- assembly language, including how recursion is implemented at the assembly language level
- introduction to parallel processing, multithreading, and networking
- introduction to theoretical limitations of computing, such as problems of incomputability

These are among the topics covered in this book.

1.5 Structure of the Chapters

The chapters are described briefly as follows. There is more than adequate material for a one-semester course, depending on the depth of coverage.

1. ***Introduction*** is the current chapter.

2. ***Information Structures*** discusses various types of information, such as lists, trees, and directed graphs. We focus on the structure, and intentionally avoid getting into much programming until the next chapter.

3. ***High-Level Functional Programming*** discusses functions on the information structures used previously. The emphasis here is on thinking about high-level, wholesale, operations which can be performed on data.

4. ***Low-Level Functional Programming*** shows how to construct programs which carry out the high-level ideas introduced in the previous chapter. A rule-based approach is used, where each rule tries to express a thought about the construction of a function. We go into simple graph-processing notions, such as shortest path and transitive closure.

5. ***Implementing Information Structures*** presents methods of implementing a variety of information and structures in Java, including many of the structures discussed in earlier chapters.

6. ***States and Transitions*** discusses the basis of state-oriented computation, which is a prolog to object-oriented computation. It shows how state can be modeled using the

framework introduced in previous chapters. We show how conventional imperative programs can be easily transformed into functional ones. We illustrate the idea of a Turing machine and discuss its acceptance as a universal basis for computation.

7 . ***Object-Oriented Programming*** introduces object-oriented concepts for data abstraction, using Java as the vehicle. We explore ideas of polymorphism, and construct a model of polymorphic lists, matching the kind of generality available in functional programming systems. We describe the implementation of higher-order functions. We include a discussion of the uses of inheritance for normalizing software designs.

8 . ***Grammars and Parsing*** introduces the concept of grammars for specifying languages. It also shows the construction of simple parsers for such languages. The idea here is that in many cases we have to solve not just one problem but rather an entire family of problems. Indeed, we may need to provide such a language to a community of users who do not wish to get involved with a general purpose programming language. Inventing a language in which to express a family of problems, and being able to construct an interpreter for that language, is viewed as a helpful skill.

9. ***Proposition Logic*** begins with basic ideas of proposition logic from a functional point of view. We the show role these ideas play in hardware design, and go into some of the theory of simplification of logical expressions. Physical bases for computing are mentioned briefly

10. ***Predicate Logic*** introduces predicate logic and demonstrate its use in specifying and proving programs. We also show how predicate logic can be used for direct programming of databases and other applications, using the Prolog language.

11. ***Complexity*** introduces the idea of measuring the running time of a program across a wide spectrum of inputs. We use the "O" notation, defining it in a simplified way appropriate to the application at hand, analyzing programs. We show how programs can be analyzed when they are decomposed into sequential compositions, loops, and recursion. We use sorting and searching applications for many of the examples. We also mention hashing and related techniques.

12. ***Finite-State Machines*** introduces various finite-state machine models and how they are implemented. We work our way into the implementation of simple digital sub-systems based on finite-state machines. We conclude with a discussion of data communication issues, such as the use of 3-state buffer devices.

13. ***Stored-Program Computing*** talks about the structure and programming of stored-program computers from a fairly low level. This ties together programming concepts and concepts from finite-state computing. We present a simulated computer, the ISC, and its assembly language.

14. ***Parallel Computing*** discusses issues related to performing multiple computations at the same time. We review cellular automata, data-parallel computing, process-oriented approaches, suitable for multiple-instruction-stream/multiple-data-stream computers, and discuss the importance of this emerging area in the network-based computers of the future.

15. ***Limitations of Computing*** mentions some of the logical and physical limitations of computing. We discuss algorithmic lower bounds, the limitations of finite-state machines, the notions of incomputability and intractability, and the glitch phenomenon.

Figure 2 gives an approximate dependence among chapters.

1.6 How the Problems are Rated

We use a dot notation to visually suggest a problem's difficulty:

- G: Intended to be workable based on just an understanding of the prior readings.

•• PG: Intended to be workable based on an understanding of readings plus a little effort.

••• PG13: Workable with a little effort and perhaps a hint or two.

•••• R: Intended for mature audiences; requires substantial effort and possibly extra insight or perseverance.

••••• NC17: Obscenely difficult; might be the subject of a research paper, past or future. Intended for perspective, not necessarily to be done in the mainstream of the course.

Exercises

1. •• Cite an example from your own experience of an abstraction and its implementation.

2. ••• Identify some areas outside of computer science, such as in chemistry, music, dance, etc. where abstractions are heavily used. Give details.

3. •••• Identify some areas outside of computer science where several *levels* of abstraction are used.

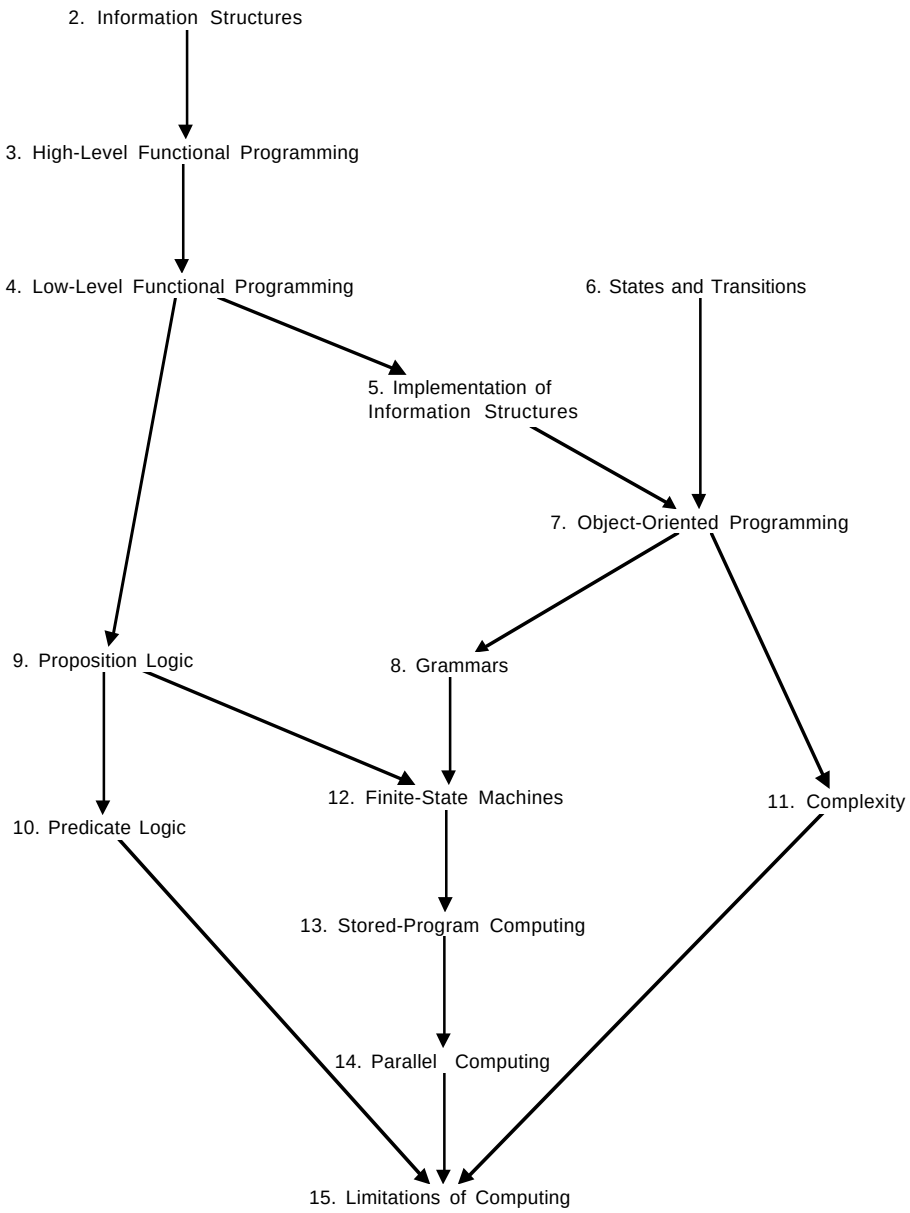


Figure 2: Chapter Dependence

1.7 Further Reading

Each chapter lists relevant further reading. In many cases, original sources are cited, which are sometimes not very light reading. We try to provide a qualitative estimate of difficulty at the end of each annotation. The following tend to lighter surveys, which cover some of the ideas in the text (as well as others) from different perspectives, but still at a more-or-less less technical level.

Alan W. Biermann, *Great Ideas in Computer Science*, M.I.T. Press, 1990. [A textbook approach to introductory examples. Algorithms are presented in Pascal. Easy to moderate.]

Glenn Brookshear, *Computer Science – An Overview*, Third Edition, Benjamin/Cummings, 1991. [Easy.]

Richard P. Feynman, Feynman Lectures on Computation, Edited by J.G. Hey and Robin W. Allen, Addison-Wesley, 1996. [A famous physicist talks about computation and computer science; moderate.]

A.K. Dewdney, *The (New) Turing Omnibus – 66 Excursions in Computer Science*, Computer Science Press, 1993. [Short (3-4 page) articles on a wide variety of computer science topics. Algorithms are presented in pseudo-code. Easy to moderate.]

David Harel, *Algorithmics – The Spirit of Computing*, Addison-Wesley, 1987. [Moderate.]

Anthony Ralston and Edwin D. Reilly, *Encyclopedia of Computer Science*, Van Nostrand Reinhold, 1993.

1.8 Acknowledgment

The following is a partial list of individuals whom the author wishes to thank for comments and corrections: Jeff Allen, James Benham, Jason Brudvik, Andrew Cosand, Dan Darcy, Jason Dorsett, Zachary Dodds, Elecia Engelmann, Kris Jurka, Ran Libeskind-Hadas, Eran Karmon, Geoffrey Kuenning, Stephen Rayhawk, David Rudel, Chuck Scheid, Virginia Stoll, Chris Stone, Ian Weiner, and Wynn Yin.

2. Exploring Abstractions: Information Structures

2.1 Introduction

Which comes first, the abstraction or the implementation? There is no fixed answer. Historically abstractions were introduced as ways of simplifying the presentation of implementations and then standardizing implementations. Increasingly, developers are being encouraged to think through the abstractions first, then find appropriate ways to implement them. In this section we will discuss various abstractions for structuring information, which is related to implementation issues of data structures.

We assume that the reader has been exposed to (uni-directional) linked-list data structuring concepts. Linked lists are often presented as a way to represents sequences in computer memory, with the following property:

> Insertion of a new item into a linked list, given a reference to the insertion point, entails at most a fixed number of operations.

Here "fixed" is as opposed to a number of operations that can vary with the length of the sequence. In order to achieve this property, each item in the list is accompanied by a *pointer* to the next item. In the special case of the last item, there is no next, so a special *null pointer* is used that does not point to anything.

After, or along with, the process of demonstrating linked list manipulation, box diagrams are typically illustrated. These show the items in the list in one compartment of a box and the pointer in a second compartment, with an arrow from that compartment to the box containing the next item. A diagonal line is typically used instead of the arrow in the case of a null pointer. Finally, a pointer leading from nowhere indicates the first element of the list.

For example, a sequence of four elements: a, b, c, d would be shown as in the following box diagram.

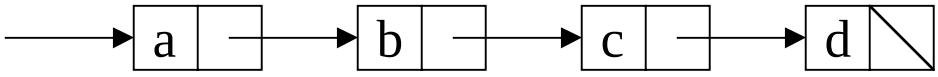


Figure 3: A box diagram for a linked list of four elements

It is worth mentioning that box diagrams are themselves abstractions of data inside the computer. In order to implement the concept of "pointing", we typically appeal to an addressing or indexing mechanism, wherein each potential box has an implied numeric address or index. (Addressing can be regarded as a special case of indexing in which the entire memory is indexed.) The underlying implementation of the box diagram

abstraction could be shown as in Figure 4, where the numbers at the left are the indices of the boxes. We use an index value of –1 to represent the null pointer. The empty boxes are currently unused. With this representation, we also need to keep track of the first element, which in this case is the one at index 0. Note that there can be many representations for a single abstraction; this is a common phenomenon.

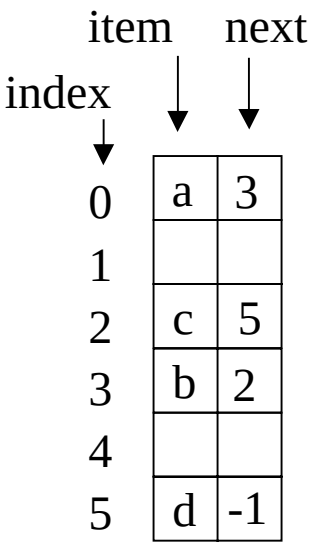


Figure 4: A representation, using indices, for a linked list of four elements

At the representation level, in order to insert a new element, we simply obtain a new box for it and adjust the structure accordingly. For example, since the box at index 1 is unused, we can use it to hold the new element. Suppose the element is e anl it is to be inserted after b. The new representation picture would be as shown in Figure 5

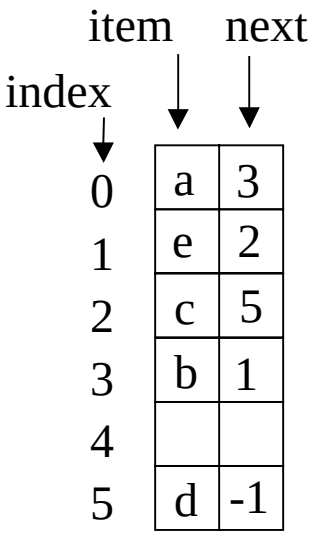


Figure 5: A representation for the modified linked list, of five elements

Returning to the abstract level, we can show the sequence of modifications for inserting the new element as in Figure 6.

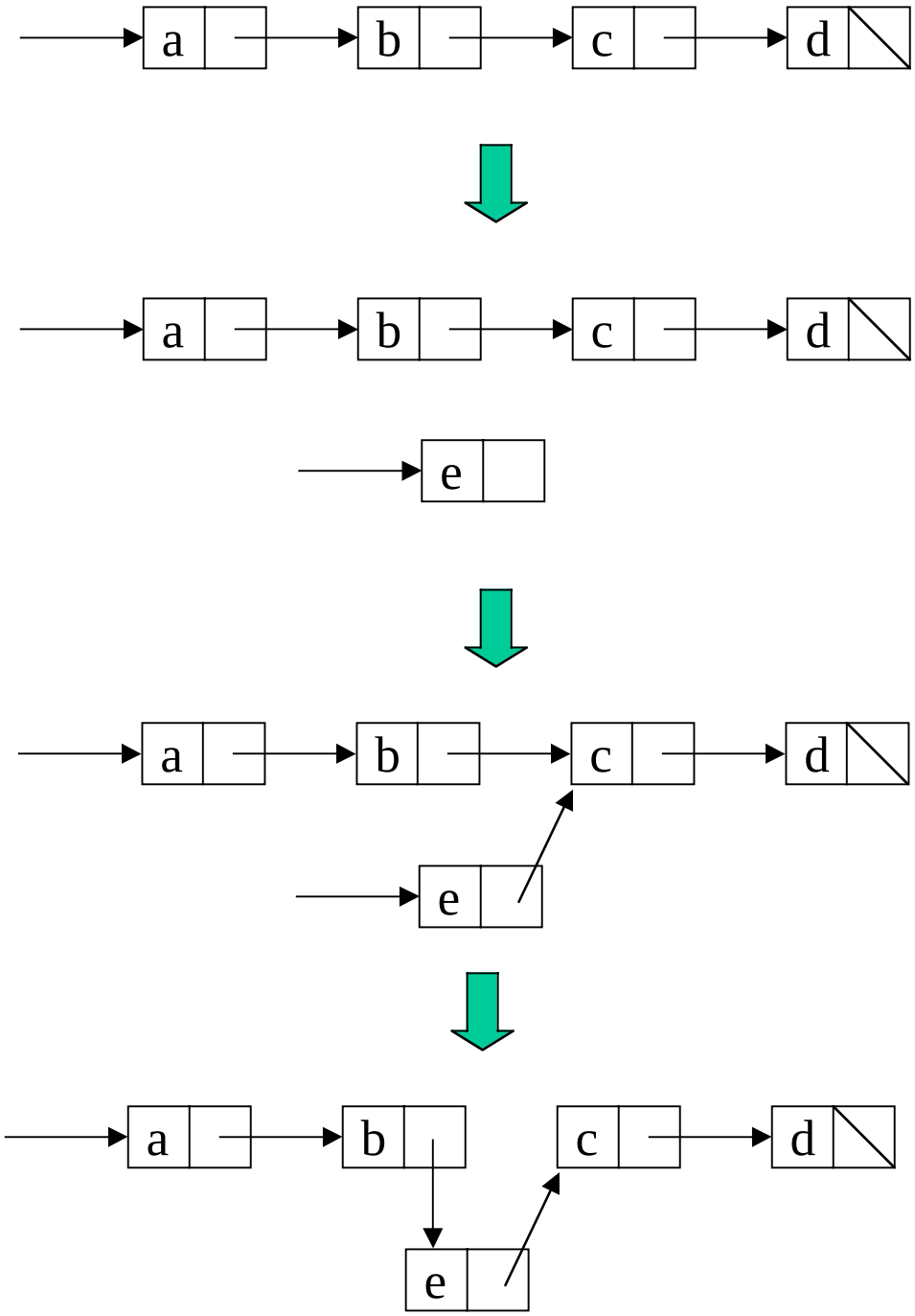


Figure 6: Sequence of changes in a box diagram to represent insertion. The arrows show the changes to the state of the list.

From the third state in the sequence of changes, we can observe something interesting: in this state, two pointers point at the box containing item, rather than one. This suggests an interesting possibility: that parts of lists can be *shared* between two or more lists. There is a fundamental split in list processing philosophies which has to do with whether such sharing is exploited or discouraged. This text refers to the philosophy in which lists are frequently shared as *open* lists, and the opposite philosophy as *closed* lists. Chapter 5 gives more details on the distinctions between the two.

The present chapter is going to focus on open lists almost exclusively. The next figure shows two lists sharing a common "tail". The positive aspect of such sharing is that the space saved for the stem is used only once even though the effect, from the viewpoint of each list, is that each enjoys equal use of the tail. The negative aspect is that we must be extremely careful about any modifications that take place in the tail; if the user of one list modifies the tail, then the user of the other list "feels" the modification. This might be unintended. Because it is hard to manage these types of changes, it is typical to *forbid modifications* to lists in the open list model.

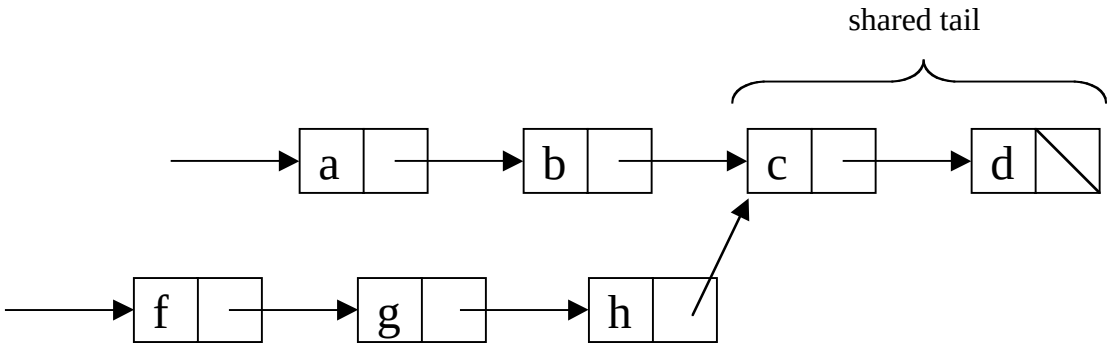


Figure 7: Two lists with a shared tail

One might wonder, if modifications to lists are forbidden in the open list model, how does anything get done? The answer may be surprising: *only create new lists.* In other words, to achieve a list that is similar to an existing list, create a new list so modified. This might seem incredibly wasteful, but it is one of the major philosophies for dealing with lists. Moreover, we still have the possibility of tail sharing, which can overcome a lot of the waste, if we do things correctly. Since we agree to modify the structure of a list, we can share a tail of it freely among an arbitrary number of lists. In fact, the shared tails need not all be the same, as shown in the Figure 8.

Notice that it only makes sense to share tails, not "heads", and to do so it simply suffices to have access to a pointer to the start of the sub-list to be shared.

So far we have seen the box abstraction for open lists, and not shown its lower-level implementation in a programming language. What we will do next is develop a textual abstraction for dealing with open lists. Then we will use *it* as the *implementation* for some higher level abstractions. This illustrates how abstractions can be stacked into

levels, with one level of abstraction providing the implementation for the next higher level.

Our approach derives structurally from languages such as Lisp, Prolog, and their various descendants. An interpreter for our language, called *rex,* is available on common platforms such as UNIX® and Windows, so that the specifications can be understood and tried interactively.

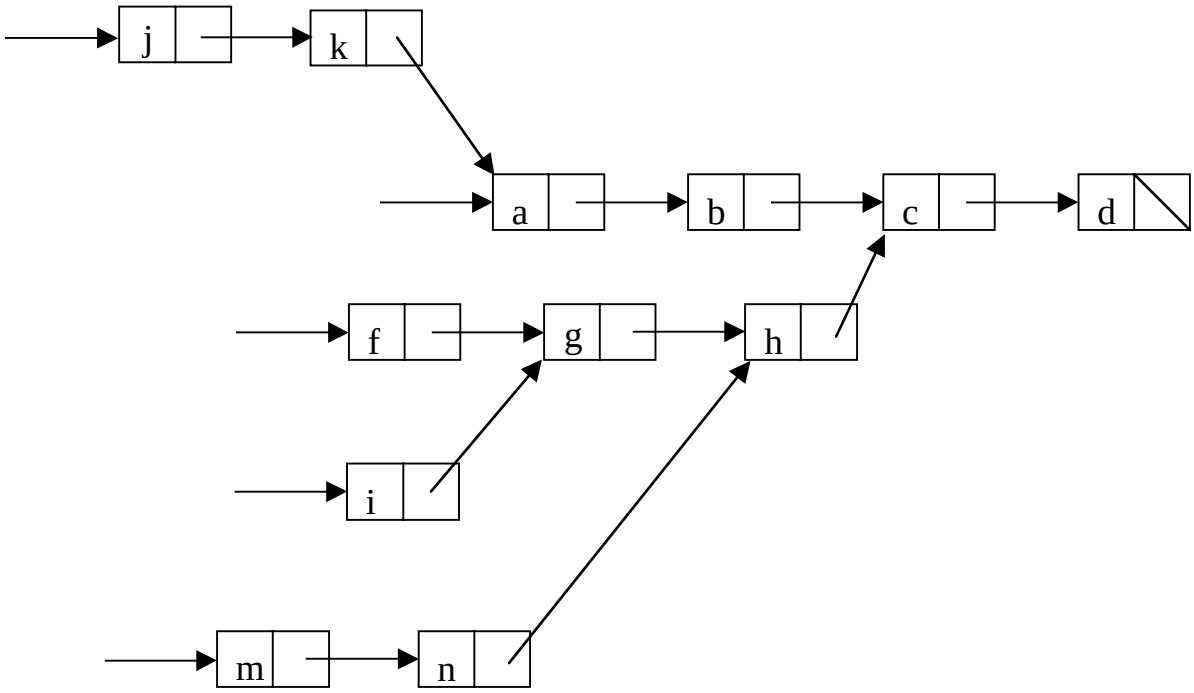


Figure 8: Multiple tail-sharing

Despite the fact that we express examples in our own language, the reader should be reassured that *the medium is not the message*. That is, we are not trying to promote a specific language; the ideas and concepts are of main interest. From this neutrality comes the possibility of using the ideas in many different languages. A few examples of mapping into specific languages will be shown in this book. For example, in addition to rex, we will show a Java implementation of the same ideas, so that the programming concepts will carry forth to Java. We contend that is easier to see the concepts for the first time in a language such as rex, since there is less syntactic clutter.

We should also comment on information structures vs. *data structures*. The latter term connotes structures built in computer memory out of blocks of memory and references or pointers. While an information structure can be implemented in terms of a data structure, the idea of information structures attempts to suppress specific lower-level implementation considerations. Put another way, an information structure is an *abstraction* of a data structure. There could be several different data structures implementing one information structure. This is not to say that all such data structures

would be equally attractive in terms of performance. Depending on the intended set of operations associated with an information structure, one data structure or another might be preferable.

2.2 The Meaning of "Structure"

Computing is expressed in terms of primitive elements, typically numbers and characters, as well as in terms of structures composed of these things. Structures are often built up hierarchically: that is, structures themselves are composed of other structures, which are composed of other structures, and so on. These more complex structures are categorized by their relative complexity: trees, dags (directed acyclic graphs), graphs, and so on. These categories are further refined into sub-categories that relate to particular applications and algorithmic performance (roughly translating into the amount of time a computation takes): binary-search trees, tries, heaps, etc.

When a datum is not intended for further subdivision, it is called *atomic*. The property of being atomic is relative. Just as in physics, atoms can be sub-divided into elementary particles, so in information, atomic units such as numbers and character strings could conceivably be sub-divided. Strings could be divided into their characters. Numbers could, in a sense, be divided by giving a representation for the number using a set of digits. Of course, in the case of numbers, there are many such representations, so the subdivision of numbers into digits is not as natural as for strings into characters.

2.3 Homogeneous List Structures

We begin by describing a simple type of structure: lists in which all elements are of the same type, which are therefore called *homogeneous* lists. However, many of the ideas will also apply to heterogeneous lists as well. To start with, we will use numerals representing numbers as our elements.

A list of items is shown in its entirety by listing the items separated by commas within brackets. For example:

```
[2, 3, 5, 7]
```

is a list of four items. Although the notation we use for lists resembles that commonly used for sets (where curly braces rather than square brackets are used), there are some key differences:

- Order matters for lists, but not for sets.
- Duplication matters for lists, but not for sets.

Thus while the sets {2, 5, 3, 7, 3} and {2, 3, 5, 7} are regarded as equal, the two lists `[2, 5, 3, 7, 3]` and `[2, 3, 5, 7]` are not.

Two lists are defined to be *equal* when they have the same elements in exactly the same order.

2.4 Decomposing a List

Consider describing an algorithm for testing whether two lists are equal, according to the criterion above. If we were to base the algorithm directly on the definition, we might do the following steps:

To test equality of two lists:

- Count the number of items in each list. If the number is different then the two lists aren't equal. Otherwise, proceed.
- Compare the items in each list to each other one by one, checking that they are equal. If any pair is not equal, the two lists aren't equal. Otherwise, proceed.
- Having passed the previous tests, the two lists are equal.

Although this algorithm seems fairly simple, there are some aspects of it that make less than minimal assumptions, for instance:

- It is necessary to know how to *count* the number of items in a list. This requires appealing to the concept of number and counting.
- It is necessary to be able to *sequence* through the two lists.

Let's try to achieve a simpler expression of testing list equality by using the device of list decomposition. To start, we have what we will call the *fundamental list-dichotomy*.

fundamental list-dichotomy:

A list is either:

- *empty*, i.e. has no elements, or
- *non-empty*, i.e. has a first element and a rest

By *rest*, we mean the list consisting of all elements other than the first element. The empty list is shown as `[ ]`. It is possible for the rest of a non-empty list to be empty. Now we can re-cast the equality-testing algorithm using these new ideas:

To test equality of two lists:

a. If both lists are empty, the two lists are equal.
(proceeding only if one of the lists is not empty.)

b. If one list is empty, the two lists are not equal.
(proceeding only if both lists are non-empty.)

c. If the first elements of the lists are unequal, then the lists are not equal.
(proceeding only if the first elements of both are equal.)

d. The answer to the equality of the two lists is the same as the answer to whether the rests of the two lists are equal. This equality test in this box can be used.

Let us try this algorithm on two candidate lists: `[1, 2, 3]` and`[1, 2]`.

Equality of `[1, 2, 3]` and`[1, 2]`:

Case *a*. does not apply, since both lists are non-empty, so we move to case *b*.

Case *b*. does not apply, since neither list is empty, so we move to case *c*.

Case *c*. does not apply, since the first elements are equal, so we move to case d.

Case *d*. says the answer to the equality question is obtained by asking the original question on the rests of the lists: `[2, 3]` and `[2]`.

Equality of `[2, 3]` and `[2]`:

Cases *a*. through *c*. again don't apply.

Case *d*. says the answer to the equality question is obtained by asking the original question on the rests of the lists: `[3]` and `[ ]`.

Equality of `[3]` and `[ ]`:

Case *a*. does not apply.

Case *b*. does apply: the two lists are not equal.

The differences between this algorithm and the first one proposed are that this one does not require a separate counting step; it only requires knowing how to decompose a list by

taking its first and rest. Although counting is implied by the second algorithm, it can be "short-circuited" if the two lists are found to be unequal.

In the next chapter we will see how express such algorithms in more succinct terms.

2.5 List Manipulation

Here we introduce the interactive rex system, which will provide a test-bed for dealing with lists. When we type an expression to rex, followed by the semi-colon, rex shows the *value* of the expression. The reason for the semi-colon is that some expressions take more than one line. It would thus not work to use the end-of-line for termination of expressions. Other approaches, such as using a special character such as \ to indicate that the list is continued on the next line, are workable but more error-prone and less aesthetic (lots of \'s traded for one ;). For lists as introduced so far, the value of the list as an expression will be the list itself. Below, the boldface shows what is entered by the user.

```
rex > [2, 3, 5, 7];
[2, 3, 5, 7]

rex > [ ];
[ ]
```

As it turns out, list equality is built into rex. The equality operator is designated ==.

```
rex > [1, 2, 3] == [1, 2];
0
```

The value 0 is used by rex to indicate *false*: that the two lists are not equal. A value of 1 is used to indicate *true*.

```
rex > [ ] == [ ];
1
```

Suppose we wanted to access the first and rest of a list in rex. Two approaches are available: (*i*) use built-in functions *first* and *rest*; (*ii*) use list matching, which will be described in Chapter 4. Here we illustrate (*i*):

```
rex > first([1, 2, 3]);
1

rex > rest([1, 2, 3]);
[2, 3]
```

Correspondence with Box Diagrams

It is worthwhile at this point to interject at this time the correspondence between textual representation and box diagrams. This mainly helps to motivate the way of manipulating

lists that will be described presently, especially to justify the choice of primitive functions being used.

Every list in the open list model can be identified with a pointer to the first box in the box representation. The first element of the list is the item in the box and the list of the remaining elements is the pointer. The pointer is the special null pointer if, and only if, the rest of the list is empty. Figure 9 is meant to illustrate this correspondence.

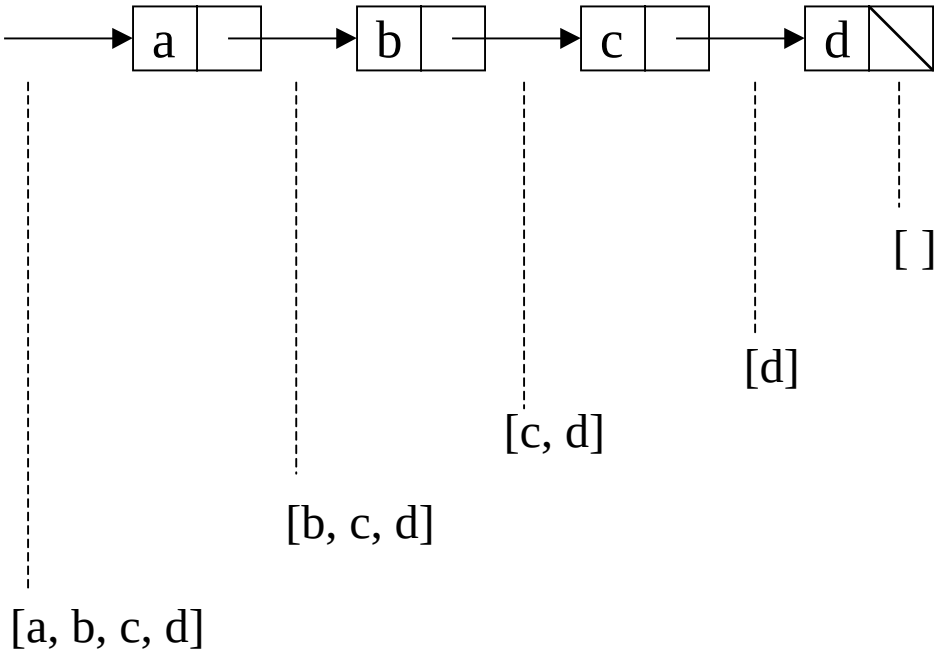


Figure 9: Equating pointers to lists in the open list representation

Intuitively, the `rest` operation is very fast in this representation. It does not involve creating any new list; it only involves obtaining the pointer from the first box in the list.

Identifiers in rex

An *identifier* in rex is a string of symbols that identifies, or stands for, some item such as a list or list element. As with many programming languages, a rex identifier must begin with a letter or an underscore, and can contain letters, digits, and underscores. The following are all examples of identifiers:

```
a     Temp  x1    _token_     move12      _456        ZERO
```

The most basic way to cause an identifier to stand for an item is to equate it to the item using the *define* operator, designated = (recall that == is a different operator used testing equality):

```
rex > a = 2;
1
```

This defines identifier `a` to stand for the number 2. In computer science it is common to say `a` is *bound to* 2 rather than "stands for" 2.

Bindings

If an identifier is bound to something, we say that it is *bound*, and otherwise it is *unbound*. By a *binding*, we mean a pairing of an identifier with its value, as in the binding `["a", 2]`, which is implicitly created inside the system by the above definition. Finally, an *environment* is a set of bindings. In other words, an environment collects together the meanings of a set of symbols.

Above, the reported value of 1 indicates that the definition was successful. We can check that *a* now is bound to 2 by entering the expression a by itself and having rex respond with the value.

```
rex > a;
2
```

On the other hand, if we try this with an identifier that is not bound, we will be informed of this fact:

```
rex > b;
*** warning: unbound symbol b
```

Forming New Lists

The same symbols used to decompose lists in rex can also be used to form new lists. Suppose that `R` is already bound to a list and `F` is an intended first element of a new list. Then `[F | R]` (again read `F` “followed by” `R`) denotes a new list with `F` as its first element and `R` as the rest of the new list:

```
rex > F = 2;
1

rex > R = [3, 4, 5];
1

rex > [F | R];
[2, 3, 4, 5]
```

One might rightfully ask why we would form a list this way instead of just building the final list from the beginning. The reason is that we will have uses for this method of list creation when either `F` or `R` are *previously* bound, such as being bound to data being supplied to a function. This aspect will be used extensively in the chapter on low-level functional programming.

As with our observation about rest being fast, creating a new list in this way (called "consing", short for "constructing") is also fast. It only entails getting a new box, initializing its item, and planting a pointer to the rest of the list.

Lists of Other Types of Elements

In general, we might want to create lists of any type of element that can be described. For example, a sentence could be a list of strings:

```
["This", "list", "represents", "a", "sentence"]
```

A list can also be of mixed types of elements, called a *heterogeneous* list:

```
["The", "price", "is", 5, "dollars"]
```

We can get the type of any element in rex by applying built-in function `type` to the argument:

```
rex > type(99);
integer

rex > type("price");
string

rex > type(["price", 99]);
list
```

The types are returned as strings, and normally strings are printed without quotes. There is an option to print them with quotes for development purposes. Notice that giving the type of a list as simply "list" is not specific as to the types of elements. Later we will see how to define another function, `deep_type`, to do this:

```
rex > deep_type(["price", 99]);
[string, integer]
```

Here the deep type of a list is the list of deep types of the individual elements.

Using Lists to Implement Sets

For many computational problems, the ordering of data and the presence of repetitions either are not to be important or are known not to occur at all. Then it is appropriate to think of the data as a *set* rather than a list. Although a list inherently imposes an ordering on its elements, we could simply choose to ignore the ordering in this case. Also, although repetitions are allowed in lists, we could either ignore repetitions in the case of sets or we could take care not to permit repetitions to enter into any lists we create.

Since abstraction has already been introduced as a tool for suppressing irrelevant detail, let us think of sets as a distinct abstraction, with open lists as one way to represent sets. An implementation, then, would involve giving list functions that represent the important functions on sets

For example, suppose we wanted to represent the set function `union` on lists. A use of this function might appear as follows:

```
rex > union([2, 3, 5, 7, 9], [1, 4, 9, 16]);
[2, 3, 5, 7, 1, 4, 9, 16]
```

Above the element occurs in both original sets, but only occurs once in the union. Similarly, we can represent the set function `intersection`:

```
rex > intersection([2, 3, 5, 7, 9], [1, 4, 9, 16]);
[9]
```

This result is correct, since `9` is the only element occurring in both sets. These functions are not built into rex; we will have to learn how to define them.

Representation Invariants

Our discussion of representing sets as lists provides an opportunity to mention an important point regarding representations. Implementing functions on sets represented as lists is more economical (is faster and takes less memory space) if we can assume that the list used to represent the set contains no duplicates. A second reason for maintaining this assumption is that the implementation of certain functions becomes simpler. For example, consider a function `remove_element` that removes a specified element from a set, provided that it occurs there at all. Without the assumption that there are no duplicates, we would need to check the entire list to make sure that all occurrences of the element were removed. On the other hand, if we can assume there are no duplicates, then we can stop when the first occurrence of the element in question has been found.

Assumptions of this form are called *representation invariants*. Their articulation is part of the implementation and each function implemented has the obligation of making sure that the invariant is true for the data representation for the value it returns. At the same time, each function enjoys the use of the assumption to simplify its work, as described in the preceding paragraph.

2.6 Lists of Lists

Lists are not restricted to having numbers as elements. A list can have lists as elements:

```
[ [2, 4, 8, 16], [3, 9, 27], [5, 25], [7] ]
```

The elements of the outer list can themselves be lists.

```
[ [ [1, 2, 3], [4, 5, 6] ], [ [7, 8, 9], [ ] ] ]
```

Pursuing our sets-implemented-as-lists idea, a list of lists could be used to represent a set of sets. For example, the function `subsets` returns all subsets of its argument, interpreted as a set:

```
rex> subsets([1, 2, 3]);
[ [ ], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3] ]
```

This function is not built in, but we will see how to define it in a later chapter.

Length of Lists

The *length* of a list is the number of elements in the list:

```
rex > length([2, 4, 6, 8]);
4
```

When the list in question has lists as elements, *length* refers to the number of elements in the outer or *top-level* list, rather than the number of elements that are not lists:

```
rex > length([ [2, 4, 8], [3, 9], [5], [ ] ]);
4
```

The number of elements that are not lists is referred to as the *leafcount*:

```
rex > leafcount([ [2, 4, 8], [3, 9], [5], [ ] ]);
6
```

Exercises

1. • Which pairs of lists are equal?

   a. `[1, 2, 3]` *vs.* `[2, 1, 3]`

   b. `[1, 1, 2]` *vs.* `[1, 2]`

   c. `[1, [2, 3] ]` *vs.* `[1, 2, 3]`

   d. `[1, [2, [3] ] ]` *vs.* `[1, 2, 3]`

2. •• Execute the equality testing algorithm on the following two lists: `[1, [2, 3], 4]`, and `[1, [2], [3, 4]]`.

3. ••• As we saw above, although lists are not the same as sets, lists can be used to represent sets. Present an equality testing algorithm for two sets represented as lists.

4. • What is the length of the list `[1, [2, 3], [4, 5, 6]]`? What is its leafcount?

2.7 Binary Relations, Graphs, and Trees

A *binary relation* is a set of ordered pairs. Since both sets and pairs can be represented as lists, a binary relation can be represented as a list of lists. For example, the following list represents the relation *father_of* in a portion of the famous Kennedy family:

```
[     [“Joseph”, “Bobby”],
      [“Joseph”, “Eunice”],
      [“Joseph”, “Jean”],
      [“Joseph”, “Joe Jr.”],
      [“Joseph”, “John”],
      [“Joseph”, “Pat”],
      [“Joseph”, “Ted”]
      [“Bobby”,  “David”],
      [“Bobby”,  “Joe”]
      [“John”,   “Caroline”],
      [“John”,   “John, Jr.”],
      [“Ted”,    “Edward”] ]
```

Here Joseph, for example, is listed as the first element in many of the pairs because he had many children. Notice also that no individual is listed as the second element in more than one pair. This reflects the fact that an individual cannot have more than one father.

Graphical Representation of Binary Relations

Binary relations are sometimes rendered more understandable by depiction as a *directed graph*: the nodes in the graph are items being related; arrows in the graph represent the pairs in the relation. For example, the preceding *father_of* relation would be shown as below:

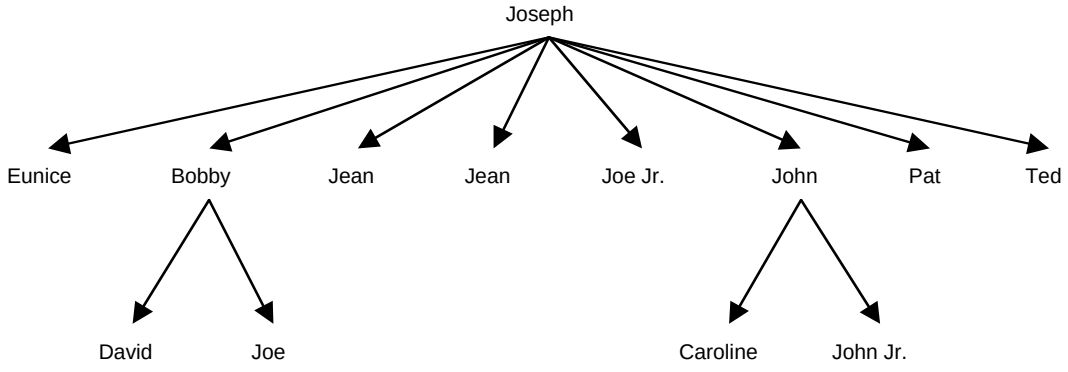


Figure 10: a father_of relation

Notice that each arrow in the directed graph corresponds to one ordered pair in the list. This particular directed graph has a special property that makes it a *tree*. Informally, a graph is a tree when its nodes can be organized hierarchically.

Let's say that node *n* is a *target* of node *m* if there is an arrow from *m* to *n*. (In the above example, the targets happen to represent the children of the individual.) Also, the *target set* of a node *m* is the set of nodes that are targets of node *m*. For example, above the target set of `"Bobby"` is {`"David"`, `"Joe"`}, while the target set of `"Joe"` is the empty set.

Let's say the nodes *reachable* from a node consist of the targets of the node, the targets of those targets, and so on, all combined into a single set. For example, the nodes reachable from `"Joseph"` above are all the nodes other than `"Joseph"`. If the nodes reachable from some node include the node itself, the graph is called *cyclic*. If a graph is not cyclic then it is called *acyclic*. Also, a node that is not a target of any node is called a *root* of the graph. In the example above, there is one root, `"Joseph"`. A node having no targets is called a *leaf*. Above, `"Eunice"`, `"Joe"`, and `"Caroline"` are examples of leaves.

Given these concepts, we can now give a more precise definition of a tree.

> A *tree* is a directed graph in which the following three conditions are present:
>
> - The graph is acyclic
> - There is exactly one root of the graph.
> - The intersection of the target sets of any two different nodes is always the empty set.

If we consider any node in a tree, the nodes reachable from a given node are seen to form a tree in their own right. This is called the *sub-tree* determined by the node as root. For example, above the node `"Bobby"` determines a sub-tree containing the nodes {`"Bobby"`, `"David"`, `"Joe"`} with `"Bobby"` as root.

Hierarchical-List Representation of Trees

A list of pairs, while simple, is not particularly good at allowing us to spot the structure of the tree while looking at the list. A better, although more complicated, representation would be one we call a *hierarchical list*:

To represent a tree as a hierarchical list:

- The root is the first element of the list.
- The remaining elements of the list are the sub-trees of the root, each represented as a hierarchical list.

These rules effectively form a representation invariant.

This particular list representation has an advantage over the previous list-of-pairs representation in that it shows more clearly the *structure* of the tree, *viz*. the children of each parent. For the tree above, a list representation using this scheme would be:

```
["Joseph",
   ["Eunice"],
   ["Bobby",
      ["David"],
      ["Joe"] ],
   ["Jean"],
   ["Joe Jr."],
   ["John",
      ["Caroline"],
      ["John Jr."] ],
   ["Pat"],
   ["Ted",
      ["Edward"] ] ]
```

Note that in this representation, a leaf will always show as a list of one element, and every such list is a leaf. Also, the empty list only occurs if the entire tree is empty; the empty list representing a sub-tree would not make sense, because it would correspond to a sub-tree with no root.

As a variant on the hierarchical list representation, we may adopt the convention that leaves are not embedded in lists. If we do this for the current example, we would get the representation

```
["Joseph",
   "Eunice",
   ["Bobby",
      "David",
      "Joe"],
   "Jean",
   "Joe Jr.",
   ["John",
      "Caroline",
      "John Jr."],
   "Pat",
   ["Ted",
      "Edward"] ]
```

which has the virtue of being slightly less cluttered in appearance. From such a list we can reconstruct the tree with labels on each node. Accordingly, we refer to this as the *labeled-tree interpretation* of a list.

Another example that can exploit the hierarchical list representation of a tree abstraction is that of a *directory structure* as used in computer operating systems to manage collections of files. A common technique is to organize files in "directories", where each directory has an associated collection of files. A file space of this form can be thought of as a list of lists. Typically directories are not required to contain only files; they can contain other directories, which can contain files or directories, etc. in any mixture. So a user's space might be structured as follows (we won't put string quotes around the names, to avoid clutter; however, if this were rex, we would need to, to avoid confusing the labels with variables):

```
[ home_directory,
  mail,
  [ mail_archive, current_mail, old_mail],
  [ programs,
    [ sorting, quicksort.rex, heapsort.rex],
    [ searching, depth_first.rex, breadth_first.rex]
  ],
  [ games, chess, checkers, tic-tac-toe]
]
```

representing the tree in Figure 11.

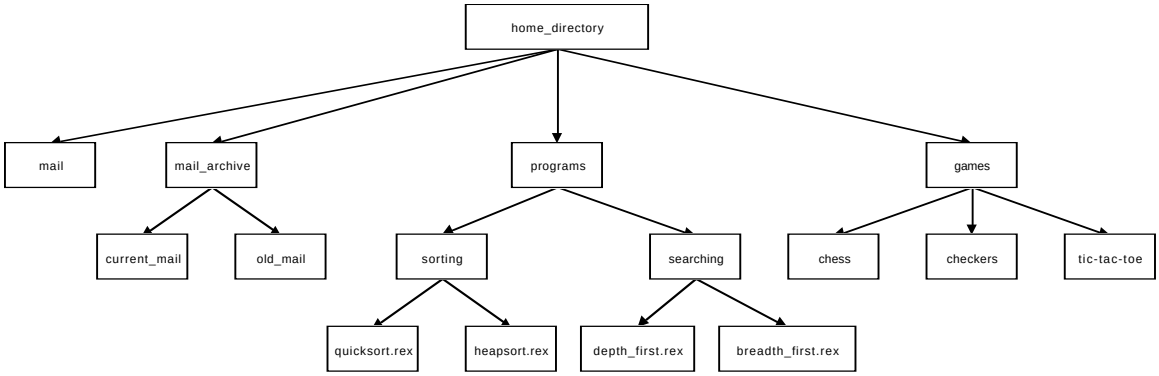


Figure 11: A directory tree

Unlabelled vs. Labeled Trees as Lists

Sometimes it is not necessary to carry information in the non-leaf nodes of the tree. In such cases, we can eliminate this use of the first list element, saving one list element. We call this the *unlabeled-tree interpretation* of the list. The only list elements in such a representation that are not lists themselves are the leaves of the tree. In this variation, the list

```
[a, [b, c], [d, [e, f], g], h]
```

represents the following tree (as is common practice, we often omit the arrow-heads, with the understanding that they all point the same direction, usually downward):

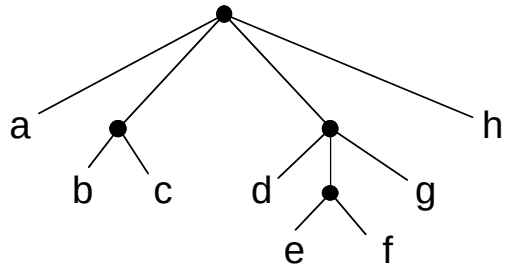


Figure 12: The list `[a, [b, c], [d, [e, f], g], h]` as an unlabeled tree

In the hierarchical list representation described previously, the first element in a list represents the label on a root node. This tree would have a similar, but slightly different shape tree (assuming the convention that we don't make lists out of single leaves):

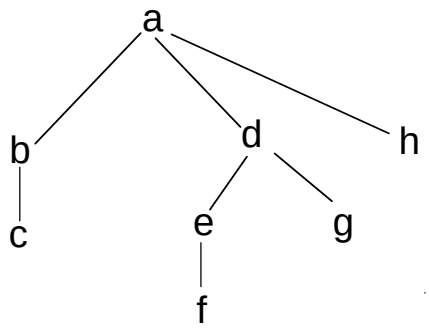


Figure 13: The list `[a, [b, c], [d, [e, f], g], h]` as a labeled tree

Clearly, when we wish to interpret a list as a tree, we need to say which interpretation we are using. We summarize with the following diagrams:

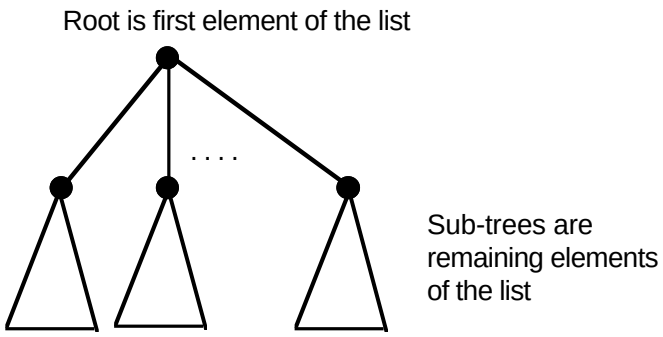


Figure 14: Hierarchical list representation of a labeled tree

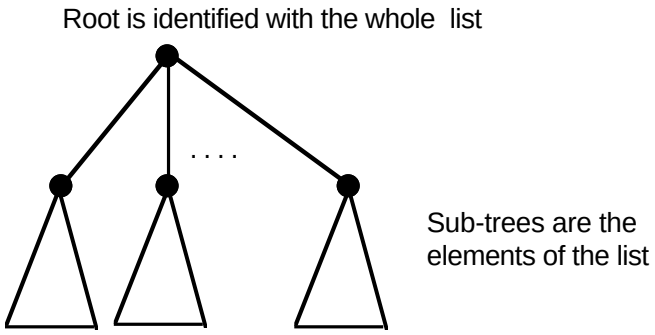


Figure 15: Interpretation of a list as an unlabeled tree

Note that the *empty list* for an unlabeled tree would be the empty tree, or the slightly anomalous "non-leaf" node with no children (since leaves have labels in this model).

Binary Trees

In the *binary-tree representation* of a list, we recall the view of a list as [*First* | *Rest*] where, as usual, *First* is the first element of the list and *Rest* is the list consisting of all but the first element. Then the following rules apply:

***Binary tree representation as lists*:**

- The **empty list** is shown as a leaf [ ].
- An **atom** is shown as a leaf consisting of the atom itself.
- A list [*First* | *Rest*] is shown as a node with two subtrees, one tree corresponding to *First*, the other the tree corresponding to *Rest*.

A representation invariant here is the right-hand branch is always a list, never an atom.

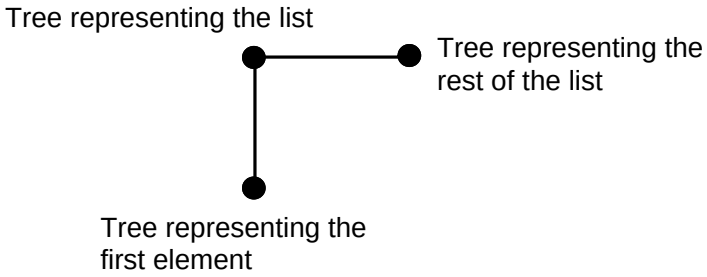


Figure 16: Binary tree representation of a list

In particular, a list of n (top-level) elements can easily be drawn as a binary tree by first drawing a "spine" with n sections:

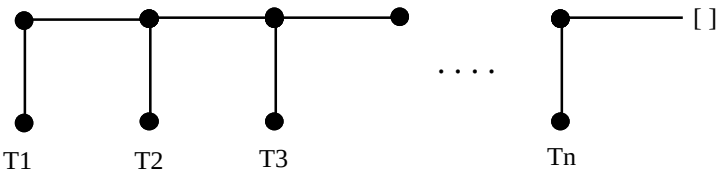


Figure 17: Binary tree representation of a list with top-level elements T1, T2, ..., Tn

The elements on the branches of the spine can then drawn in.

For the preceding example, `[a, [b, c], [ ], [d, [e, f], g], h]`, these rules give us the following tree, where the subtrees of a node are below, and to the right of, the node itself.

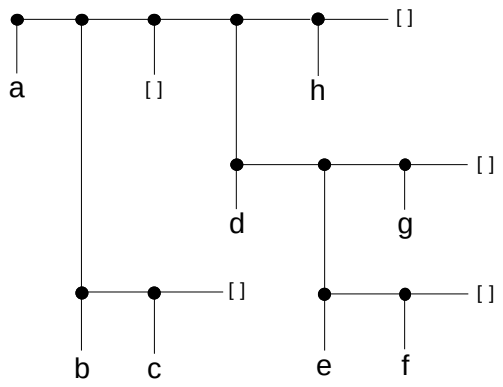


Figure 18: The binary tree representation of a list
`[a, [b, c], [ ], [d, [e, f], g], h]`

Sometimes preferred is the rotated view where both subtrees of a node are below the node. The same tree would appear as follows.

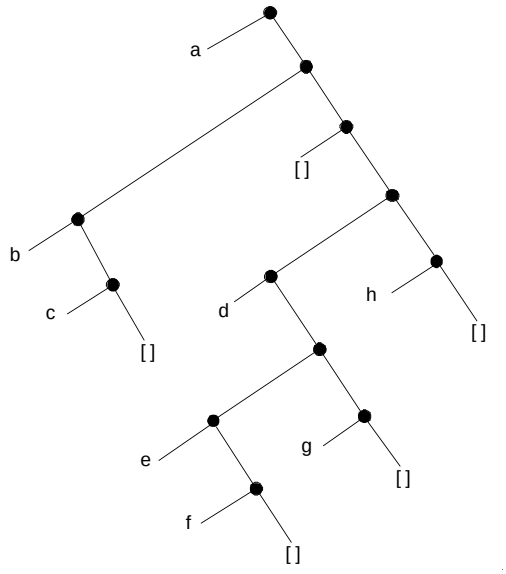


Figure 19: A rotated rendering of the binary tree representation of a list
`[a, [b, c], [ ], [d, [e, f], g], h]`

The binary tree representation corresponds to a typical *internal representation of lists* in languages such as rex, Lisp, and Prolog. The arcs are essentially *pointers*, which are represented as internal memory addresses.

In a way, we have now come a full cycle: we started by showing how trees can be represented as lists, and ended up showing how lists can be represented as trees. The reader may ponder the question: Suppose we start with a tree, and represent it as a list,

then represent that list as a binary tree. What is the relationship between the original tree and the binary tree?

Quad-Trees

An interesting and easy-to-understand use of trees is in the digital representation of images. Typically images are displayed based on a two-dimensional array of *pixels* (picture elements), each of which has a black-white, color, or gray-scale value. For now, let us assume black and white. An image will not usually be homogeneous, but will instead consist of regions that are largely black or largely white. Let us use the following 16-by-16 pixel image of a familiar icon as an example:

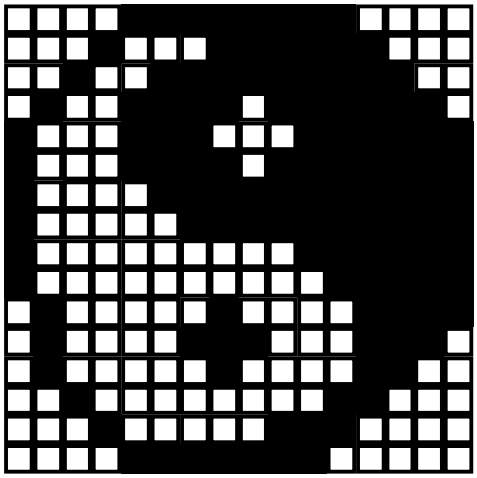


Figure 20: 16 x 16 pixel image of yin-yang icon

The quad-tree is formed by recursively sub-dividing the image into four quadrants, sub-dividing those quadrants into sub-quadrants, and so on, until a quadrant contains only black or only white pixels. The sub-divisions of a region are the four sub-trees of a tree representing the entire region.

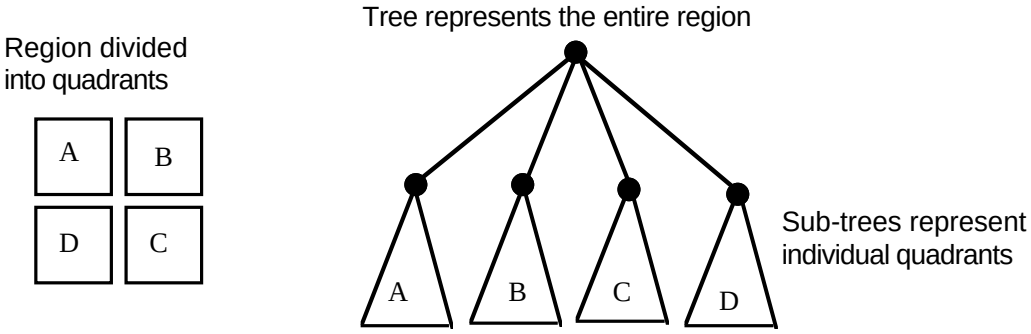


Figure 21: Quad tree recursive sub-division pattern

The first level of sub-division is shown below, and no quadrant satisfies the stopping condition. After the next level sub-division, several sub-quadrants satisfy the stopping condition and will thus be encoded as 0.

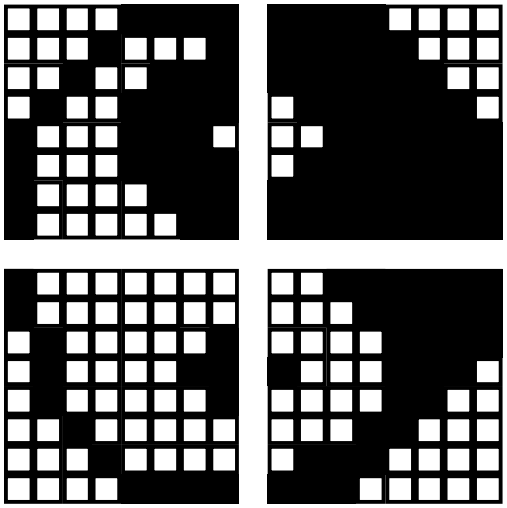


Figure 22: The yin-yang image showing the first level of quad-tree sub-division

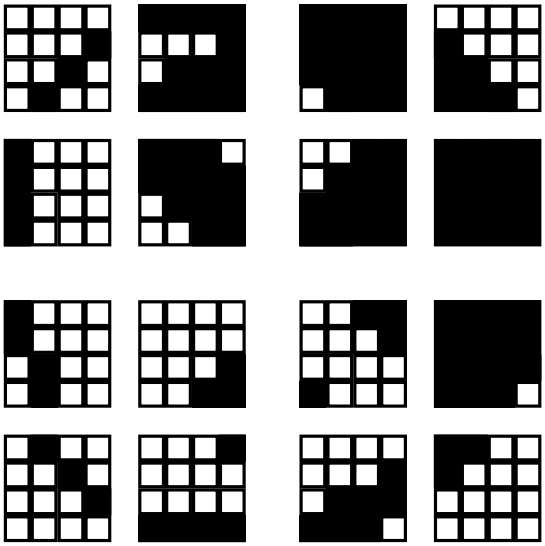


Figure 23: The image after the second level of quad-tree sub-division. Note that one quadrant is all black and will not be further sub-divided.

Continuing in this way, we reach the following set of sub-divisions, in which each quadrant satisfies the stopping condition:

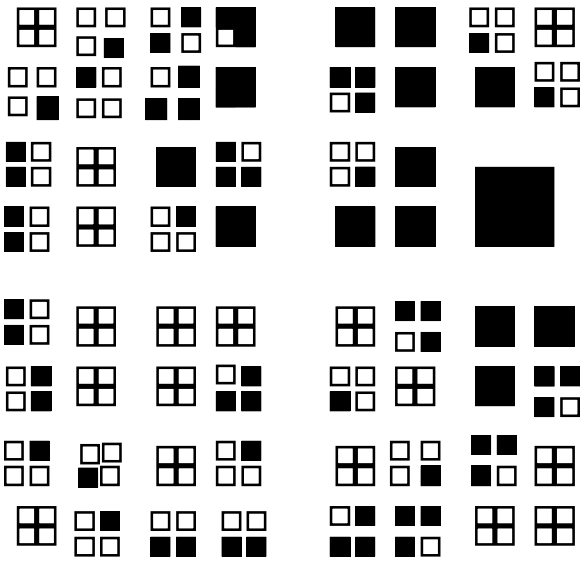


Figure 24: The yin-yang icon sub-divided until all sub-grids are either all black or all white

Encoding the sub-quadrants as 0 for white, 1 for black, we get the quad-tree in Figure 25.

A linear encoding of the quad-tree, using the symbol set { '0', '1', '[', ']' }, reading the quadrants left-to-right and upper-to-lower, is:

```
[[[0[0001][0001][1000]][[0110][1101][0111]1][[1010]0[1010]0][1[1011][01
00]1]][[11[1101]1][[0111]01[0001]][[0001]011]1][[[1010]0[0101]0][000[01
11]][[0100][0010]0[0100]][0[0100][0011][0011]][[0[1101][0111]0][111[111
0]][0[0001][0111][1110]][[1110]000]]]
```

We need to accompany this linear encoding by a single number that indicates the depth of a single pixel in the hierarchy. This is necessary to know the number of pixels in a square area represented by an element in the quad-tree. For example, in the tree above, the depth is 4. An element at the top level (call this level 1) corresponds to a 8-by-8 pixel square. An element at level 2, corresponds to a 4-by-4 area, an element at level 3 to a 2-by-2 area, and an element at level 4 to a single pixel.

The linear quad-tree encoding can be compared in length to the *raster encoding*, that is the array of pixels with all lines concatenated:

```
00001111111100000001000111110000010011111111111000100111101111111010001
11100011111111001111011111111110000111111111111000001111111111000000000
11111110000000000111110100000100001111010000111000111001000001000011000
01000000011000000100000111000000001111111100000
```

Here it is not necessary to know the depth, but instead we need to know the row size. Since both of these are printed in equal-space fonts, based purely on length, modest compression can be seen in the quad-tree encoding for this particular image. In images with larger solid black or solid white areas, a significantly greater compression can be expected.

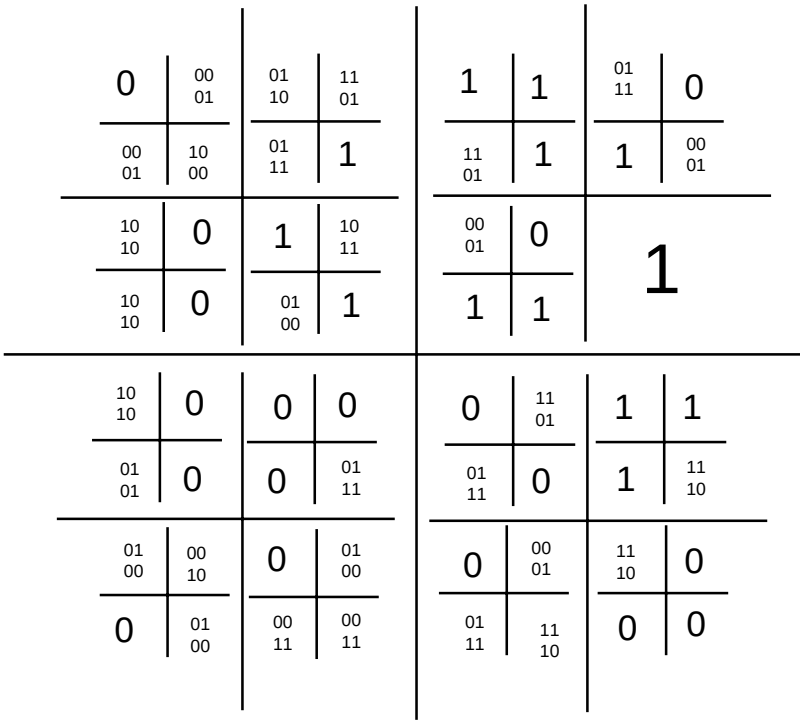


Figure 25: Quad-tree for the yin-yang icon in two dimensions

Quad trees, or their three-dimensional counterpart, oct-trees, have played an important role in algorithm optimization. An example is the well-known Barnes-Hut algorithm for doing simulations of N bodies in space, with gravitational attraction. In this approach, oct-trees are used to structure the problems space so that clusters of bodies can be treated as single bodies in certain computational contexts, significantly reducing the computation time.

Tries

A *trie* is a special kind of tree used for information re*trie*val. It exploits indexability in lists, or arrays, which represent its nodes, in order to achieve more efficient access to the information stored. A trie essentially implements a function from character strings or numerals into an arbitrary domain. For example, consider the function `cities` that gives

the list of cities corresponding to a 5-digit ZIP code. Some sample values of this function are:

```
cities(91711) = ["Claremont"]
cities(91712) = [ ]
cities(94305) = ["Stanford", "Palo Alto"]
cities(94701) = ["Berkeley"]
```

The value of `cities` for a given argument could be found by searching a list of the form

```
[
 [91711, "Claremont"],
 [91712],
 [94305, "Stanford", "Palo Alto"],
 [94701, "Berkeley"],
   ....
]
```

(called an *association list* , and elaborated on in the next chapter). However, this would be slower than necessary. A quicker search would result if we used a tree with each non-leaf node having ten children, one for each digit value. The root of the tree would correspond to the first digit of the ZIP code, the second-level nodes would correspond to the second digit, and so on. For any digit combination that cannot be continued to a proper ZIP code, we could terminate the tree construction with a special empty node [] at that point. The overall trie would appear as follows:

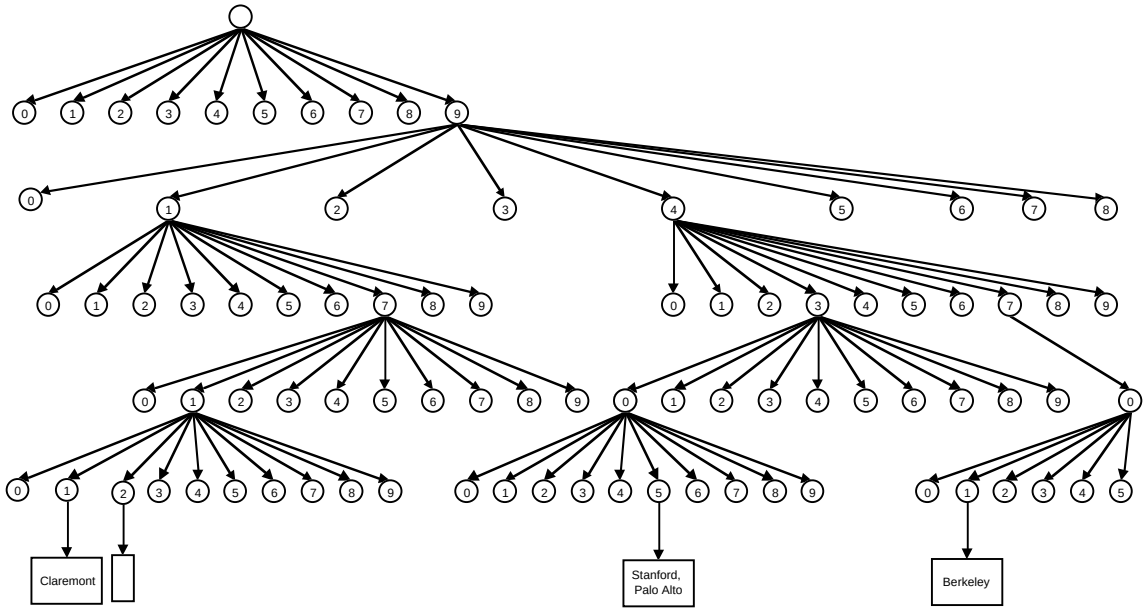


Figure 26: Sketch of part of a trie representing assignment of sets of cities to zip codes.

General Graphs as Lists

If we omit, in the definition of tree, the condition “there is exactly one root”, thus allowing multiple roots (note that there will always be at least one root if the graph is acyclic), we would define the idea of a *forest* or set of trees. In general, taking the root away from a tree will leave a forest, namely those trees with roots that were the targets of the original root.

We can also use the idea of target set to clarify the structure of an *arbitrary* directed graph: Represent the graph as a list of lists. There is exactly one element of the outer list for each node in the graph. That element is a list of the node followed by its targets. For example, Figure 27 shows a directed graph that is not a tree.

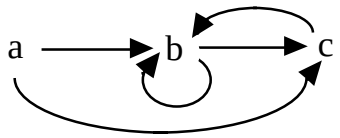


Figure 27: A graph that is not a tree.

The corresponding list representation would be:

```
[ [a, b, c],
  [b, b, c],
  [c, b] ]
```

We have gone through a number of different list representations, but certainly not exhausted the possibilities. It is possible to convert any representation into any other. However, the choice of a working representation will generally depend on the algorithm; some representations lead to simpler algorithms for a given problem than do other representations. The computer scientist needs to be able to:

- Determine the best information representation and algorithm for a given data abstraction and problem.
- Be able to convert from one representation into another and to write programs that do so.
- Be able to present the information abstraction in a means that is friendly to the user. This includes various tabular and graphical forms.

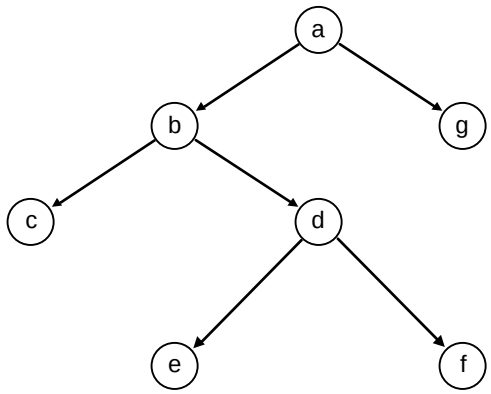
There is no one form that is universally the most correct or desirable. Each application and user community will have its own ways of viewing information and it may even be necessary to provide several views of the same information and the ability of software to switch between them.

One thing is certain: graphs and their attendant information structures are ubiquitous in computer applications. Our entire world can be viewed as sets of intersecting networks of items that connect to one another in interesting ways. The directed graph is the typical way in which a computer scientist would view these connections. To give a modern example, consider the "World-Wide Web". This is an information structure that is distributed over a pervasive network of computers known as the *Internet*. A node in the World-Wide Web is a document containing text and graphics. Each node has a symbolic *address* that uniquely identifies it among all other documents. Typically a document will refer to one or more other documents in the form of *hypertext links*. These links specify the addresses of other nodes. Thus they correspond directly to the arcs in a directed graph. When the user clicks on the textual representation of a link using a *web browser* the browser changes its focus to the target of that particular link.

The World-Wide Web is thus a very large directed graph. It is not a tree, since it is cyclic: nodes can refer to each other or directly to themselves. Moreover, it is common for the target sets of two nodes to have a non-empty intersection. Finally there is no unique root. Thus all three of the requirements for a directed graph to be a tree are violated.

Exercises

1. •• What are some ways of representing the following tree as a list:



2. •• Draw the trees corresponding to the interpretation of the following list as (i) a labeled tree, (ii) an unlabeled tree, and (iii) a binary tree:

```
[1, [2, 3, 4, 5], [6, [7, [8, 9] ] ] ]
```

3. ••• What lists have interpretations as unlabeled trees but not as labeled trees? Do these lists have interpretations as binary trees?

4. ••• Identify some fields outside of computer science where directed graphs are used. Give a detailed example in each case.

5. •• The decoding of the *Morse code* alphabet can be naturally specified using a trie. Each character is a sequence of one of two symbols: dash or dot. For example, an 'a' is dot-dash, 'b' is dash-dot-dot-dot, and so on. Consult a source, such as a dictionary, which contains a listing of the Morse code, then construct a decoding trie for Morse code.

6. ••• A *tune identifier* based on the trie concept can be developed to provide a simple means of identifying unknown musical tunes. With each tune we may associate a signature that describes the pattern of intervals between the initial notes, say the first ten, in the tune. The symbol D indicates that the interval is a downward one, U indicates it is upward, and S represents that the note is the same. For example, the tune of *The Star Spangled Banner* has the signature DDUUU UDDDU, corresponding to its start:

```
O - oh say can you see, by the dawn's ear-ly
  D   D   U   U   U    U  D   D      D   U
```

   Some other tunes and their signatures are:

```
Bicycle Built for Two                    DDDUU UDUDU
Honeysuckle Rose                         DDUUU DDUUU
California Here I come                   SSSUD DDSSS
One-Note Samba                           SSSSS SSSSS
```

   Show how a trie can be used to organize the songs by their signatures.

7. ••• The three-dimensional counterpart of a pixel is called a *voxel* (for volume element). Describe how a 3-dimensional array of voxels can be compacted using the idea of an *oct-tree* (eight-way tree).

8. •• An *oriented directed graph* is like a directed graph, except that the targets have a specific order, rather than just being a set. Describe a way to represent an oriented directed graph using lists.

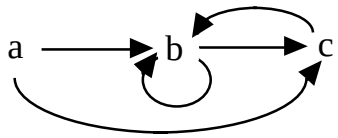
9. ••• A *labeled directed graph* is a directed graph with labels on its arrows. Describe a way to represent a labeled directed graph using lists.

10. •• Draw enough nodes of the World-Wide Web to show that it violates all three of the tree requirements.

2.8 Matrices

The term *matrix* is often used to refer to a table of two dimensions. Such a table can be represented by a list of lists, with the property that all elements of the outer list have the same length.

One possible use of matrices is as yet another representation for directed graphs. In this case, the matrix is called a *connection matrix* (sometimes called *adjacency matrix*) and the elements in the matrix are 0 or 1. The rows and columns of the matrix are indexed by the nodes in the graph. There is a 1 in the i*th* row j*th* column of the matrix exactly when there is a connection from node i to node j. For example, consider the graph below, which was discussed earlier:



The connection matrix for this graph would be:

	a	**b**	**c**
a	0	1	1
b	0	1	1
c	0	1	0

Note that there is one entry in the connection matrix for every arrow. We can use the connection matrix to quickly identify nodes with certain properties:

A root corresponds to a column of the matrix with no 1's.

A leaf corresponds to a row of the matrix with no 1's.

As a list of lists, the connection matrix would be represented

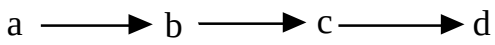
```
[ [0, 1, 1],
  [0, 1, 1],
  [0, 1, 0] ]
```

A related matrix is the *reachability matrix*. Here there is a 1 in the i*th* row j*th* column of the matrix exactly when there is a directed *path* (one following the direction of the arrows) from node i to node j. For the current example, the reachability matrix would be

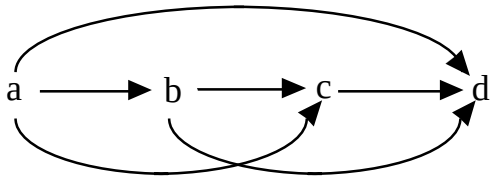
	a	**b**	**c**
a	0	1	1
b	0	1	1
c	0	1	1

From the reachability matrix, we can easily determine if the graph is cyclic: It is iff there is a 1 on the (upper-left to lower-right) diagonal, indicating that there is a path from some node back to itself. In the present example, both rows b and c have 1's in their diagonal elements.

Notice that the reachability matrix, since it does represent a set of pairs, also corresponds to a relation in its own right. This relation is called the *transitive closure* of the original relation. We typically avoid drawing graphs of transitive closures because they tend to have a lot more arrows than does the original graph. For example, the transitive closure of this simple graph



would be the more complex graph



As a second example of a matrix, we could use a table of 0's and 1's to represent a black-and-white *image*. Of course, the entries in a matrix aren't limited to being 0's and 1's. They could be numeric or even symbolic expressions.

Apparently we can also represent tables of *higher dimension* using the same list approach. For every dimension added, there would be another level of nesting.

Exercises

1. • Show a list of lists that does not represent a matrix.

2. •• Suppose a binary relation is given as the following list of pairs. Draw the graph. Construct its connection matrix and its reachability matrix. What is the list of pairs corresponding to its transitive closure?

```
[ [1, 2], [2, 3], [2, 4], [5, 3], [6, 5], [4, 6] ]
```

3. ••• Devise a method, using connection matrices, for determining whether a graph is acyclic.

4. •• Consider the notion of a labeled directed graph as introduced earlier (a graph with labels on its arrows). How could you use a matrix to represent a labeled directed graph? How would the representation of this matrix as a list differ from previous representations of graphs as lists?

5. ••• Devise a method for computing the connection matrix of the transitive closure of a graph, given the graph's connection matrix.

2.9 Undirected graphs

An *undirected graph* is a special case of a directed graph (and not the other way around, as you might first suspect). An undirected graph is usually presented as a set of nodes with lines connecting selected nodes, but with no arrow-heads on the lines. In terms of a directed graph, i.e. a set of pairs of nodes, a line of an undirected graph connecting a node `n` to a node `m` represents *two* arrows, one from node `n` to `m` and another from node `m` to `n`. Thus the connection matrix for an undirected graph is always *symmetric* about the diagonal (that is, if there is a 1 in row i column j, then there is a 1 in row j column i).

On the left below is an undirected graph, and on the right the corresponding directed graph.

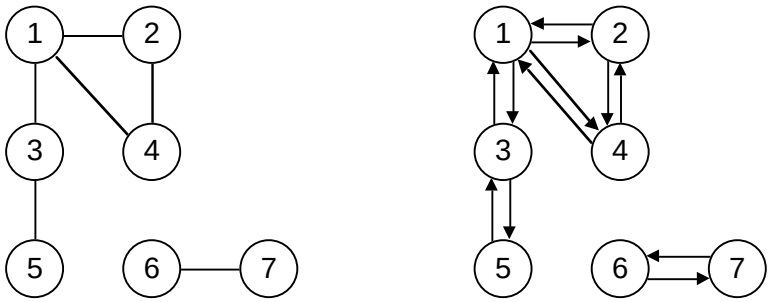


Figure 28: An undirected graph and its corresponding directed graph

In summary, an undirected graph may be treated as an abbreviation for a directed graph. All the representational ideas for directed graphs thus apply to undirected graphs as well.

Exercises

1. • Show the list of pairs representing the undirected graph above. Show its connection matrix.

2. •• Devise an algorithm for determining whether a directed graph could be represented as an undirected graph.

3. ••• The transitive closure of an undirected graph has an unusual form. What is it? Devise a more efficient way to represent such a transitive closure as a list based on its properties.

2.10 Indexing Lists vs. Arrays

We will *index* elements of a list by numbering them starting from 0. Given a list and index, an element of the list is determined uniquely. For example, in

```
["Canada", "China", "United Kingdom", "Venezuela"]
```

the index of `"China"` is 1 and that of `"Venezuela"` is 3. The important thing to note is that indices are not an explicit part of the data. Instead, they are implicit: the item is determined by "counting" from the beginning.

In rex, an item at a specific index in the list may be determined by treating that list as if it were a function: For example

```
rex > L = ["Canada", "China", "United Kingdom", "Venezuela"];
1

rex > L(2);
United Kingdom
```

Indexing lists in this way is not something that should be done routinely; for routine indexing, *arrays* are a better choice. This is now getting into representational issues somewhat, so we only give a preview discussion here.

The time taken to index a *linked list* is proportional to the index.

On the other hand, due to the way in which computer memory is constructed, the time taken to index an array is nearly constant. We refer to the use of this fact as the *linear addressing principle*.

linear addressing principle: The time taken to index an *array* is constant, independent of the value of the index.

We use the modifier "linear" to suggest that we get the constant-time access only when elements can be indexed by successive integer values. We cannot expect constant-time access to materialize if we were to instead index lists by arbitrary values. (However, as will be seen in a later chapter, there are ways to come close to constant-time access using an idea known as *hashing*, which maps arbitrary values to integers.)

In addition to lists, arrays are available in rex. For example, a list may be converted to an array and vice-versa. Both are implementations of the abstraction of *sequence*, differing primarily in their performance characteristics. Both can be indexed as shown above. We will discuss this further in the next chapter.

2.11 Structure Sharing

Consider lists in which certain large items are repeated several times, such as

```
[ [1, 2, 3], 4, [1, 2, 3], 5, [1, 2, 3], 6]
```

In the computer we might like to normalize this structure so that space doesn't have to be taken representing the repeated structure multiple times. This is especially pertinent if we do not tend to modify any of the instances of the recurring list. Were we to do that, we'd have to make a decision as to whether we wanted the structures to be shared or not.

A way to get this kind of sharing in rex is to first bind an identifier to the item to be shared, then use the identifier in the outer structure thus:

```
rex > shared = [1, 2, 3];
1

rex > outer = [shared, 4, shared, 5, shared, 6];
1

rex > outer;
[[1, 2, 3], 4, [1, 2, 3], 5, [1, 2, 3], 6]
```

Within the computer, sharing is accomplished by *references*. That is, in the place where the shared structure is to appear is a data item that *refers* to the *apparent* item, rather than being the item itself. From the user's viewpoint, it appears that the item itself is where the reference is found. In rex, there is no way to tell the difference.

A reference is usually implemented using the lower-level notion of a *pointer*. Pointers are based on another use of the linear addressing principle: all items in memory are stored in what amounts to a very large array. This array is not visible in a functional language such as rex, but it is in languages such as C. A pointer or reference is an index into this large array, so we can locate an item given its reference in nearly constant time.

In programming languages, references typically differ from pointers in the following way: a pointer must be prefixed or suffixed with a special symbol (such as prefixed with a * in C or suffixed with a ^ in Pascal) in order to get the value being referenced. With references, no special symbol is used; references *stand for* the data item to which they refer. Most functional languages, as well as Java, have no pointers, although they may have references, either explicit or implicit.

In addition to sharing items in lists, *tails* of lists can be shared. For example, consider

```
rex > tail = [2, 3, 4, 5];
1

rex > x = [0 | tail];
1

rex > y = [1 | tail];
```

```
1

rex > x;
[0, 2, 3, 4, 5]

rex > y;
[1, 2, 3, 4, 5]
```

Although `x` and `y` appear to be distinct lists, they are sharing the list tail as a common *rest*. This economizes on the amount of storage required to represent the lists.

As we saw earlier, lists have a representation as unlabeled trees. However, a tree representation is not appropriate for indicating sharing. For this purpose, a generalization of a tree called a *dag* (for *directed acyclic graph*) helps us visualize the sharing.

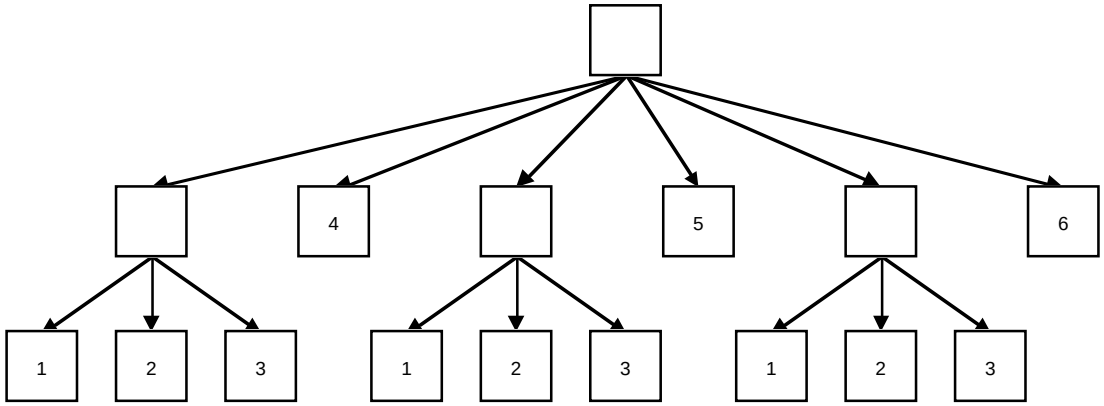


Figure 29: A tree with sharable sub-trees.

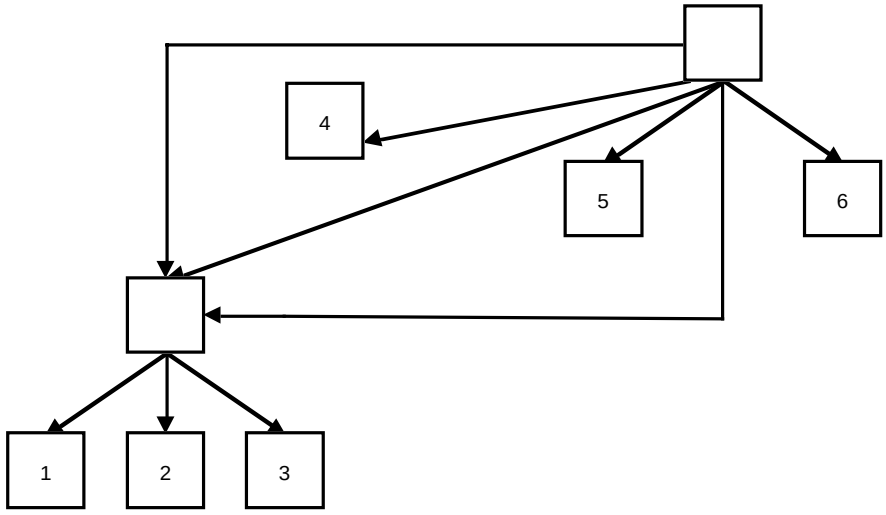
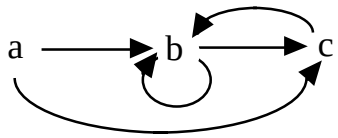


Figure 30: Showing shared structure using a dag.

Strictly speaking, the above tree and dag are not simply directed graphs, but instead directed *oriented* graphs, meaning that *ordering* and *repetition* of targets is important. If it were purely a graph, then the targets would be regarded as a set and repetitions wouldn't matter, defeating our use of trees. These fine distinctions will not be belabored, since we are using lists to represent them anyway and we are trying to avoid building up a vocabulary with too many nuances.

Graphs as well as dags can be represented using *references*. This results in a more compact representation. In order to demonstrate such a use of references in rex, we will introduce an identifier for each node in the graph. The identifier will be bound to a list consisting of the name of the node followed by the values of the identifiers of the targets of the node. Consider for example the graph introduced earlier:



We would like to define the three node lists simultaneously. To a first approximation, this would be done by the rex statement:

```
[a, b, c] = [ ["a", b, c], ["b", b, c], ["c", b] ];
```

There is a problem with this however; the right-hand side is evaluated *before* binding the variables on the left-hand side, but the symbols `a`, `b`, and `c` on the right are not yet defined. The way to get around this is to *defer* the use of those right-hand side symbols. In general, *deferred binding* means that we use a symbol to designate a value, but give its value later. Deferring is done in rex by putting a `$` in front of the expression to be deferred, as in:

```
rex > a = ["x", $b, $c];
1

rex > b = ["y", $b, $c];
1

rex > c = ["z", $b];
1
```

While these equations adequately define a graph information structure inside the computer, there is a problem if we try to print this structure. The printing mechanism tries to display the list structure as if it represented a tree. If the graph above is interpreted a tree, the tree would be infinite, since node `a` connects to `b`, which connects to `b`, which connects to `b`, ... Our rex interpreter behaves as follows:

```
rex > a;
[x, [y, [y, [y, [y, [y, [y, [y, [y, [y, [y, ...
```

where the sequence continues until terminated from by the user. We would need to introduce another output scheme to handle such cycles, but prefer to defer this issue at the moment.

Exercises

1. •• Describe how sharing can be useful for storing quad trees.

2. ••Describe how sharing can be useful for storing matrices.

3. ••• Describe some practical uses for deferred binding.

4. ••• Devise a notation for representing an information structure with sharing and possible cycles.

5. ••• For a computer operating system of your choice, determine whether and how sharing of files is represented in the directory structure. For example, in UNIX there are two kinds of *links*, regular and symbolic, which are available. Describe how such links work from the user's viewpoint.

2.12 Abstraction, Representation, and Presentation

We have used terms such as “abstraction” and “representation” quite freely in the foregoing sections. Having exposed some examples, it is now time to clarify the distinctions.

- By information *presentation*, we mean the way in which the information is presented to the user, or in which the user presents information to a computational system.

- By information *representation*, we mean the scheme or method used to record and manipulate the information, such as by a list, a matrix, function, etc., for example, inside a computer.

- By information *abstraction*, we mean the intangible set of “behaviors” that are determined by the information when accessed by certain functions and procedures.

A presentation can thus be viewed as a representation oriented toward a user's taste.

Number Representations

As a very simple example to clarify these distinctions, let us consider the domain of natural numbers, or “counting numbers”, which we normally denote 0, 1, 2, 3, ... When we write down things that we say represent numbers, we are dealing with the

presentation of the numbers. We do this in using *numerals*. For example, 5280 would typically be understood as a decimal numeral. The characters that make up a numeral are called *digits*. The actual *number* itself is determined by *abstract* properties. For example, the number of "ticks" we'd need to perform in counting starting from 0 to arrive at this number determines the number exactly and uniquely. There is only one number requiring exactly that many actions. Alternatively, the number of times we can remove a counter from a number before reaching no counters also determines the number.

Regarding representation of numbers in the computer, typically this would be in the form of a sequence of electrical voltages or magnetic flux values, which we abstract to *bits*: 0 and 1. This is the so-called *binary representation* (not to be confused with binary relations). Each bit in a binary representation represents a specific power of 2 and the number itself is derived by summing the various powers of two. For example, the number presented as decimal numeral 37 typically would be represented as the sequence of bits

1 0 0 1 0 1

meaning

$$\mathbf{1} * 2^5 + \mathbf{0} * 2^4 + \mathbf{0} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0$$

i.e.

32 + 0 + 0 + **4** + 0 + **1**

The appeal of the binary representation is that it is complete: every natural number can be represented in this way, and it is simple: a choice of one of two values for each power of two is all that is necessary.

An enumeration of the binary numerals for successive natural numbers reveals a pleasing pattern:

```
0          0
1          1
2         10
3         11
4        100
5        101
6        110
7        111
8       1000
    ...
```

The pattern is: if we look down the rightmost column, we see that 0's and 1's alternate on every line. If we look down the second rightmost column, they alternate every two lines, in the third every four lines, etc. In order to find the least-significant (rightmost) digit of the binary representation, we can divide the number by 2 (using whatever representation we want). The *remainder* of that division is the digit. The reader is invited, in the exercises, to extend this idea to a general method. In the programming languages we use, `N % M` denotes the remainder of dividing `N` by `M`.

Most real applications deal with information abstractions far more complex than just the natural numbers. It is important for the computer scientist to be familiar with typical abstractions and their attendant presentations and representations and to be able to spot instances of these abstractions as they arise in problems. This is one of the reasons that we have avoided focusing on particular common programming languages thus far; it is too easy to get lost in the representational issues of the language itself and loose sight of the information structural properties that are an essential component of our goal: to build well-engineered information processing systems.

Sparse Array and Matrix Representations

As another example of representation vs. abstraction, consider the abstract notion of arrays as a sequence indexed by a range of integers 0, 1, 2, ... N. A sparse array is one in which most of the elements have the same value, say 0. For example, the following array might be considered sparse:

```
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

since most of the elements are `0`. We could represent this array in a more compact, and in some ways more readable, as a list of only the non-zero elements and the values of those elements:

```
[ [1, 1], [3, 1], [7, 1], [9, 1], [13, 1], [27, 1] ]
```

Here the first element of each pair is the index of the element in the original array.

Likewise, we can consider a matrix to be an array indexed by a pair of values: row and column. The following sparse matrix (where blank entries imply to be some fixed value, such as 0)

0							
				5			
							3
					6		
		2					
						4	
	7						
			1				

could be represented as a list of non-zero elements of the form [row, col, value] (recalling that row and column indices start at 0):

```
[ [0, 0, 0], [6, 1, 7],
  [4, 2, 2], [7, 3, 1],
```

```
[1, 4, 5], [3, 5, 6],
[5, 6, 4], [2, 7, 3]]
```

This representation required 24 integers, rather than the 100 that would be required for the original matrix. While sparse array and matrix representations make efficient use of memory space, they have a drawback: linear addressing can no longer be used to access their elements. For example, when a matrix is to be accessed in a columnar fashion, we may have to scan the entire list to find the next element in a given column. This particular problem can be alleviated by treating the non-zero elements as nodes in a pair of directed graphs, one graph representing the columns and the other representing the rows, then using pointer or reference representations to get from one node to another.

Other Data Presentations

As far as different presentations, the reader is probably aware that there are many different presentations of numbers, for example different bases, Arabic vs. Roman numerals, etc. The same is true for lists. We have used one presentation of lists. But some languages use *S expressions*, ("S" originally stood for "symbolic"), which are similar to ours but with rounded parentheses rather than square brackets and omitting the commas. To differentiate, we could call ours *R expressions* (for rex expressions).

R expression: `[1, [2, 3], [ [ ], 4] ]`

S expression: `(1 (2 3) (() 4))`

S expressions often use entirely the same representation in the computer that we use: singly-linked lists. S expressions are used in languages Lisp and Scheme, while R expressions are used in the language Prolog. In Lisp and Scheme, programs, as well as data, are S expressions. This uniformity is convenient when programs need to be manipulated as data.

Regarding representations, we do not regard singly-linked lists as universally the most desirable representation. They are a simple representation and in many, but not all, cases can be a very efficient one. We will look at this and other alternatives in more detail in later chapters.

In our view, it is too early to become complacent about the abstractions supported by a given language. There are very popular languages on the scene that do a relatively poor job of supporting many important abstractions. We are not proposing rex as a solution, but only as a source of ideas, some of which some of our readers hopefully will design into the next generation of languages.

Exercises

1. •• Enumerate the numbers from 9 to 31 as binary numerals.

2. ••• Devise a method for converting a number into its binary representation.

3. ••• Devise a method for converting a binary numeral into decimal.

4. •• Devise a method for transposing a matrix stored in the sparse representation described in this section. Here *transpose* means to make the columns become rows and vice-versa.

5. •• A *formal polynomial* is an expression of the form

$$a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_{N-1}x^{N-1}$$

While polynomials are often used as representations for certain functions, the role played by a formal polynomial is just to represent the sequence of coefficients

$$[\ a_0, a_1, a_2, a_3, \ ...., a_{N-1}\ ]$$

The "variable" or "indeterminate" x is merely a symbol used to make the whole thing readable. This is particularly useful when the sequence is "sparse" in the sense that most of the coefficients are 0. For example, it would not be particularly readable to spell out all of the terms in the sequence represented by the formal polynomial

$$1 + x^{100}$$

Devise a way to represent formal polynomials using lists, without requiring the indeterminate symbol.

2.13 Abstract Information Structures

Previously in this chapter, we exposed a number of different information structures. A common characteristic of the structures presented was that we could *visualize* the structure, using presentations such as lists. But there is another common way to characterize structures and that is in terms of *behavior*. Information structures viewed this way are often called *abstract data types* (ADTs). This approach is the essence of an important set of ideas called *object-oriented programming*, which will be considered later from an implementation point of view. An object's class can be used to define an ADT, as the class serves to group together all of the operators for a particular data type.

As an example, consider the characterization of an array. The typical behavioral aspects of arrays are these:

- We can make an array of a specified length.
- We can set the value of the element at any specified index of the array.

- We can get the value of the element at any specified index.

These are the things that principally characterize an array, although we can always add others, such as ones that search an array, search from a given index, search in reverse, etc. All of these aspects can be understood carried out in terms of the three basic behaviors given, without actually "seeing" the array and how it is represented. We can postulate various abstract operators to represent the actions:

`makeArray(N)`	returns an array of size `N`
`set(A, i, V)`	sets the element of array `A` at index `i` to value `V`
`get(A, i)`	returns the element of array `A` at index `i`
`size(A)`	returns the size of array `A`

Note that aspects of *performance* (computation time) are not shown in these methods. While it would be efficient to use the linear addressing principle to implement the array, it is not required. We could use a sparse representation instead. What is implied, however, is that there are certain relationships among the arguments and return values of the methods. The essence is characterized by the following relationships, for any array `A`, with .... representing a series of zero or more method calls:

A sequence

```
A = makeArray(N); .... size(A)
```

returns `N`, the size of the array created.

A sequence

```
set(A, i, V); .... get(A, i)
```

returns `V`, provided that within .... there is no occurrence of `set(A, i, V)` and provided that `i > 0` and `i < size(A)`.

These summarize an *abstract array*, independent of any "concrete" array (*concrete* meaning a specific implementation, rather than an abstraction).

This kind of thinking should be undertaken whenever building an abstraction. That is, begin by asking the question

What are the properties that connect the operators?

The answers to such questions are what guide the choice or development of implementations, which will be the subject of much of this book.

Exercises

1. ••• Express the idea of a *list* such as we constructed using [.... | ....] from an abstract point of view, along the lines of how we described an abstract array.

2. •••• A stack is an abstraction with a behavior informally described as follows: A stack is a repository for data items. When the stack is first created, the stack contains no items. Item are inserted one at a time using the *push* method and removed one at a time using the *pop* method. The stack determines which element will be removed when pop is called: elements will be removed in order of most recently inserted first and earliest inserted last. (This is called a LIFO, or last-in-first-out ordering). The *empty* method tells whether or not the stack is empty. Specify the relationships among methods for a stack.

3. •• Many languages include a *struct* facility for structuring data. (An alternate term for struct is *record*.) A struct consists of a fixed number of components of possibly heterogenoeous type, called *fields*. Each component is referred to by a name. Show how structs can be represented by lists.

2.14 Conclusion

In this chapter, we have primarily exposed ways of thinking about information structures. We have not written much code yet, and have said little about the implementations of these structures. We mentioned in the previous section the idea of abstract data types and the related idea of “objects”. Objects and information structures such as lists should not be viewed as exclusive. Instead, there is much possible synergism. In some cases, we will want to use explicit information structures containing objects as their elements, and in others we will want to use information structures to implement the behavior defined by classes of objects. It would be unfortunate to box ourselves into a single programming language paradigm at this point.

2.15 Chapter Review

Define the following concepts or terms:

acyclic
ADT (abstract data type)
array
binary relation
binary representation of numbers
binary tree representation of list
binding
connection matrix
dag
deferred binding
directed graph
forest
functional programming
index
information sharing
labeled-tree interpretation of a list

leaf
leafcount
length (of a list)
linear addressing principle
list
list equality
list matching
numeral vs. number
quad tree
queue
R expression
reachable
reachability matrix
reference
root
S expression
sharing
sparse array
stack
target set
transitive closure
tree
trie
unlabeled-tree interpretation of a list

3. High-Level Functional Programming

3.1 Introduction

This chapter focuses on *functional programming*, a very basic, yet powerful, paradigm in which computation is represented solely by applying functions. It builds upon the information structures in the preceding chapter, in that those structures are representative of the kinds of structures used as data in functional programming. Some additional characteristics of the functional programming paradigm are:

- Variables represent values, rather than memory locations.
- In particular, there are no assignment statements; all bindings to variables are done in terms of *definitions* and function arguments.
- Data are never modified once created. Instead, data are created from existing data by functions.
- Each occurrence of a given functional expression in a given context always denotes a single value throughout its lifetime.

This last point is sometimes called “referential transparency” (although it might have been equally well called “referential opacity”) since one cannot see or discern any differences in the results of two different instances of the same expression.

A term often used for modifying the value of a variable is “side effect”. In short, in a functional language there is no way to represent, and thus cause, side effects.

Some advantages that accrue from the functional style of programming are:

- Debugging a program is simpler, because there is no dependence on sequencing among assignment statements. One can re-evaluate an expression many times (as in debugging interactively) without fear that one evaluation will have an effect on another.
- Sub-expressions of a program can be processed *in parallel* on several different processors, since the meaning of an expression is inherent in the expression and there is no dependence on expression sequencing. This can produce a net speed-up of the execution, up to a factor of the number of processors used.
- Storage is managed by the underlying system; there is no way to specify allocation or freeing of storage, and the attendant problems with such actions are not present.

One does not need a functional language to program in a functional style. Most of the principles put forth in this section can be applied to ordinary imperative languages, provided that the usage is disciplined. Please note that while functional-programming has much to recommend it, we are not advocating its exclusive use. It will later be combined with other programming paradigms, such as object-oriented programming. It is difficult to present functional programming in that context initially, because the key ideas tend to become obscured by object-oriented syntax. For this reason we use the language rex rather than other more common languages. Once the ideas are instilled, they can be applied in whatever language the reader happens to be working.

3.2 Nomenclature

Before starting, it is helpful to clarify what we mean by *function*. We try to use this term mostly in the mathematical sense, rather than giving the extended meaning as a synonym for *procedure* as is done in the parlance of some programming languages.

> A *function* on a set (called the *domain* of the function) is an entity that associates, with each member of the set, a single item.

The key word above is *single*, meaning *exactly one*. A function never associates two or more items with a given member of the domain, nor does it ever fail to associate an item with any member of the domain.

We say the function, given a domain value, *yields* or *maps to* the value associated with it. The syntax indicating the element associated with a domain element x, if f represents the function, is $f(x)$. However this is only one possible syntax of many; the key idea is the association provided.

Examples of functions are:

- The *add1* function: Associates with any number the number + 1. (The domain can be any set of numbers).
- The *multiply* function: Associates with any pair of numbers the product of the numbers in the pair. (The domain is a set of pairs of numbers.)
- The *reverse* function: Associates with any list of elements another list with elements in reverse order.
- The *length* function: Associates with any list a number giving the length of the list.
- The *father* function: Associates with any person the person's father.

- The *zero* function: Associates with any value in its domain the value 0.

Note that there is no requirement that two different elements of the domain can't be associated with a single value. That is, *f*(*x*) could be the same as *f*(*y*), even though *x* and *y* might be bound to different values. This occurs, for example, in the case of the multiply function: multiply(3, 4) gives the same value as multiply(4, 3). Functions that prohibit *f*(*x*) from being the same as *f*(*y*) when *x* and *y* are bound to different values are called *one-to-one* functions. Functions such as the *zero* function, which associates the same value with *all* elements of the domain are called *constant functions*.

A related definition, where we will tend to blur the distinction with function as defined, is that of partial function:

> A *partial function* on a set is an entity that associates, with each member of a set, *at most one* item.

Notice that here we have replaced "single" in the definition of "function" with "at most one". In the case of a partial function, we allow there to be *no* item associated with some members of the set. In this book, the same syntax is used for functions and partial functions. However, with a partial function *f*, it is possible to have no value *f*(*x*) for a given *x*. In this case, we say that *f*(*x*) *is undefined*.

An example of a partial function that is not a function is:

> The *divide* function: It associates with any pair of numbers the first number divided by the second, except for the case where the second is 0, in which case the value of the function is undefined.

An example of a partial function on the integers is a *list*:

> A list may be viewed as a partial function that returns an element of the list given its index (0, 1, 2, 3, ...). For any integer, there is at most one element at that index. There is no element if the index is negative or greater than N-1 where N is the length of the list. Finally, there must be no "holes", in the sense that the partial function is defined for all values between 0 and N-1.

We will use the common notation

> f: A $\rightarrow$ B

to designate that f is a partial function on set A, and that every value f(a) for a in A is in the set B.

Evidently, any partial function is a function over a sufficiently selective domain, namely the set of values for which the partial function is defined. Another way to remove the *partial* aspect is by defining a *special element* to indicate when the result would have otherwise been undefined. For example, in the case of divide by 0, we could use `Infinity` to indicate the result (rex does this). However, it is important to note than in computational systems, there are cases where this scheme for removing the *partial* nature of partial function can only be done for the sake of mathematical discussion; that is, we cannot, in some cases, *compute* the fact that the result will be undefined. The phenomenon to which we allude is *non-termination* of programs. While we often would like to think of a program as representing a function on the set of all possible inputs, for some inputs the program might not terminate. Moreover, it cannot always be detected when the program will not terminate. So in general, programs represent partial functions at best.

Two of the main ways to represent a partial function for computational purposes are: *by equation* and *by enumeration*. When we say "by equation", we mean that we give an equation that defines the function, in terms of constants and simpler functions. Examples in rex are:

```
f(x) = x*5;

g(x, y) = f(x) + y;

h(x) = g(f(x), 9);
```

Here `*`, `+`, `5`, and `9` are symbols that are "built-in" to rex and have their usual meaning (multiplication, addition, and two natural numbers). The semi-colon simply tells rex that the definition stops there (rather than, say, continuing on the next line).

When we say "by enumeration", we mean giving the set of pairs, the left elements of which are the domain elements and the right elements the corresponding values of the function. For example, we enumerate the *add1* function on the domain {0, 1, 2, 3, 4} by the list:

```
[ [4, 5], [3, 4], [2, 3], [1, 2], [0, 1] ]
```

Here we are treating a list as a set. Any reordering of the list would do as well.

While we can apply a function defined by equation by simply juxtaposing it with its arguments, e.g.

```
rex > f(1);
5

rex > g(2, 3);
13
```

we cannot do this in rex with a function enumerated by a list of pairs. We must instead pass the list to another function, such as one that we call `assoc` (short for *associate*), which will be described a bit later.

Likewise, we can go from a computed version of a function to an (internally) tabulated version using an idea known as *caching*. Caching allows rex to possibly bypass re-computing the function by first consulting the table. If the domain value is not present in the table, it will compute it, then put it in the table.

Pragmatically, to cause caching to occur in rex, we issue a one-time *directive*, such as:

```
rex > sys(on, cache(f#1));
```

The # sign is used to identify the number of arguments to this particular function (since different functions with different numbers of arguments can use the same function name, the selection being determined by the number of arguments). So here we are causing caching of the one-argument function with name `f`.

Another case of enumeration of a function occurs when the domain consists of consecutive natural numbers from 0 to some value. In this case, we can list the corresponding function values *in order* and apply the list. For example, in the preceding example of the domain-limited `add1` function, we could list the value (leaving the domain implicit) as:

```
[1, 2, 3, 4, 5]
```

This list can be applied by merely juxtaposing it with an argument:

```
rex > [1, 2, 3, 4, 5](3);
4
```

Exercises

1. ••• Specify, by equation, a function that gives the area of a triangle given the lengths of the sides as arguments. (Use Heron's formula, which may be found in various mathematics references).

2. •• Specify, by enumeration, a function that gives the number of days in each month in the year (assume 28 for February).

3.3 Using High-Level Functions

In the previous chapter, we used some of rex's built-in functions such as `length` and `type` to demonstrate properties of information structures. Later we will show how to construct some functions on our own. First, however, we want to get more practice in simply using functions. This will help in thinking at a relatively high level about information structures and functions that use them and create them.

Suppose we wish to create a list of numbers over a given range, where each number is 1 greater than its predecessor. To do this, we can use the function `range` that takes the lower and upper limit of the range as its arguments:

```
rex > range(1, 10);
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

rex > range(1.5, 9.5);
[1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5]
```

If we want the increment to be other than 1, we can use a three-argument version of `range` that specifies the increment:

```
rex > range(0, 20, 2);
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

rex > range(20, 0, -2);
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
```

As can be seen above, in rex the same function name can be used to designate different functions, based upon the number of arguments. The number of arguments is called the *arity* of the function (derived from using *n*-ary to mean *n* arguments). The use of one name for several different functions, the choice of which depends on the arity or the type of arguments, is called *overloading* the function's name.

Function `scale` multiplies the elements of an arbitrary list to produce a new list.

```
rex > scale(5, [1, 2, 20]);
[5, 10, 100]
```

As examples of functions on lists, we have `prefix`, which returns a specified-length prefix of a sequence:

```
rex > prefix(4, [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]);
[1, 2, 3, 5]
```

`antiprefix`, which returns the sequence with the prefix of that length taken off:

```
rex > antiprefix(4, [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]);
[7, 11, 13, 17, 19, 23]
```

and `suffix`, which returns the last so many elements of the list.

```
rex > suffix(4, [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]);
[13, 17, 19, 23]
```

Function `remove_duplicates` returns a list of the elements in the argument list, with subsequent duplicates removed:

```
rex > remove_duplicates([1, 2, 3, 2, 3, 4, 3, 4, 5]);
```

```
[1, 2, 3, 4, 5]
```

Function `zip` interlaces elements of two lists together, as if they were the two sides of a zipper:

```
rex > zip([1, 3, 5, 7], [2, 4, 6]);
[1, 2, 3, 4, 5, 6, 7]
```

Function `reverse` reverses elements of a list:

```
rex > reverse([1, 2, 3, 4, 5]);
[5, 4, 3, 2, 1]
```

Function `append` appends the elements of the second list to those of the first:

```
rex > append([1, 2, 3, 4], [5, 6]);
[1, 2, 3, 4, 5, 6]
```

Sorting Lists

Another high-level function is `sort`, which sorts its argument list into ascending order:

```
rex > sort(["peas", "beans", "oats", "barley"]);
[barley, beans, oats, peas]
```

When we do this with functional programming, we are *not modifying* the original list. Instead, we are creating a new list from the original. To verify this using rex:

```
rex > L = ["peas", "beans", "oats", "barley"];
1

rex > M = sort(L);
1

rex > M;
[barley, beans, oats, peas]

rex > L;
[peas, beans, oats, barley]
```

We can see that `L` has not changed from the original.

Function `sort` will also work when the list's *elements* are lists. It uses the idea of *lexicographic ordering* to compare any two elements. Lexicographic ordering is like dictionary ordering extended to any ordered set, such as the set of numbers. For example, the empty list `[ ]` is < than any other list. One non-empty list is < another provided that the first element of one is < the other or the first elements are equal and the rest of the first list is < the rest of the second. For example:

```
rex > [1,2,3] < [1,2,4];
```

```
1

rex > [1,2,3] < [1,4];
1

rex > [1,2] < [1,1,2];
0
```

Notice that it is this same principle that allows us to sort a list of words, as if the words were lists of characters.

In rex, numbers are < strings, and strings are < lists, but this is somewhat arbitrary. The purpose of it is for certain algorithms such as removing duplicates, where an ordering is helpful.

Finding Things in Lists

A couple of frequently-used functions that can be used to tell us things about the contents of lists are `member` and `assoc`. Function `member` is a predicate that tells whether a specific item occurs in a list: `member(E, L)` returns `1` if `E` occurs in `L` and returns `0` otherwise. For example,

```
rex > member(99, range(90, 100));
1

rex > member(99, range(90, 95));
0
```

Function `assoc` applies to a special kind of list called an *association list*, a list of lists, each with at least one element. Usually this will be a list of pairs representing either a dictionary or a binary relation. The purpose of `assoc` is to find an element in an association list having a specified first element. If the list is a dictionary, we can thus use `assoc` to find the meaning of a given word. As an example,

```
rex > assoc(5, [ [4, 16], [5, 25], [6, 36], [7, 49] ]);
[5, 25]
```

Note that by definition of an association list, if an element is found, the returned value will always be non-empty (because each list is supposed to have at least one element in it). Thus we can use the empty list as a returned value to indicate no element was found:

```
rex > assoc(3, [ [4, 16], [5, 25], [6, 36], [7, 49] ]);
[ ]
```

Typically the association list represents an enumerated *partial function*, which, as we stated earlier, is a binary relation such that no two different pairs in the relation have the same first element. For example,

```
[ [5, “apple”], [6, “banana”], [10, “grapefruit”] ]
```

is a partial function, whereas

```
[ [5, “apple”], [6, “banana”], [10, “grapefruit”], [6, “orange”]
]
```

is not, because in the latter there are two pairs with first component 6. However, even if the list were not a partial function, `assoc` treats it as if it were by only returning the *first* pair in the list that matches the argument and ignoring the rest of the pairs. If one wanted to verify that the pair was unique, one could use function `keep` to find out.

Implementing Ordered Dictionaries

An *ordered dictionary* associates with each item in a set (called the set of keys) another item, the “definition” of that item. An implementation of an ordered dictionary is a list of ordered pairs, where a pair is a list of two elements, possibly of different types. The first element of each pair is the thing being defined, while the second is its. For example, in the following ordered dictionary implementation, we have musical *solfege* symbols and their definitions. Each definition is a list of words:

```
[ ["do", ["a", "deer", "a", "female", "deer"]],
  ["re", ["a", "drop", "of", "golden", "sun"]],
  ["me", ["a", "name", "I", "call", "myself"]],
  ["fa", ["a", "long", "long", "way", "to", "run"]],
  ["so", ["a", "needle", "pulling", "thread"]],
  ["la", ["a", "note", "that", "follows", "sol"]],
  ["ti", ["a", "drink", "with", "jam", "and", "bread"]]
]
```

Exercises

1. •• Using the functions presented in this section, along with arithmetic, give a one-line rex definition of the following function:

   `infix(I, J, L)` yields elements `I` through `J` of list `L`, where the first element is counted as element 0. For example,

   ```
   rex > infix(1, 5, range(0, 10));
   [1, 2, 3, 4, 5]
   ```

2. •• An *identity* on two functional expressions indicates that the two sides of the expression are equal for all arguments on which one of the two sides is defined. Which of the following identities are correct?

```
append(L, append(M, N)) == append(append(L, M), N)

reverse(reverse(L)) == L

reverse(append(L, M)) == append(reverse(L), reverse(M))

sort(append(L, M)) == append(sort(L), sort(M))

reverse(sort(L)) == reverse(L)

sort(reverse(L)) == sort(L)

[A | append(L, M)] == append([A | L], M)

reverse([A | L]) == [A | reverse(L)]

reverse([A | L]) == [reverse(L) | A]

reverse([A | L]) == append(reverse(L), [A])
```

3. ••• Two functions of a single argument are said to *commute* provide that for every argument value `X`, we have `f(g(X)) == g(f(X))`. Which pairs of functions commute?

   `sort` and `remove_duplicates`

   `reverse` and `remove_duplicates`

   `sort` and `reverse`

3.4 Mapping, Functions as Arguments

An important concept for leveraging intellectual effort in software development is the ability to use functions as arguments. As an example of where this idea could be used, the process of creating a list by doing a common operation to each element of a given list is called *mapping* over the list. Mapping is an extremely important way to think about operations on data, since it captures many similar ideas in a single high-level thought. (The word mapping is also used as a noun, as a synonym for function or partial function; this is not the use for the present concept.) Later on we will see how to define our own mapping functions. One of the attractive uses of high-level functions is that we do not over-specify how the result is to be achieved. This leaves open many possibilities for the compiler to optimize the performance of the operation.

The function `map` provides a general capability for mapping over a single sequence. For example, suppose we wish to use mapping to create a list of squares of a list of numbers. The first argument to the function `map` is itself a *function*. Assume that `sq` is a function that squares its argument. Then the goal can be accomplished as follows:

```
rex > map(sq, [ 1, 7, 5, 9 ]);
[1, 49, 25, 81]
```

Likewise, one way to create a list of the cubes of some numbers would be to *define* a function `cube` (since there isn't a built in one) and supply that as an argument to `map`. In rex, a function can be defined by a single equation, as shown in the first line below. We then supply that user-defined function as an argument to map:

```
rex > cube(X) = X*X*X;
1

rex > map(cube, [1, 7, 5, 9]);
[1, 343, 125, 729]
```

There is also a version of `map` that takes three arguments, the first being a function and the latter two being lists. Suppose that we wish to create a list of pairs of components from two argument lists. Knowing that function `list` will create a list of its two given arguments, we can provide `list` as the first argument to `map`:

```
rex > map(list, [1, 2, 3], [4, 5, 6]);
[[1, 4], [2, 5], [3, 6]]
```

A function such as `map` that takes a function as an argument is called a *higher-order*function.

3.5 Anonymous Functions

A powerful concept that has a use in conjunction with `map` is that of *anonymous function*. This is a function that can be created by a user or programmer, but which does not have to be given a name.

As a simple example of an anonymous function, suppose we wish to create a list by adding 5 to each element of a given list. We could do this using `map` and an "add 5" function. The way to define an "add 5" function without giving it a name is as follows:

```
(X) => X + 5
```

This expression is read:

> *the function that, with argument* `X`*, returns the value of* `X + 5`.
>
> The "arrow" `=>` identifies this as a *functional expression*.

We apply the anonymous function by giving it an argument, just as with any other function. Here we use parentheses around the functional expression to avoid ambiguity:

```
((X) => X + 5)(6)
```
function *argument*

Here the function designated by `(X) => X + 5` is applied to the actual argument `6`. The process is that formal argument `X` gets bound to the actual argument `6`. The body, with this identification, effectively becomes `6 + 5`. The value of the original expression is therefore that of `6 + 5`, i.e. `11`.

Here is another example, an anonymous function with two arguments:

```
((X, Y) => Y - X)(5, 6)
```

The function designated by `(X, Y) => Y - X` is applied to the actual pair of arguments `(5, 6)`. Formal argument `X` is bound to `5`, and `Y` to `6`. The result is that of `6 - 5`, i.e. `1`.

A common use of such anonymous expressions is in conjunction with functions such as `map`. For example, in order to cube each element of a list `L` above, we provided a function `cube` as the first argument of the function `map`, as in `map(cube, L)`. In some cases, this might be inconvenient. For example, we'd have to disrupt our thought to think up the name "`cube`". We also "clutter our name-space" with yet another function name. The use of such an anonymous function within `map` then could be

```
rex > map( (X) => X*X*X, range(1, 5));
[1, 8, 27, 64, 125]
```

Anonymous functions can have the property that identifiers mentioned in their expressions can be given values apart from the parameters of the function. We call these values *imported* values. In the following example

```
((X) => X + Y)(6)
```

`Y` is not an argument. It is assumed, therefore, that `Y` has a value defined from its context. We call this a *free variable* as far as the functional expression is concerned. For example, `Y` might have been given a value earlier. The function designated by `(X) => X + Y` is applied to the argument `6`. The result of the application is the value of `6 + Y`. We need to know the value of `Y` to simplify it any further. If `Y` had the value `3`, the result would be `9`. If `Y` had been given no value, the result would be undefined (rex would complain about `Y` being an *unbound variable*).

Here is an application of the idea of imported values. Consider defining the function `scale` that multiplies each element of a list by a factor. Since this is a `map` -like concept, the hope is we could use `map` to do it. However, to do so calls for an anonymous function:

```
scale(Factor, List) = map( (X) => Factor*X,  List);
```

Here `X` represents a "typical" element of the list, which is bound to values in the list as map applies its function argument to each of those elements. The variable `Factor`, on the other hand, is not bound to values in the list. It gets its value as the first argument to function `scale` and that value is *imported* to the anonymous function argument of `map`.

As a still more complex anonymous function example, consider:

```
((F) => F(6))((X) => X * 3)
```

Here the argument `F` to the functional expression `(F) => F(6)` is itself a functional expression `(X) => X * 3`. We identify the formal argument `F` with the latter, so the body becomes `((X) => X * 3)(6)`. We can then simplify this expression by performing the application indicated, with `X` identified with `6`, to get `6*3`, which simplifies to `18`.

In computer science, another, less readable, notation is often used in place of the `=>` notation we use to define anonymous functions. This is called "lambda notation", "lambda abstraction", or Church's **lambda calculus**. Instead of the suggestive `(X) => Y*X - 1`, lambda notation prescribes `λX.(Y*X - 1)`. In other words, the prefix `λ` takes the place of the infix `=>`.

3.6 Functions as Results

An interesting aspect about anonymous functions is that they can be returned as *results*. Consider

```
(Y) => ( (X) => X + Y )
```

This is read

> "the function that, with argument `Y`, returns:
>
> > the function that, with argument `X`, returns the value of `X + Y`"

In other words, the first function mentioned returns a function as its value. The second outer parentheses can be omitted, as in

```
(Y) => (X) => X + Y
```

because grouping around `=>` is to the right. Obviously we are again applying the idea of an *imported* variable, since the value of `Y` is not an argument but rather is imported to the inner expression.

When we apply such a function to a number such as `9`, the result is a function, namely the function represented by

```
(X) => X + 9
```

What happens, for example, when we map a function-returning function over a list of numbers? The result is a *list of functions*:

```
rex > map( (Y) => (X) => X + Y, [5, 10, 15] );
[(X) => (X+5), (X) => (X+10), (X) => (X+15)]
```

While the idea of a list of functions might not be used that often, it is an interesting to exercise the concept of one function returning another. The point here is to get used to thinking of functions as whole entities in themselves.

Suppose we wanted to apply each of the functions in the list above to a single value, say `9`. We could use `map` to do that, by mapping a function with a *function* argument:

```
(F) => F(9)
```

is, of course, the function that with argument `F` returns the result of applying `F` to `9`. When we map this function over the previous result, here's what we get:

```
rex > L = map((Y) => (X) => X + Y, [5, 10, 15]);
1

rex > map( (F)=>F(9),  L);
[14, 19, 24]
```

Consider the problem of making a list of *all possible pairs* of two given lists. For example, if the lists were:

`[1, 2, 3, 4]` and `[96, 97, 98]`

then we want the result to be something like

```
[ [1, 96], [1, 97], [1, 98],  [2, 96], [2, 97], [2, 98],
  [3, 96], [3, 97], [3, 98],  [4, 96], [4, 97], [4, 98]]
```

Note that this is quite different from the result of

```
map(list, L)
```

discussed earlier. We solve this problem by first considering how to make all pairs of a *given* element, say `X`, with each element of the second list, say `M`. This can be accomplished by using map:

```
map((Y) => [X, Y], M)
```

Now we make a function that does this mapping, taking `X` as its argument:

```
(X) => map((Y) => [X, Y], M)
```

Now consider mapping this function over the first list `L`:

```
map((X) => map((Y) => [X, Y], M), L)
```

For the present example, this doesn't quite do what we want. The result is a list of lists of pairs rather than a list of pairs:

```
[ [ [1, 96], [1, 97], [1, 98] ],
  [ [2, 96], [2, 97], [2, 98] ],
  [ [3, 96], [3, 97], [3, 98] ],
  [ [4, 96], [4, 97], [4, 98] ] ]
```

Instead of the outer application of `map`, we need to use a related function `mappend` (`map`, then `append`) to produce the list of all the second level lists appended together:

```
mappend((X) => map((Y) => [X, Y], M), L)
```

Let's try it:

```
rex > L = [1, 2, 3, 4];
1

rex > M = [96, 97, 98];
1

ex > mappend((X) => map((Y) => [X, Y], M), L);

[ [1, 96], [1, 97], [1, 98], [2, 96], [2, 97], [2, 98],
  [3, 96], [3, 97], [3, 98], [4, 96], [4, 97], [4, 98] ]
```

We can package such useful capabilities as functions by using the expression in a function-defining equation.

```
pairs(L, M) = mappend((X) => map((Y) => [X, Y], M), L);
```

3.7 Composing Functions

Suppose we wish to construct a function that will take two other functions as arguments and return a function that is the *composition* of those functions. This can be accomplished using the following rule:

```
compose(F, G) = (X) => F(G(X));
```

We could alternatively express this as a rule using a *double layer of arguments*:

```
compose(F, G)(X) = F(G(X));
```

Let's try using compose in rex:

```
rex > compose(F, G) = (X) => F(G(X));
1

rex > square(X) = X*X;
1

rex > cube(X) = X*X*X;
1
```

```
rex > compose(square, cube)(2);
64

rex > compose(cube, square)(2);
64

rex > compose(reverse, sort)([3, 5, 1, 2, 6, 7]);
[7, 6, 5, 3, 2, 1]

rex > map(compose(square, cube), range(1, 10));
[1, 64, 729, 4096, 15625, 46656, 117649, 262144, 531441, 1000000]
```

In mathematical texts, the symbol o is often used as an infix operator to indicate composition:

compose(F, G) ≡ F o G

The following diagram suggests how *compose* works. It in effect splices the two argument functions together into a single function.

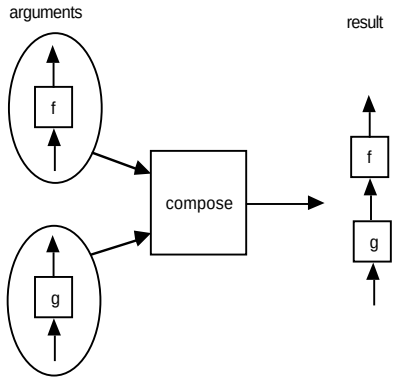


Figure 31: Diagram of the action of the function-composing function compose

3.8 The Pipelining Principle

A key design technique in computer science and engineering involves the *decomposition* of a specified function as a composition of simpler parts. For example, in the UNIX® operating system, single application programs can be structured as functions from a stream of input characters to a stream of output characters. The **composition** of two such functions entails using the output stream of one program as input to another. The result behaves as a program itself, and can be further composed in like fashion. One can regard this style of programming as building a **pipeline** connecting stages, each stage being an application program. Indeed, the operator for constructing such pipelines is known as "pipe" and shown as a vertical bar |. If P, Q, and R are programs, then

P | Q | R

defines the composition that connects the output of P to the input of Q and the output of Q to the input of R. The overall input then is the input to P and the output is the output of R. In terms of the function composition notation introduced earlier, we have R o (Q o P).

The pipelining principle is pervasive in computer science. Not only is it used in the construction of software; it is also extremely important at low levels of processor design, to enable parts of successive instructions to execute simultaneously. In later discussion, we will extend the idea of function composition to allow us to derive function composition networks of arbitrary structure.

3.9 Functions of Functions in Calculus (Advanced)

The failure to differentiate expressions from functions can be the source of confusion in areas such as differential and integral *calculus*, where it is easy to forget that *functions*, not numbers, are generally the main focus of discussion. For example, we are used to seeing equations such as

$$\frac{d}{dx} \; sin\ x = cos\ x$$

What is really meant here is that the result of operating on a function, *sin*, is another function, *cos*. The use of *x* is really irrelevant. It might have been less confusing to state

$$\text{derivative}(sin) = cos$$

and leave the dummy argument x out of the picture. A step in the right direction is to use the "prime" notation, wherein the derivative of a function *f* is shown as *f '*. But then we don't often see written equalities such as

$$sin' = cos$$

even though this, coupled with a proper understanding of functions as entities, would be less confusing than the first equation above.

As an example of the confusion, consider the **chain rule** in calculus, which can be correctly expressed as

$$(f \circ g)'(x) = f'(g(x)) * g'(x)$$

Using the *d/dx* notation, the chain rule cannot be expressed as nicely. If we are willing to *define* the product of two functions to be the function whose value for a given x is the product of the values of the individual functions, i.e.

$$(f * g)(x) = f(x) * g(x)$$

then the chain rule can be nicely expressed as:

$$(f \circ g)' = (f' \circ g) * g'$$

Translated to English, this statement says that the derivative of the composition of two functions is equal the product of the derivative of the second function and the composition of the derivative of the first function with the second function.

Exercises

1. • Create a function that cubes every element of a list, using `map` as your only named function.

2. •• Describe the functions represented by the following functional expressions:

```
a. (X) => X + 1

b. (Y) => Y + 1

c. (X) => 5

d. (F) => F(2)

e. (G, X) => G(X,X)

f. (X, Y) => Y(X)
```

3. •• Argue that `compose(F, compose(G, H))` == `compose(compose(F, G), H)`, i.e. that composition is associative. Written another way, `F o (G o H)` ≡ `(F o G) o H`. This being the case, we can eliminate parentheses and just write `F o G o H`.

4. ••• Express the calculus chain rule for the composition of three functions:

$$(F \circ G \circ H)' = ??$$

5. •• In some special cases, function composition is commutative, that is (F o G) ≡ (G o F). Give some examples of such cases. (Hint: Look at functions that raise their argument to a fixed power.)

6. •• Give an example that shows that function composition is not generally commutative.

7. •• Which functional identities are correct?

```
a. map(compose(F, G), L) ==  map(F, map(G, L))

b. map(F, reverse(L)) == reverse(map(F, L))

c. map(F, append(L, M)) == append(map(F, L), map(F, M))
```

3.10 Type Structure

A concern that occupies many computer scientists is that of the *types* of data and functions that operate on that data. The language we have been using so far, rex, is rather "loose" in its handling of types. This has its purpose: we don't wish to encumber the discussion with too many nuances at one time. Nonetheless, it is helpful to have a way to talk about expected types of data in functions; it helps us understand the specification of the function.

The basic types of most programming languages include:

integer numerals
characters or character strings
floating-point numerals

In order to provide a safe computational system, rex has to be able to discern the type of a datum dynamically: Although a rex variable is not annotated with any type, the basic operations in rex use the type of the data. For example, the + operator applies to integers or floating-point numerals, but not to character strings. Nothing prevents us from trying to use + on strings, but doing so will result in a run-time error that terminates the computation. Thus it is important that the programmer be aware of the type likely to be passed to a function. The rex language includes some built-in predicates for determining the type of data. For example, the predicate *is_number* establishes whether its argument is either an integer or floating point. The predicate *is_integer* establishes whether its argument is integer. The programmer can use these predicates to steer clear of run-time type errors.

It is common to treat data types as sets and to assert the type of functions using the customary domain-range notation on those sets. For example:

f: integer x integer → integer

asserts that function (or partial function) f takes two integer arguments and returns an integer.

In general, A x B, where A and B are sets, means the set of all pairs, the first element drawn from A and the second from B. This is called the *Cartesian product* of the sets. For example, the Cartesian product is computed by the function `pairs` worked out earlier.

In dealing with types, we use | to mean *union*, i.e. to describe elements that can be one of two different types. For example,

g: (integer | float) x integer → float

describes a function *g*, the first argument of which can be an integer or a float.

We often see type equations used to define intermediate classes. For example, we could define the type numeric to be the union of integer and float thus:

numeric = integer | float

We could also use equations to define types for functions:

binary_numeric_functions = (numeric x numeric) → numeric

treating the usual arrow notation as defining a set of functions.

Perhaps more important than the particular choice of basic types is the means of dealing with composite or aggregate types. The fundamental aggregation technique in rex is creating lists, so we could enlist the * notation to represent lists of arbitrary type things. For example,

integer*

could represent the type of lists of integers. Then

integer**

would represent the type of lists of lists of integers, etc. Since rex functions do not, in general, require their argument to be of any specific type, it is helpful to have a designation for the union of all types. This will be called *any*. For example, the type of the function length that counts the number of items in a list, is:

`length`: any* → integer

since this function pays absolutely no attention to the types of the individual elements in the argument list. On the other hand, some functions are best described using type variables. A good example is the function `map` that takes two arguments: a list of elements and a function. The domain of that function must be of the same type as the elements in the list. So we would describe `map` by

`map`: ((A → B) x A*) → B* where A and B are arbitrary types

A function such as `map` that operates on data of many different types is called *polymorphic*.

Function *compose* is a polymorphic function having following type:

((B → C) x (A → B)) → (A → C)

where A, B, and C are any three types.

Notice that although anonymous functions don't have names, we can still specify their *types* to help get a better understanding of them. For example, the type of `(X)=>X*X` is:

numeric → numeric

The set of lists permitting arbitrary nesting, which we have already equated to trees, deserves another type designator. If A is a type, then let's use $A^{\dagger}$ to designate the type that includes A, lists of A, lists of lists of A, and so on, ad infinitum. In a sense, $A^{\dagger}$ obeys the following type equation:

$$A^{\dagger} = A \mid (A^{\dagger})^{*}$$

That is to say $A^{\dagger}$ is the set that contains A and all lists of things of type $A^{\dagger}$. We shall encounter objects of this type again in later chapters when we deal with so called "S expressions".

We should cultivate the habit of checking that types *match* whenever a function is applied. For example, if

f: A → B

then, for any element a of type A, we know that f(a) is an element of type B.

Exercises

1. • Describe the types of the following functions:

   a. `(X) => X + 1`

   b. `(Y) => Y + 1`

   c. `(X) => 5`

   d. `(F) => F(2)`

   e. `(G, X) => G(X,X)`

   f. `(X, Y) => Y(X)`

2. •• Describe the type structure of the following functions that have been previously introduced:

   a. `range`

   b. `reverse`

   c. `append`

3.11Transposing Lists of Lists

In a previous section, we showed how to use `map` to pair up two lists element-wise:

```
rex > map(list, [1, 2, 3], [4, 5, 6]);
[[1, 4], [2, 5], [3, 6]]
```

Another way to get the effect of pairing elements would be to cast the problem as an instance of a more general function `transpose`. This function would understand the representation of matrices as lists of lists, as described in the previous chapter. By giving it a list of the two argument lists above, we get the same result, pairing of corresponding elements in the two lists. What we have done, in terms of the matrix view, is transposed the matrix, meaning exchanging the rows and columns in the two-dimensional presentation. For example, the transpose of the matrix

1	2	3
4	5	6

is

1	4
2	5
3	6

Using a list of lists, the transposition example would be shown as:

```
rex > transpose([ [1, 2, 3], [4, 5, 6] ]);
[[1, 4], [2, 5], [3, 6]]
```

However, `transpose` is more general than a single application of `map` in being able to deal with matrices with more than two rows, for example.

```
rex > transpose([ [1,   2,  3],
                  [4,   5,  6],
                  [7,   8,  9],
                  [10, 11, 12] ] );

[[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

The effect could be achieved with a double application of `map`, but to do so is a little tricky.

Exercises

1. • Give the type of function `transpose`.

2. •• Give the type of function `mappend` defined earlier.

3.12 Predicates

By a predicate, we mean a function to be used for discrimination, specifically in determining whether its arguments have a certain property or do not. Here we will use 1 (representing *true*) to indicate that an argument has the property, and 0 (also called *false*) otherwise, although other pairs of values are also possible, such as {yes, no}, {red, black}, etc. The *arity* of a predicate is the number of arguments it takes.

A simple example is the 2-ary predicate that we might call `less`, of two arguments, for example `less(3, 5) == 1`, `less(4, 2) == 0`, etc. Informally, `less(x, y)` is `1` whenever `x` is less than `y`, and `0` otherwise. We are used to seeing this predicate in infix form (with the symbol `<` between the arguments), i.e. `x < y` instead of `less(x, y)`. We could also use the symbol `<` instead of the name `less` in our discussion. Actually this is the name by which rex knows the predicate.

```
rex > map(<, [1, 2, 4, 8], [2, 3, 4, 5]);
[1, 1, 0, 0]
```

When an argument combination makes a predicate true, we say that the combination *satisfies* the predicate. This is for convenience in discourse.

Some functions built into rex expect predicates as arguments. An example is the function `some`: if `P` is a predicate and `L` is a list, then `some(P, L)`, returns 1 if some element of list `L` satisfies predicate `P`, and 0 otherwise. For example, `is_prime` is a predicate that gives the value 1 exactly if its argument is a prime number (a number that has no natural number divisors other than 1 and itself). We can ask whether any member of a list is prime using some in combination with `is_prime`. For example:

```
rex > some(is_prime, [4, 5, 6]);
1

rex > some(is_prime, [4, 6, 8, 10]);
0
```

Here 5 is the only prime. Note that `some` itself is a predicate. It would be called a *higher-order* predicate, because it takes a predicate as an argument. It could also be called a *quantifier*, since it is related to a concept in logic with that name. We shall discuss quantifiers further in a later chapter. A related quantifier is called `all`. The expression `all(P, L)` returns 1 iff all elements of `L` satisfy `P`. For example:

```
rex > all(is_prime, [2, 3, 5, 7]);
1
```

Often we want to know more than just whether some element of a list satisfies a predicate `P`; we want to know the identity of those elements. To accomplish this, we can use the predicate `keep`. The expression `keep(P, L)` returns the list of those elements of `L` that satisfy `P`, in the order in which they occur in `L`. For example:

```
rex > keep(is_prime, [2, 3, 4, 5, 6, 7, 8, 9]);
[2, 3, 5, 7]
```

Note that if sets are represented as lists, `keep` gives a facility like set selection in mathematics.

$$\{x \in S \mid P(x)\}$$

(read “the set of x in S such that P(X) is true”) is analogous to:

```
keep(P, S)
```

where we are representing the set S as a list. Function `keep` gives us a primitive database search facility: the list could be a list of lists representing records of some kind. Then `keep` can be used to select records with a specific property. For example, suppose our database records have the form

[*Employee*, *Manager*, *Salary*, *Department*]

with an example database being:

```
DB = [ ["Jones",   "Smith", 25000, "Development"],
       ["Thomas",  "Smith", 30000, "Development"],
       ["Simpson", "Smith", 29000, "Development"],
       ["Smith",   "Evans", 45000, "Development"]];
```

Then to pose the query “What records correspond to employees managed by Smith with a salary more than 25000, we could ask rex:

```
rex > keep((Record) => second(Record) == "Smith"
                     && third(Record) > 25000, DB);

[[Thomas, Smith, 30000, Development],
 [Simpson, Smith, 29000, Development]]
```

This could be made prettier by using pattern matching, however we have not yet introduced pattern matching in the context of anonymous functions and don’t wish to digress to do so at this point.

The complementary predicate to `keep` is called `drop`. The expression `drop(P, L)` returns the list of elements in `L` that do *not* satisfy `P`:

```
rex > drop(is_prime, [2, 3, 4, 5, 6, 7, 8, 9]);
```

```
[4, 6, 8, 9]
```

Sometimes we are interested in the *first* occurrence of an element satisfying a particular predicate, and might make use of the other occurrences subsequently. The predicate `find` gives us the suffix of list `L` beginning with the first occurrence of an element that satisfies `P`. If there are no such elements, then it will give us the empty list:

```
rex > find(is_prime, [4, 6, 8, 11, 12]);
[11, 12]

rex > find(is_prime, [12, 14]);
[ ]
```

The predicate `find_indices` gives us a list of the *indices* of all elements in a list which satisfy a given predicate:

```
rex > find_indices(is_prime, range(1, 20));
[0, 1, 2, 4, 6, 10, 12, 16, 18]

rex > find_indices(is_prime, range(24, 28));
[ ]
```

Exercises

1. • Suppose `L` is a list of lists. Present an expression that will return the lists of elements in `L` having length greater than 5.

2. ••• Present as many functional identities that you can among the functions `keep`, `find_indices`, `map`, `append`, and `reverse`, excluding those presented in earlier exercises.

3. ••• Show how to use `keep` and `map` in combination to define a function `gather` that creates a list of second elements corresponding to a given first element in an association list. For example,

```
rex > gather(3, [[1, "a"], [2, "b"], [3, "c"], [1, "d"],
                 [3, "e"], [3, "f"], [2, "g"], [1, "h"]]);
[c, e, f]
```

4. ••• Then define a second version of gather that gathers the second components of *all* elements of an association list together:

```
rex > gather ([[1, "a"], [2, "b"], [3, "c"], [1, "d"],
               [3, "e"], [3, "f"], [2, "g"], [1, "h"]]);
[[1, a, d, h], [2, b, g], [3, c, e, f]]
```

3.13 Generalized Sorting

A variation on the function `sort` has an additional argument, which is expected to be a binary predicate. This predicate specifies the comparison between two elements to be used in sorting. For example, to sort a list in reverse order:

```
rex > sort(>, [6, 1, 3, 2, 7, 4, 5]);
[7, 6, 5, 4, 3, 2, 1]
```

The ability to specify the comparison predicate is useful for specialized sorting. For example, if we have a list of lists of the form

[*Person*, *Age*]

and wish to sort this list by age, rather than lexicographically, we could supply a predicate that compares second elements of lists only:

```
(L, M) => second(L) < second(M).
```

Let's try this:

```
rex > Data =
          [["John", 25], ["Mary", 24], ["Tim", 21], ["Susan", 18]];
1

rex > sort((L, M) => second(L) < second(M), Data);
[[Susan, 18], [Tim, 21], [Mary, 24], [John, 25]]
```

In the next chapter, we will show some details for how lists can be sorted.

3.14 Reducing Lists

By *reducing* a list, we have in mind a higher-order function in a spirit similar to `map` introduced earlier. As with `map`, there are many occasions where we need to produce a single value that results from applying a *binary* function (i.e. 2-argument function, not be confused with binary relation or binary number representation introduced earlier) to elements of a list. Examples of reducing include adding up the elements in a list, multiplying the elements of a list, etc. In abstract terms, each of these would be considered reducing the list by a different binary operation.

For completeness, we need to say what it means to reduce the empty list. Typically reducing the empty list will depend on the operator being used. It is common to choose the mathematical *unit* of the operation, if it exists, for the value of reducing the empty list. For example, the sum of the empty list is 0 while the product is 1. The defining characteristic of the unit is that when another value is combined with the unit using the binary operator, the result is that other value:

For any number `x`:

```
0 + x == x

1 * x == x
```

The function `reduce` performs reductions based on the binary operator, the unit or other base value, and the list to be reduced:

```
rex > r = range(1, 10);
1

rex > r;
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

rex > reduce(+, 0, r);
55
```

That is, `55` is the sum of the natural numbers from 1 to 10.

```
rex > reduce(*, 1, r);
3628800
```

That is, `3628800` is the product of the natural numbers from 1 to 10.

Suppose we wished to create a single list out of the elements present in a list of lists. For two lists, the function that does this is called `append`:

```
rex > append([1, 2, 3], [4, 5]);
[1, 2, 3, 4, 5]
```

For a list of lists, we can use the higher-order function `reduce` with `append` as its argument. However, we have to decide what the unit for append is. We are looking for a value U such that for any list L

```
append(U, L) == L
```

That value is none other than the null list `[ ]`. So to append together an arbitrary list of lists L, we can use

```
reduce(append, [ ], L)
```

For example,

```
rex > reduce(append, [ ], [ [1, 2], [3, 4, 5], [ ], [6]] );
[1, 2, 3, 4, 5, 6]
```

Actually there are at least two different ways to reduce a list: since the operator operates on only two things at a time, we can group pairs starting from the right or from the left. Some languages make this distinction by providing two functions, `foldl` and `foldr`, with the same argument types as `reduce`. For example, if the operator is +, then

```
foldl(+, 0, [x0, x1, x2, x3, x4])
```

evaluates in the form

```
((((0 + x0)  + x1) + x2) + x3) + x4
```

whereas

```
foldr(+, 0, [x0, x1, x2, x3, x4])
```

evaluates in the form

```
x0 + (x1 + (x2 + (x3 + (x4 + 0))))
```

We can show this idea with rex by inventing a "symbolic" operator `op` that displays its arguments:

```
rex > op(X, Y) = concat("op(", X, ",", Y, ")");
1

rex > foldl(op, "unit", ["x0", "x1", "x2", "x3", "x4"]);
op(op(op(op(op(unit,x0),x1),x2),x3),x4)

rex > foldr(op, "unit", ["x0", "x1", "x2", "x3", "x4"]);
op(x0,op(x1,op(x2,op(x3,op(x4,unit)))))
```

Note that the base value in this example is not the unit for the operator.

Currently, rex uses the `foldl` version for `reduce`, but this is implementation-defined. For many typical uses, the operator is associative, in which case it does not matter which version is used. If it does matter, then one should use the more specific functions.

Exercises

1. •• Suppose we wish to regard lists of numbers as vectors. The inner product of two vectors is the sum of the products of the elements taken element-wise. For example, the "inner product" of `[2, 5, 7]`, `[3, 2, 8]` is `2*3 + 5*2 + 7*8 ==> 72`. Express an equation for the function `inner_product` in terms of `map` and `reduce`.

2. • Show that the rex predicate `some` can be derived from `keep`.

3. •• Suppose that `P` is a 1-ary predicate and `L` is a list. Argue that

   ```
   some(P, L) == !all((X) => !P(X), L)
   ```

   In other words, `P` is satisfied by some element of `L` if, and only if, the negation of `P` is not satisfied by all elements of `L`.

4. •• Show that the rex predicate `drop` can be derived from `keep`.

5. •• Using `reduce`, construct a version of `compose_list` that composes an arbitrary list of functions. An example of `compose_list` is:

```
compose_list([(A)=>A*A, (A)=>A+1, (A)=>A-5])(10) ==> 36
```

   Hint: What is the appropriate *unit* for function composition?

6. •• Which of the following expressions is of the proper type to reproduce its third argument list `L`?

```
a. foldl(cons, [ ], L)
b. foldr(cons, [ ], L)
```

3.15 Sequences as Functions

In computer science, it is important to be aware of the following fact:

> Every list can be viewed as a partial function on the domain of natural numbers (0, 1, 2, 3, ...).
>
> When the list is infinite, this partial function is a function.

That is, when we deal with a list $[x_0, x_1, x_2, \ldots]$ we can think of this as the following function represented as a list of pairs:

$$[[0, x_0], [1, x_1], [2, x_2], \ldots]$$

In the case that the list is finite, of length N, this becomes a partial function on the natural numbers, but a total function on the domain {0, 1, ..., N-1}.

This connection will become more important in subsequent chapters when we consider arrays, a particular sequence representation that can also be modeled as a function. The thing that it is important to keep in mind is that if we need to deal with functions on the natural numbers, we can equivalently deal with sequences (lists, arrays, etc.).

In rex, special accommodation is made for this idea, namely *a sequence can be applied as if it were a function*. For example, if *x* denotes the sequence [0, 1, 4, 9, 16, 25, ...] then x(2) (*x* applied to 2) is 4, x(3) is 9, etc. Moreover, rex sequences need not be only lists; they can also be arrays. An array applied to an argument gives the effect of array indexing.

One way to build an array in rex is to just give the elements of the array to the function `array`:

`array(`a_0`, `a_1`, ...., `a_{n-1}`)`

(The function array has an arbitrary number of arguments.) Another way is to use the function `make_array`. The latter takes two arguments, a function, say `f`, and a natural number, say `n`, and gives the effect of

```
array(f(0), f(1), ...., f(n-1))
```

One reason we might prefer such an array to a function itself is to avoid re-evaluating the function at the same argument multiple times. Once the function values are "cached" in the array, we can access them arbitrarily many times without recomputing.

Array access is preferred over list access for reasons of efficiency. For arrays, we can get to any element in constant time. For lists, the computer has to "count up" to the appropriate element. This takes time proportional to the index argument value. For this reason, we emphasize the following:

Sequencing through a list L by repeated indexing L(i), L(i+1), L(i+2), ... is to be avoided, for reasons of efficiency.

We already know better ways to do this (using the list decomposition operators).

Exercises

1. ••• Construct a function that composes two functions represented as association lists. For example, the following shows the association list form of a composition:

```
    [ [0, 0], [1, 3], [2, 2], [3, 3] ]
o   [ [0, 3], [1, 2], [2, 1], [3, 0] ]
==> [ [0, 3], [1, 2], [2, 3], [3, 0] ]
```

   (Hint: Use `map`.)

2. ••• Construct a function that composes two functions represented as lists-as-functions, for example:

```
[0, 3, 2, 3] o [3, 2, 1, 0] ==> [3, 2, 3, 0]
```

   (Hint: Use `map`.)

3.16 Solving Complex Problems using Functions

Most computational problems can be expressed in the form of implementing some kind of function, whether or not functional programming is used as the implementation method. In this section, we indicate how functions can provide a way of thinking about decomposing a problem, to arrive at an eventual solution.

As we already observed, functions have the attractive property closure under composition: composing two functions gives a function. Inductively, composing any number of functions will give a function. In solving problems, we want to reverse the composition process:

> Given a specification of a function to be implemented, find simpler functions that can be composed to equal the goal function.

By continuing this process of decomposing functions into compositions of simpler ones, we may arrive at some functions that we can use that are already implemented. There may be some we still have to implement, either by further decomposition or by low-level methods, as described in the next chapter.

A Very Simple Example

Implement a function that, with a list of words as input, produces a list that is sorted in alphabetical order and from which all duplicates have been removed. Our goal function can be decomposed into uses of two functions: `sort` and `remove_duplicates`:

```
goal(L) = remove_duplicates(sort(L));
```

If we were working in rex, then those two functions are built-in, and we'd be done. Alternatively, we could set out to implement those functions using low-level methods.

An Example Using Directed Graphs

Consider the problem of determining whether a graph is acyclic (has no cycles). Assume that the graph is given as a list of pairs of nodes.

Examples of an acyclic vs. a cyclic graph is shown below:

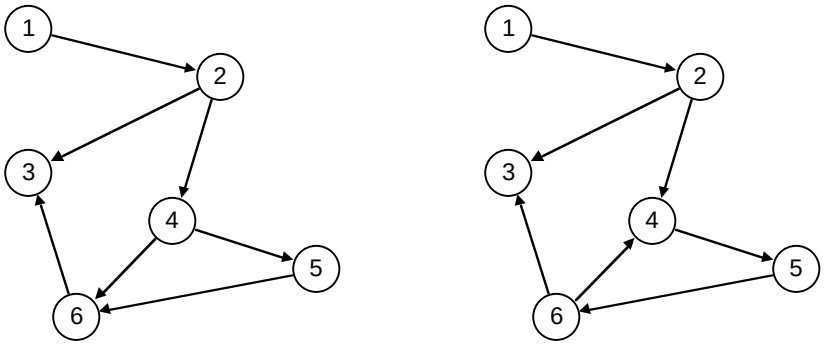


Figure 32: Acyclic vs. cyclic directed graphs

We'd like to devise a function by composing functions that have been discussed thus far. This function, call it `is_acyclic`, will take a list of pairs of nodes as an argument and return a 1 if the graph is acyclic, or a 0 otherwise.

Here's the idea we'll use in devising the function:

> If a graph is acyclic, then it must have at least one leaf.

A *leaf* is defined to be a node with no targets (also sometimes called a “sink”). So if the graph has no leaf, we immediately know it is not acyclic. However, a graph can have a leaf and still be cyclic. For example, in the rightmost (cyclic) graph above, node 3 is a leaf. The second idea we'll use is:

> Any leaf and attached arcs can be removed without affecting whether the graph is acyclic or not.

Removing a leaf may produce new leaves in the resulting graph. For example, in the leftmost (acyclic) graph above, node 3 is a leaf. When it is removed, node 6 becomes a leaf.

The overall idea is this:

> Starting with the given graph, repeat the following process as much as possible:
>
> > Remove any leaf and its connected arcs.

There are two ways in which this process can terminate:

1. All nodes have been eliminated, or

2. There are still nodes, but no more leaves.

In case 1, our conclusion is that the original graph was acyclic. In case 2, it was not. In fact, in case 2 we know that a cycle in the remaining graph exists and it is also a cycle of the original graph.

We now concentrate on presenting these ideas using functions. As a running example, we'll use the graph below, the representation of which is the following list:

```
[ [1, 2], [2, 3], [2, 4], [4, 5], [6, 3], [4, 6], [5, 6] ]
```

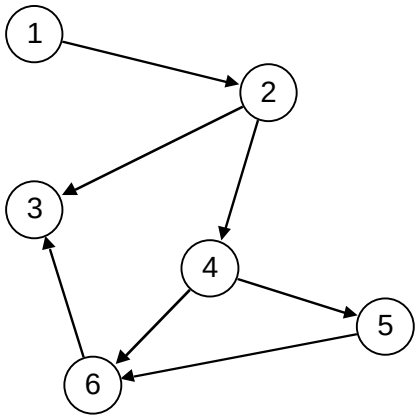


Figure 33: An acyclic graph for discussion

First we need a function that can determine whether there is a leaf. By definition, a leaf is a node with no arcs leaving it. A good place to start would seem to be devising a function that can determine whether a given node of a graph is a leaf, then iterate that function over the entire set of nodes. The following function is proposed:

```
is_leaf(Node, Graph) =

    no( (Pair) => first(Pair) == Node, Graph );
```

The function `no` applies its first argument, a predicate, to a list. If there is an element in the list satisfying the predicate, then there is a leaf. In this case, the predicate is given by the anonymous function

```
(Pair) => first(Pair) == Node
```

that asks the question: *is* `Node` *the first element of* `Pair`? The function `no` asks this question for each element of the list, stopping with 1 when a leaf is found, or returning 0 if no leaf is found.

On our example graph, suppose we try this function with arguments 3 and 4 in turn:

```
rex > graph = [ [1, 2], [2, 3], [2, 4], [4, 5],
                [6, 3], [4, 6], [5, 6] ];
1

rex > is_leaf(3, graph);
1

rex > is_leaf(4, graph);
0
```

Now let's use the `is_leaf` function to return a leaf in the graph, if there is one. Define `find_leaf` as follows:

```
find_leaf(Graph) =

    find((Node) => is_leaf(Node, Graph), nodes(Graph));
```

Here we are assuming that `nodes(Graph)` gives a list of all nodes in the graph. We're going to leave the implementation of this function as an exercise. The result of `find_leaf` will be a list beginning with the first leaf found. Only the first element of this list is really wanted, so we will use `first` to get that element.

Let's try `find_leaf` on the example graph:

```
rex > find_leaf(graph);
[3, 4, 5, 6]
```

indicating that 3 is a leaf, since it is the first element of a non-empty list. We can thus incorporate function `find_leaf` into one that tests whether there is a leaf:

```
no_leaf(Graph) = find_leaf(Graph) == [];
```

To remove a *known* leaf from a graph represented as a list of pairs, we must drop all pairs with the leaf as second element (there are no pairs with the leaf as first element, by definition of "leaf"). Here we use the function `drop` to do the work:

```
remove_leaf(Leaf, Graph) =

    drop((Pair) => second(Pair) == Leaf, Graph);
```

Similar to uses of `no` and `find`, the first argument of `drop` is a predicate. The resulting list is like the original list `Graph`, but with all pairs satisfying the predicate removed.

To test `remove_leaf` in action:

```
rex > remove_leaf(3, graph);
[[1, 2], [2, 4], [4, 5], [4, 6], [5, 6]]
```

Now we work this into a function that finds a leaf and removes it. We'll use the same name, but give the new function just one argument. By the way, here's where we apply `first` to the result of `find`:

```
remove_leaf(Graph) =

    remove_leaf(first(find_leaf(Graph)), Graph);
```

This function in action is exemplified by:

```
rex > remove_leaf(graph);
[[1, 2], [2, 4], [4, 5], [4, 6], [5, 6]]
```

Now we have one issue remaining: the iteration of leaf removal until we get to a stage in which either the graph is empty or no leaf exists. The following scenario indicates what

we'd like to have happen: We create new graphs as long as `no_leaf` is false, each time applying `remove_leaf` to get the next graph in the sequence. The reader is encouraged to trace the steps on a directed graph diagram.

```
rex > graph1;
[[1, 2], [2, 3], [2, 4], [4, 5], [6, 3], [4, 6], [5, 6]]

rex > no_leaf(graph1);
0

rex > graph2 = remove_leaf(graph1);
1

rex > graph2;
[[1, 2], [2, 4], [4, 5], [4, 6], [5, 6]]

rex > no_leaf(graph2);
0

rex > graph3 = remove_leaf(graph2);
1

rex > graph3;
[[1, 2], [2, 4], [4, 5]]

rex > no_leaf(graph3);
0

rex > graph4 = remove_leaf(graph3);
1

rex > graph4;
[[1, 2], [2, 4]]

rex > no_leaf(graph4);
0

rex > graph5 = remove_leaf(graph4);
1

rex > graph5;
[[1, 2]]

rex > no_leaf(graph5);
0

rex > graph6 = remove_leaf(graph5);
1

rex > no_leaf(graph6);
1

rex > graph6;
[]
```

The fact that the final graph is `[ ]` indicates that the original graph was acyclic.

Of course it is not sufficient to apply the transformations manually as we have done; we need to automate this iterative process using a function. Let's postulate a function to do the iteration, since we've not introduced one up until now:

```
iterate(Item, Action, Test)
```

will behave as follows. If `Test(Item)` is `1`, then iteration stops and `Item` is returned. Otherwise, iteration continues with the result being

```
iterate(Action(Item), Action, Test).
```

In other words, `Action` is applied to `Item`, and iteration continues. To accomplish our overall acyclic test then, we would use:

```
is_acyclic(Graph) =

    iterate(Graph, remove_leaf, no_leaf) == [];
```

which reads: "iterate the function `remove_leaf`, starting with `Graph`, until `Graph` is either empty or has no leaf, applying `Action` at each step to get a new `Graph`;. If the result is empty, then `Graph` is acyclic, otherwise it is not." In other words, the functional description succinctly captures our algorithm.

To demonstrate it on two similar test cases:

```
rex > is_acyclic([ [1, 2], [2, 3], [2, 4], [4, 5],
                   [6, 3], [4, 6], [5, 6] ]);
1

rex > is_acyclic([ [1, 2], [2, 3], [2, 4], [4, 5],
                   [6, 3], [6, 4], [5, 6] ]);
0
```

A Game-Playing Example

In this example, we devise a function that plays a version of the game of *nim*: There are two players and a list of positive integers. Each integer can be thought of as representing a pile of tokens. A player’s turn consists of selecting one of the piles and removing some of the tokens from the pile. This results in a new list. If the player removes all of the tokens, then that pile is no longer in the list. On each turn, only one pile changes or is removed. Two players alternate turns and the one who takes the last pile wins.

Example: The current set of piles is [2, 3, 4, 1, 5]. The first player removes 1 from the pile of 5, leaving [2, 3, 4, 1, 4]. The second player removes all of the pile of 4, leaving [2, 3, 1, 4]. The first player does the same, leaving [2, 3, 1]. The second player takes 1 from the pile of 3, leaving [2, 2, 1]. The first player takes the pile of 1, leaving [2, 2]. The second player takes one from the first pile, leaving [1, 2]. The first player takes one from

the second pile, leaving [1, 1]. The second player takes one of the piles, leaving [1]. The first player takes the remaining pile and wins.

There is a strategy for playing this form of nim. It is expressed using the concept of *nim sum*, which we will represent by ⊕. The nim sum of two numbers is formed by adding their binary representations column-wise *without carrying*. For example, 3 ⊕ 5 = 6, since

```
3      = 0 1 1
5      = 1 0 1
-------------
6      = 1 1 0
```

The nim sum of more than two numbers is just the nim sum of the numbers taken pairwise.

The strategy to win this form of nim is: *Give your opponent a list with a nim-sum of 0.* This is possible when you are given a list with a non-zero nim sum, and only then. Since a random list is more likely to have a non-zero nim sum than a zero one, you have a good chance of winning, especially if you start first and know this strategy.

To see why it is possible to convert a *non-zero* nim sum list to a zero nim sum list by removing tokens from one pile only, consider how that sum got the way it is. It has a high-order 1 in its binary representation. The only way it could get this 1 is that there is a number in the list with a 1 in that position. (There could be multiple such numbers, but we know there will be an odd number of them.) Given `s` as the nim sum of the list, we can find a number `n` with a 1 in that high-order position. If we consider `n ⊕ s`, that high-order 1 is added to the 1 `in s`, to give a number with a 0 where there was a 1. Thus this number is less than `n` . Therefore, we can take away a number of tokens, which reduces `n` to `n ⊕ s`. For example, if `n` were 5 and `s` were 6, then we want to leave 5 ⊕ 6 = 3. So what really counts is what we leave, and what we take is determined by that.

To see why the strategy works, note that if a player is given a non-empty list with a nim sum of 0, the player cannot both remove some tokens and leave the sum at 0. To see this, suppose that the player changes a pile with `n` tokens to one with `m` tokens, `m < n`. The nim sum of the original list is `n ⊕ r`, where `r` stands for the nim sum of the remainder of the piles. The new nim sum is `m ⊕ r`. Consider the sum `n ⊕ m ⊕ r`. This sum is equal to `m ⊕ n ⊕ r`, which is equal to `m`, since `n ⊕ r` is 0. So the new nim sum can only be 0 if he leaves 0 in some pile, i.e. he takes the entire pile. But taking a whole pile of size `n` is equivalent to nim-adding `n` to the nim sum, since `r` is the new sum and using the associative property for ⊕:

```
n ⊕ (n ⊕ r) == (n ⊕ n) ⊕ r
            == r
```

since

`(n ⊕ r)` is the original nim sum, assumed to be 0.
`(n ⊕ n) == 0` is a property of ⊕
`(0 ⊕ r) == r` is a property of ⊕

The only way nim-adding `n` to 0 can produce 0 is if `n` itself is 0, which is contradictory.

By always handing over a list with a sum of zero, a player is guaranteed to get back a list with a non-zero sum, up until the point where there is one pile left, which he takes and wins the game.

Now let's see how to make a good nim player as a function `play_nim`. Assume that the argument to our function is a non-empty list with a nim sum of non-zero. Then the player would decompose the problem into:

> Find the nim sum of the list, call it `s`.
>
> Find a pile to which `s` can be nim-added to produce a number less than `s`. Make a new list reflecting that change.

We can show this as:

```
play_nim(L) = make_new_list(nim_sum(L), L);
```

We can construct `nim_sum` by using `reduce` and a function that makes the nim sum of two numbers. Let's call this function `xor`. Then we have

```
nim_sum(L) = reduce(xor, 0, L);
```

The value of `make_new_list(s, L)` must make a list by replacing the *first* element `n` of `L` such that `s` ⊕ `n` < `n` with the value `s` ⊕ `n`. We don't have a function like this in our repertoire just yet, so let's postulate one:

```
change_first(P, F, L)
```

creates a new list from `L` by finding the first element satisfying predicate `P` and applying the function `F` to it, leaving other elements unchanged. We will have to appeal to the next chapter to see how to go further with `change_first`. We also need to drop the element in the case that it is 0. This can be done by our function `drop`:

```
make_new_list(s, L) =
  drop((n) => n == 0,
       change_first((n) => (xor(s, n) < n), (n) => xor(s, n), L));
```

So we have reduced the original problem to that of providing `xor` and `change_first`.

To complete our nim player, we have to provide an action for the case it is given a list with a non-zero sum. How to provide such alternatives will be discussed in the next chapter.

Exercises

1. •• Develop an implementation of the function `nodes` that returns the list of nodes of a graph represented as a list of pairs.

2. ••• Develop an implementation of the function `xor` used in the nim example.

3.17 Conclusion

This chapter has shown how information structures can be transformed using functions. Such techniques provide a powerful collection of viewpoints, even if we do not use the tools exclusively. The language rex was used to illustrate the concepts, however the ideas carry forward into many varieties of language, functional and otherwise. Later on, for example, we show how to express them in Java.

3.18 Chapter Review

Define the following concepts or terms:

acyclic graph test
anonymous function
`assoc` function
association list
composition of functions
definition by enumeration
definition by equation
higher-order function
lists as functions
leaf of a graph
`map` function
`mappend` function
nim sum
pipeline principle
predicate
`reduce` function
satisfy

4. Low-Level Functional Programming

4.1 Introduction

In the previous chapter, we saw how to use various functions over information structures defined in chapter 2. The emphasis in the previous chapter was to develop high-level thinking about working with such structures. The functions with which we worked were assumed to be built into rex. However large a repertoire of functions is provided in a functional language, there will usually be some things that we'd like to do that can't be captured in a manner that is as readable or as efficient as we might like. In the current chapter, we show how to write custom definitions of functions.

4.2 List-matching

Now we illustrate list decomposition using matching within a definition. Consider the form

```
[F | R]
```

(read `F` "followed by" `R`). This form represents a *pattern* or *template* that matches all, and only, non-empty lists. The idea is that identifier `F` matches the first of the list and identifier `R` matches the rest of the list. We can test this by trying a definition:

```
rex > [F | R] = [1, 2, 3];
1
```

The `1` (for *true*) indicates that the definition was successful. We can check that identifiers `F` and `R` are now bound appropriately, `F` to the first element of the list and `R` to the rest of the list:

```
rex > F;
1

rex > R;
[2, 3]
```

A definition of the form

identifier = *expression*;

will always succeed, but it is possible for a definition involving list matching to *fail*. For example,

```
rex > [F | R] = [ ];
0
```

Here the definition fails because we attempt to match `[F | R]` against the empty list. Such a match is impossible, because the empty list has no elements, and therefore no first element. In a similar way, an attempt to apply functions `first` or `rest` to the empty list results in an error:

```
rex > first([ ]);
*** warning: can't select from null list [ ]

rex > rest([ ]);
*** warning: can't take rest of null list [ ]
```

Extended Matching

The list-matching notation may be extended to extract an arbitrary number of initial elements of a list. For example, to extract the first, second, and third elements:

```
rex > [F, S, T | R] = [1, 4, 9, 16, 25, 36];
1
```

As before, the 1 indicates the definition succeeded. We can check that the correct identifications were made:

```
rex > F;
1

rex > S;
4

rex > T;
9
```

This time, however, `R` is bound to the portion of the list after the first three elements:

```
rex > R;
[16, 25, 36]
```

When we match using the vertical bar `|` we can do so only if it is the last punctuation item in a match template. For example, the following attempted match is syntactically *ill-formed*.

```
[F | R, S, T] = [1, 3, 9, 16];
```

rex would report this as a syntax error. On the other hand, the bar would not be used if we wanted to match a list with an exact number of elements:

```
rex > [F, S, T] = [1, 3, 9];
1
```

We must get the number right however, or the match will fail.

```
rex > [F, S, T, X] = [1, 3, 9];
0
```

Also, using an identifier twice with the same left-hand side will fail unless it happens that both occurrences would get bound to the same value.

```
rex > [F, F, S] = [1, 3, 9];
0
```

The above match failed because it is ambiguous whether `F` is getting bound to `1` or `3`.

```
rex > [F, F, S] = [1, 1, 9];
1
```

The above match succeeded despite there being two definitions of `F`, because both definitions are the same. This style is nonetheless regarded as awkward and should be avoided.

We mention these points not because they are so essential in what we will be doing, but because they help further emphasize that = is definition, not assignment. The difference is admittedly subtle: assignment presupposes a memory location containing a value; definition merely identifies a value with an identifier, but there is not necessarily any memory location. So assignment can be treated as definition, if desired, by making sure the location has a value before any use of its value and by not re-assigning the value once established.

Matching in Lists of Lists

The idea of binding variables by matching a template to a list extends naturally to lists of lists. For example, consider the template

```
[ [A, B] | X ]
```

This would match a list of at least one element, the first element of which is a list of exactly two elements. In other words, the only lists it would fail to match would be the empty list and a list that did not begin with an element that is a pair. Let’s test this idea using rex:

```
rex > [ [A, B] | X] = [ [1, 2], 3];
1

rex > A;
1

rex > B;
2

rex > X;
[3]
```

```
rex > [ [A, B] | X] = [1, 2, 3];
0
```

We see that the match failed in the last attempt; the list does not begin with an element that is a pair.

Exercises

5. •• For all possible pairs of pattern vs. list below, which patterns match which lists, and what bindings are defined as a result of a successful match? For those pairs that don't match, indicate why.

patterns	**lists**
`[F | R]`	`[1, 2, 3]`
`[F, S | R]`	`[1, [2, 3] ]`
`[F, S, T]`	`[ [1], 2, 3]`
`[ [F], S]`	`[1, 2 | [3] ]`
`[ [F, S] | R]`	`[ [1, 2], 3]`
`[F, S, T | R]`	`[1, 2, [3, 4] ]`

6. •• For the patterns above, give a word description for the lists that they match.

7. ••• Give an algorithm for determining whether a pattern matches a list. It should be something like the equality checking algorithm.

4.3 Single-List Inductive Definitions

In chapter 2 we mentioned the *fundamental list-dichotomy*: A list is either:

- *empty*, i.e. has no elements, or
- *non-empty*, i.e. has a first element and a rest

A great many, but not all, low-level definitions are structured according to this dichotomy. In defining a function that takes a list as argument, we:

- Define what the function does on the empty list.
- Define what the function does on a typical non-empty list.

Typically in the second case, the non-empty list is expressed in terms of its first and rest: `[F | R]`. The definition in the second case would generally use the function's value for the argument `R` to define the value for argument the larger argument `[F | R]`. This type of definition is called *inductive* or *recursive*. The difference between these two terms is primarily one of viewpoint. In inductive definitions, we think in terms of *building up* from definitions on simpler structures to definitions on the more complex ones, while in recursive definitions, we think in terms of *decomposing* a complex structure into simpler ones.

Let's consider the definition of the function `length` that returns the length of its list argument. Using the list-dichotomy, we must say what length does with an empty list. The obvious answer is to return 0. We express this as a *rewrite rule*:

```
length( [ ] ) => 0;
```

It is called a *rewrite* rule because whenever we see `length( [ ] )` we can just as well rewrite it as `0`. The symbol

```
=>
```

is read "rewrites as". Thanks to the idea of referential transparency, we are evaluating the expression for a value, not for an effect. This rule is called the *basis* of the induction, since it does not convert to an expression involving `length`.

The other part of the dichotomy is a non-empty list. We express this using the generic form `[F | R]`. What is the length of a list of this form. The answer is it is 1 more than the length of `R`. So we give a second rule:

```
length( [F | R] ) => length(R) + 1;
```

Again this is a rewrite rule because whenever we see `length(L)` where `L` is a non-empty list, we can effectively replace it with `length(R) + 1` where `R` is the rest of the list. Because this rule appeals to the definition of `length` for its final value, it is called the *induction rule* rather than the basis.

For example, `length([2, 3, 5, 7])` is replaceable with `length([3, 5, 7]) + 1`. By continuing this replacement process, using one of the two rules each time, and evaluating the final result as a sum, we can a number that is the actual length of the list. This result is called *irreducible*, because it contains no further function applications that could be rewritten.

```
   length([2, 3, 5, 7])
=> (length([3, 5, 7])             + 1)
=> ((length([5, 7])          + 1) + 1)
=> (((length([7])       + 1) + 1) + 1)
=> ((((length([ ]) + 1) + 1) + 1) + 1)
=> ((((          0 + 1) + 1) + 1) + 1)
=> (((               1  + 1) + 1) + 1)
```

```
=> ((                    2  + 1) + 1)
=> (                       3  + 1)
=>                           4
```

Here we have assumed that the + operator is grouped from left to right, so we can only rewrite the + expressions when a numeric value for the left argument is known. In evaluating the + expressions, we are assuming very simple properties. These could be expressed more rigorously themselves using rewrite rules.

Sometimes we do not wish to see the rewrite sequence in this much detail. We use the symbol

```
==>
```

to represent a collapsed sequence of intermediate steps. As a relation, ==> represents the *transitive closure* of the relation =>, as described earlier. For example, we could outline the major plateaus in the above derivation as:

```
    length([2, 3, 5, 7])

==> ((((length([ ]) + 1) + 1) + 1) + 1)

=>  ((((          0 + 1) + 1) + 1) + 1)

==> 4
```

Now let's try another low-level definition, this time for the function `append`. Recall that `append` takes two arguments, both lists, and produces the result of appending the second argument to the first. However, in the spirit of functional programming, neither argument is modified, so the term *append* is slightly misleading; nothing gets appended to the first list in place; instead a new list is created. For example,

```
append([1, 2, 3], [4, 5]) ==> [1, 2, 3, 4, 5]
```

We are in a section on single-list definitions, yet `append` has two arguments. What gives? Regardless of the number of arguments a function has, we should first look for the possibility of using only *one* of the list arguments on which to apply the fundamental list dichotomy. This argument, if it exists, is called the *inductive argument*. In the case of `append`, the first argument rather than the second turns out to be the right choice for the inductive one. Let us see why.

To append a list `M` to an empty list gives the list `M`. This is expressible by the a rule:

```
append( [ ], M ) => M;
```

To append a list `M` to non-empty list, one that matches `[A | L]` say, we observe that the first element of the result list must be the binding of `A`. Furthermore the rest of the result can be obtained by appending `M` to the shorter list `L`, which is the rest of the original list.

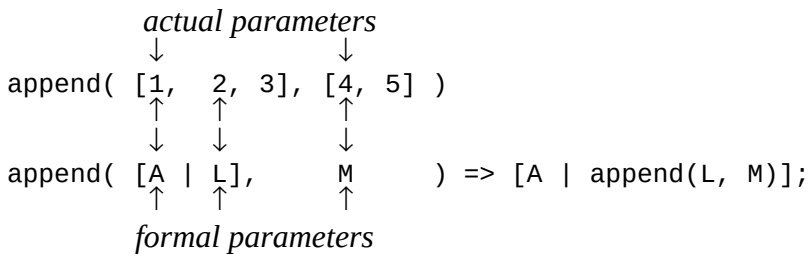
This works out perfectly, in that the parts we get by decomposing the original list are exactly what we need to construct the new one.

```
append( [A | L], M ) => [A | append(L, M)];
```

We can check this by a specific example: In evaluating

```
append( [1, 2, 3], [4, 5] )
```

we see how the arguments, or *actual parameters*, of the expression match up to the *formal parameters* of the rule:



Formal parameter `A` matches the first element `1` of list `[1, 2, 3]`. Formal parameter `L` matches the rest of that list, `[2, 3]`. Formal parameter `M` matches the entire list `[4, 5]`. When we rewrite, or replace the expression with the right-hand side of the rule, the formal parameters carry their values along and the expression that results is constructed from those values:

```
[A | append(L, M)]
```

becomes

```
[1 | append([2, 3], [4, 5])]
```

simply by substituting the value of each variable for the value itself. In the computer science literature, this type of rewriting is sometimes called the ***copy rule*** or ***beta reduction***. Rewriting is certainly more complicated to describe than it is to use. Once we have worked through a number of examples, its use should come fairly naturally. Note that the idea of matching formal and actual parameters is not very language specific; some variation of this idea occurs in most computer languages, although not all languages support decomposing lists by pattern matching.

Continuing the above example, we are thus left with another expression containing `append`. Hence another rewrite is requested, this time with a different set of actual parameters. The process of rewriting continues until we get to apply the first rule for `append`, which leaves us with no further instances of `append` to be rewritten.

```
   append([1, 2, 3], [4, 5])
=> [1 | append([2, 3], [4, 5]) ]
=> [1 | [2 | append([3], [4, 5]) ] ]
```

```
== [1, 2 | append([3], [4, 5]) ]
=> [1, 2 | [3 | append([ ], [4, 5]) ] ]
== [1, 2, 3 | append([ ], [4, 5]) ]
=> [1, 2, 3 | [4, 5] ]
== [1, 2, 3, 4, 5]
```

Here the == steps just recall that these are two ways of writing equivalent expressions. The main difference between this series of rewrites and the earlier `length` series is that these construct a *list* from outside in, whereas `length` constructs a number.

What would have happened had we chose to use the second argument instead as the induction variable? The basis would still have been okay:

```
append( L, [ ] ) => L;
```

However, there is a snag when we get to the induction rule:

```
append( L, [A | M] ) => ??
```

There is no elegant way to use the constructs we have here to achieve the result. The single element A does not start the desired resulting list. Being able to recognize the correct variable for induction is a skill that will develop as we do more exercises.

A confusion often held by newcomers is the difference between the following two expressions:

`append(L, M)` *vs.* `[L | M]`

The expression on the right produces a new list starting with `L` as an element of the result list. This element does not have to be a list, although it could be if the result is going to be a list of lists. In contrast, the expression on the left *always* needs a list for `L` and the elements of `L`, not `L` itself, are elements of the result list. Now let's see what happens if we use the right-hand expression in place of append in one of the preceding examples.

```
rex > [ [2, 3, 5, 7] | [11, 13] ];
[[2, 3, 5, 7], 11, 13]
```

This is quite different from what `append` produces, a list of six elements:

```
rex > append( [2, 3, 5, 7], [11, 13] );
[2, 3, 5, 7, 11, 13]
```

4.3 Rules Defining Functions

In the previous section, we presented two rules for the function `append` that creates a new list having the elements of the second argument appended to the first:

```
append( [ ], M ) => M;
append( [A | L], M ) => [A | append(L, M)];
```

A question that is proper to ask is *in what sense does a set of rules define a function?* For the case of `append`, we can answer this question using the same reasoning that leads to the construction of the rules in the first place: `append` is a function on the set of all lists if it prescribes a correct rewriting for all pairs of lists. Examination of the rules reveals that the second argument plays a very minor role: It is never decomposed. The only thing that is required is that it be a list, in order that the first rule make sense when that argument is returned as the result (the result is supposed to be a list). So we can focus on the first argument, the inductive one.

An arbitrary list is either the empty list or a non-empty list. The space of all lists is exhausted by these two possibilities. The case of the first argument being empty is handled by the first rule, and the other case, an infinite set of possibilities, is handled by the second rule. This reasoning leads us to conclude that *there will never be a pair of lists for which no rule is applicable*. This is certainly a good start toward defining a function. Furthermore, for a given pair of lists, *only one rule is applicable*; there is never ambiguity as to which rule to choose. So this tells we have at least a *partial* function.

But there is another issue in establishing that the rules give us a function. What we have argued above is that there will always be a rule for a given pair of lists. We haven't shown that, once we apply the rule, the ensuing series of rewrites will always *terminate*. In the case of `append`, here is the way to show termination: Notice that if the first rule applies, no expression in need of rewriting is introduced. This is the *basis* of the definition. In other words, for a list of length 0, the rewrite series will terminate. Next consider the case of a non-empty first argument. We see that the length of the argument of `append` on the right-hand side is *one less* than the length on the left-hand side, thanks to having taken away the first element `A` of the first list. In other words, every application of the second rule effectively *shrinks* the first argument by one, figuratively speaking (because we are not modifying the actual argument in any way). Thus, no matter what length we start with in that argument, it will keep shrinking as further rule applications occur. But when it shrinks to length 0, i.e. the empty list, it can shrink no further. The first rule then applies and rewriting terminates.

What we have just described is a narrative version of an *inductive argument*, or proof by induction. More succinctly, it could be captured as follows:

Claim:

For every finite list `L`, `append(L, M)` produces a terminating sequence of rewrites.

Proof:

(Basis): For `L` being `[ ]`, `append([ ], M)` generates a terminating rewrite sequence, since there are no further rewrites according to the first rule.

> (Induction Step): Assume that `append(L, M)` generates a terminating rewrite sequence. Consider the case of an argument one longer, `append([A | L], M)`. According to the second rule, the continuation of the rewriting sequence is based on the sequence for `append(L, M)`. This sequence terminates by assumption, therefore the sequence for `append([A | L], M)` also terminates, it being one step longer.

We will not go through such proofs for most of the functions to be presented. However, it is important that the reader get comfortable with the logic of the proof, in order to be able to construct sound sets of rules.

4.4 Lists vs. Numbers

Some of our functions involve lists, some involve only numbers, and some, such as `length`, involve combinations of both. A useful viewpoint for reasoning is that *natural number* (numbers in the set {0, 1, 2, 3, …}) can be thought of as special cases of lists. Consider an arbitrary element, say •. Then the number *n* can be thought of as a list of *n* of these elements. For example,

0 is `[ ]`
1 is `[•]`
2 is `[•, •]`
3 is `[•, •, •]`
…

Representing numbers in this way is sometimes called the *1-adic* representation. This is, no doubt, the representation for numbers used in the stone age (think of each • as a stone). We are not proposing a return to that age for actual calculation, but we do suggest that this analogy provides a basis for reasoning about numbers and for analogies between various functions. For example, if the function `append` is restricted to lists of stones, then it becomes the *addition* function.

A very important theory, known as *recursive function theory*, starts out by defining functions on natural numbers in this very way. While we do not intend to make explicit use of recursive function theory in this book, the ideas are useful as exercises in creating rule sets, so we pursue it briefly. Recursive function theory typically starts with a *successor function*, which in list terms would be defined by one rule:

```
successor(L) => [• | L];
```

In other words, successor adds one to its argument. We can then define addition by using successor and recursion. But rather than showing an argument lists as `[A | L]`, we would show it as `L+1`. (Since all elements of the list are the same, the identity of A is unimportant.) The definition of `add` would be presented:

```
add(0, N) => N;

add(M+1, N) => add(M, N) + 1;
```

The `M+1` on the left is another form of pattern matching available in rex, and can be viewed as a direct translation of `append` specialized to lists of one type of element:

```
append([ ], N) => N;

append([• | M], N) => [• | append(M, N)];
```

On the right-hand side, the `+1` indicates application of the successor function.

Reasoning about such definitions typically uses induction, in the same way we reasoned about `append`. This style of definition is used to build up a *repertoire* of functions. For example, having defined `add`, we can then define `multiply`:

```
multiply(0, N) => 0;

multiply(M+1, N) => add(multiply(M, N), N);
```

Reasoning that `add` and `multiply` are functions for all natural numbers is essentially the same as reasoning that `append` terminates.

Defining subtraction in this way is a little tricky. If you don't believe me, try it before reading on. We start with a simpler function, `predecessor`. Informally, the predecessor of a number is the number minus 1, but since 0 has no predecessor in the natural numbers, we make its predecessor 0 for sake of convention and completeness. Likewise, we define `subtract`, when the first argument is smaller than the second, to be 0. This is known in the literature as *proper subtraction*. In the spirit of building up definitions from nothing but successor, we can't appeal to a comparison operator yet.

```
predecessor(0) => 0;

predecessor(M+1) => M;
```

Now we can define `subtract` using the second argument as an induction variable:

```
subtract(M, 0) => M;

subtract(M, N+1) => subtract(predecessor(M), N);
```

Actually, we could use a different set of rules and bypass `predecessor`:

```
subtract(M, 0) => M;

subtract(0, N) => 0;

subtract(M+1, N+1) => subtract(M, N);
```

However, this rule set is trickier since it uses two induction variables simultaneously. Doing so can be more error-prone unless you have a very clear idea of what you are doing.

Note also the following point about the second set of subtract rules: This is our first set of rules where there was overlap between the applicability of the rules. In particular, `subtract`(0, 0) could invoke both the second and the first rules. Fortunately in the present case, the result is the same either way. In order to avoid possible misinterpretations in the future, and to actually make rule definitions simpler, we adopt the following convention:

> **rex rule-ordering convention:**
>
> In rex, the rules are tried in top-to-bottom order. The first applicable rule is used, and subsequent rules, while they might have been applicable on their own, are not considered if an earlier rule applies.

As a simple example where this makes a difference, consider defining a function `is_zero` that tests whether its argument is 0:

```
is_zero(0) => 1;
is_zero(N+1) => 0;
```

Under the rule-ordering convention, we could have used:

```
is_zero(0) => 1;
is_zero(N) => 0;
```

since the second rule will never be used if the argument is 0, thanks to the rule-ordering convention. Similarly, define `non_zero`:

```
non_zero(0) => 0;
non_zero(N) => 1;
```

Having defined `subtract`, we can define `less_than_or_equal` (for natural numbers):

```
less_than_or_equal(M, N) => is_zero(subtract(M, N));
```

We can define equality in many ways, for example using rex's argument matching capabilities:

```
equal(M, M) => 1;
equal(M, N) => 0;
```

The second rule is valid only by virtue of the rule-ordering convention; two different variables can be bound to the same value.

If this type of matching were not available, we could still construct an equality predicate in other ways, e.g.

```
equal(M, N) => non_zero(multiply(less_than_or_equal(M, N),
                                 less_than_or_equal(N, M));
```

In other words, two numbers are equal if, and only if, each is less than or equal to the other.

Convention: Henceforth, we will use the typical symbols for the functions we have defined, rather than their "spellings", e.g. `+` instead of `add`, `*` instead of `multiply`, `<=` instead of `less_than_or_equal`, `==` instead of `equal`, etc. Keep in mind that the built-in `-` is ordinary signed subtraction, rather than proper subtraction as defined above.

Exercises

1 • Give rules that define the function `zero` that invariably returns the result 0. Similarly, show that for any number you can name, e.g. `five`, `one_thousand`, etc., you can define a function that invariably returns that number, without actually using the number directly in the definition.

2 •• Give rules that define the function `power` that raises a number to a power, using `multiply` and recursion. For example, `power(2, 3)` would ultimately rewrite to 8.

3 •• Give an inductive argument that shows that the rules given for `length` establish a function on the set of lists.

4 ••• Define rules that define the function `superpower` that bears the same relation to `power` as `power` does to `multiply`. For example, `superpower(2, 3)` ==> 16 and `superpower(2, 4)` ==>65536.

5 •••• Continuing in the above vein, we could define `supersuperpower`, `supersupersuperpower`, and so on, ad infinitum. Give rules for a three-argument function that effectively takes the number of "super"s as a first argument and applies the corresponding function to the remaining arguments. The function you have defined is a version of what is commonly known as "Ackermann's function".

4.5 Guarded Rules

One purpose in preferring a sequential list of rules to a single comprehensive rule is clarity and readability. In some cases, however, clarity is best served by conditioning the

applicability of a rule on other than the *form* of the arguments. The concept of a **guard** is useful to provide this additional clarity. A rule is *guarded* if it has the form

lhs => guard ? *body*;

format of a guarded rule

Here *guard* ? *body* is an expression for the *rhs* as before. The question-mark separator is what distinguishes this form. Both *guard* and *body* are terms that can ultimately rewrite to values. The rule as a whole is called a **guarded rule**. The meaning of a guarded rule is as follows:

> The rule is considered applicable only if the arguments match as before, *and then* only if the value of *guard* ultimately rewrites to 1 (true). In this case the *lhs* rewrites to the value of *body*.

If the condition of applicability does not hold, then the rule is unusable and we must appeal to later rules to rewrite a given term. Note: the rule ordering convention is still in effect; a later rule is applied only if all previous rules don't apply.

An example of guarded rules occurred in our first rex example, the function for testing whether a number is prime. Here is another example.

Euclid's Algorithm

Euclid's algorithm is an algorithm for finding the greatest common divisor (*gcd*) of two natural numbers. The rules are:

```
gcd(0, Y) => Y;
gcd(X, Y) => X <= Y ? gcd(Y-X, X);
gcd(X, Y) => gcd(Y, X);
```

Euclid's Algorithm

The second rule is guarded, using the `<=` (less than or equal) operator of rex. By convention, the third rule, which contains no guard, is applicable only if the first two rules are not applicable, i.e. only in the case that `X` is not 0 and `X` is greater than `Y`.

There are ways to speed up the computation, e.g. by using the operator `%` (remainder or *modulus*). This amounts to repeated subtraction, in place of division.

Let us trace the rewrite behavior of these rules on a test case, `gcd(18, 24)`. Since 18 factors into 2*3*3 and 24 factors into 2*2*2*3, we can anticipate the result will be 2*3 = 6.

```
gcd(18, 24) ==>
gcd(6, 18)  ==>
gcd(12, 6)  ==>
gcd(6, 12)  ==>
gcd(6, 6)   ==>
gcd(0, 6)   ==>
6
```

Why does Euclid's algorithm work? The rationale for the algorithm is based on two observations:

> The actual greatest common divisor of the two arguments never changes.
>
> Each time one of the second or third rules is applied, one of the arguments will soon decrease.

The first fact is due to some reasoning about division: If a number *Z* evenly divides both *X* and *Y*, and *X* <= *Y*, then *Z* also divides *Y* - *X*. So if the first rule becomes applicable, we see that *Y* is the greatest common divisor, since it is obviously the largest divisor of both *Y* and 0.

The second fact may be seen from the rule structure: If *X* <= *Y*, (and *X* is not 0, otherwise the first rule would have been used) then *Y* - *X* is clearly less than *Y*. On the other hand, if *X* > *Y*, then, based on the third rule, the second rule will be tried with *X* and *Y* reversed.

Because one of the arguments is bound to decrease and stop at 0, we have that the term will eventually be reduced to a case where the first rule applies, i.e. Euclid's algorithm always terminates.

Exercises

1 ••• Continue the development of recursive function theory by defining the following, using guards where it is helpful:

```
mod(M, N)
```

is the remainder after dividing `M` by `N` (use the convention that `mod(M, 0)` is 0. (In rex, mod`(M, N)` is available as `M % N`, also read "M modulo N", or "M mod N".)

```
div(M, N)
```

is the quotient obtained by dividing `M` by `N` (again with `div(M, 0)` defined to be 0). (In rex, `div(M, N)` is available as `M / N`.)

2 ••• Show how Euclid's algorithm can be "sped up" if `mod` were available as a primitive function.

4.6 Conditional Expressions

Although not absolutely essential due to the rule notation, for convenience rex allows the Java language notation for conditional expressions:

```
C ? A : B
```

is an expression that has the value A if C rewrites to a number other than 0, otherwise the value is B. This is an extension of the guard idea, providing an immediate alternative in case the guard is false, rather than requiring resolution through another rule. Although the same effect can be achieved with guards and additional rules, the conditional expression is a self-contained unit. As an example, an alternative set of rules for *gcd* would be

```
gcd(0, Y) => Y;
gcd(X, Y) => X <= Y ? gcd(Y-X, X) : gcd(Y, X);
```

4.7 Equations

As noted in the previous chapter, rex supports the notion of defining functions by equations as well as rules. By an *equation*, we mean a single expression that captures all cases. While it would certainly be adequate to give a single rule instead of an equation, using an equation has a signal of finality about it: there will be *no further rules* defining this function. Also, in terms of the rex implementation we provide, an equation will execute more efficiently since there will be no pattern-matching superstructure. Finally, the handling of equations vs. rules is different in the interactive environment provided: If an equation for a function is re-entered, it will be taken as a *re-definition*, whereas if a rule for a function is re-entered, it will be taken as an *additional* rule, rather than as a replacement.

Typically, conditional expressions are used to simulate what would have been separate rules. For example, a single equation defining *gcd* of the previous section would be:

```
gcd(X, Y) = X == 0 ? Y : X <= Y ? gcd(Y-X, X) : gcd(Y, X);
```

4.8 Equational Guards

The rex language allows a guard to consist of an equation that binds a variable to a value for use in the expression that follows. Such variables are used in the *rhs* in the same way any *lhs* variable would be used. A basic *equational guard* takes the form:

Var = Expression,

The meaning of this equation is that *Var* is bound to *Expression*. This simple form of equational guard always succeeds. However, equational guards that involve "matching" might not, as will be explained momentarily.

The main use of the simple form of equational guard above would be to give a name to the value of a complicated expression, for one of the following purposes:

- to *avoid multiple evaluations* of the expression, even though its value is used multiple times:

  ```
  f(X) => Y = sqrt(X), g(Y, Y);
  ```

  Here `sqrt` is supposed to be a function that is relatively expensive to evaluate.

- to *document* the meaning of the expression by giving the variable a descriptive name:

  ```
  f(X, Y) => First_Vowel = find(vowel, X),
             g(First_Vowel, Y);
  ```

  Here the expression on the *rhs* of the equation for `First_Vowel` could have been substituted directly as the argument of `g`. However, then the documentary aspect of the name would be lost.

- to *redefine the scope* of variable `X`, which gives an argument variable a value different from the one it had in the function call. This can be used to provide "wrappers" for expressions that we wish to leave intact but for which we don't wish to use the given argument variables.

  ```
  f(X, Y) => X = g(X, Y), X*Y;
  ```

  Here the `X` used in expression `X*Y` is not the argument `X`, but rather the value of `g(X, Y)`.

Equational guards can involve binding multiple variables in a single equation through the use of the list notation.

```
[X, Y, Z] = [1, 2, 3],
```

is a guard that binds each of `X`, `Y`, and `Z` simultaneously. If the result of an evaluation is a list, then this type of guard can be used to select elements from the list, in the manner used in argument pattern matching:

```
[X, Y, Z] = g(W),
```

means that `g(W)` is expected to return a list of three elements, and the variables on the *lhs* get bound to these elements. Here is one place an equational guard can

fail: If the list returned does not have exactly three elements. In general, a match must be possible with the *lhs* and the list returned. Similarly,

```
[X, Y | Z] = g(W),
```

matches a list with *at least two* elements. Variable `Z` is bound to the list after the first two elements.

Equational guards may also be cascaded:

$lhs_1 = Expression_1, lhs_2 = Expression_2, \ldots, lhs_N = Expression_N,$

If any of the left-hand sides fails, the guard is considered to have failed, in which case rex will try the next rule, if there is one. If there are no more rules, then the function returns a distinguishable failure value. When other functions operate on such values, they typically return failure values themselves.

Example – Computing *mod* from first principles

One way to compute the *mod* or remainder function is as follows:

```
mod(0, K) => 0;

mod(N+1, K) => mod(N, K)+1 == K ? 0 : mod(N, K) + 1;
```

Here we define `mod` by induction on the first variable, basing the value of `mod(N+1, K)` on the value of `mod(N, K)`. The unpleasant part of this definition is that potentially the *rhs* sub-expression `mod(N, K)+1` must be computed twice. A way to avoid this would be to introduce a variable, say `R`, to stand for the value of `mod(N, K)+1` then use the value of `R` a second time if necessary. The following alternate set of rules accomplishes this:

```
mod(0, K) => 0;

mod(N+1, K) => R = mod(N, K)+1, (R == K ? 0 : R);
```

equational guard defining `R`

The use of equational guards provides a style found in mathematically-oriented texts. It is often convenient to introduce variables to stand for large expressions. So the text would read:

Let R = … *some expression*… .

Then later on either the text or another expression can use R to represent the value of that expression.

A similar style often used in writing is "… R …, *where* R = … *some expression*… ". Both forms are especially convenient when R is referred to more than once. Some

programming languages provide a **let** construct or a **where** construct to achieve the same end. The construct **letrec** ("let recursive") is also used when the variable is defined recursively in terms of itself.

4.9 More Recursion Examples

As much as possible, we would like to use powerful concepts such as recursion to simplify our work. When a problem involves structures such as lists and numbers that can be arbitrarily large, often the only reasonable way to get a handle on the problem is to deal with simple cases directly, and deal with the general case by breaking it down into simpler cases that we have assumed can be handled. The tool of recursion can work like magic in the hands of the knowledgeable. Therefore, the **recursion manifesto** is

Let recursion do the work for you.

We applied this principle in several previous examples, but it is time now to really exercise it

Range: Creating a List

The function `range` synthesizes a list from two numbers, *M* <= *N*. Specifically,

`range(M, N)` yields [M, M+1, …, N].

If M > N, the result is specified to be the empty list. Rules that define *range* are:

```
range(M, N) => M > N ? [ ];

range(M, N) => [M | range(M+1, N)];
```

Scale: Transforming a List

Suppose we wish to multiply each element in a list of numbers by a constant K, returning a new list. A function `scale` is to be devised such that `scale(K, L)` is this new list. Here we let recursion work for us, by decomposing into the empty and non-empty list cases and only coding the scaling of the first element in the latter.

```
scale(K, [ ]) => [ ];
scale(K, [ A | L ] ) => [ K*A | scale(K, L) ];
```

Illustration:

```
scale(3, [7, 5, 2])           =>
[ 21 |  scale(3, [5, 2])]     =>
[ 21, 15 | scale(3, [2]) ]    =>
[ 21, 15, 6 | scale(3, [ ]) ] =>
[21, 15, 6 | [ ] ]            =>
[21, 15, 6]
```

> Note: The philosophy of "functional programming" (which is what we do in rex) is that we never modify lists in place. We only create new lists, possibly out of existing lists. But the original list remains intact as long as needed.

We mention the above philosophy explicitly, as it is quite possibly foreign, depending on the manner to which one is exposed to programming.

The Map Functions

In the previous chapter, we showed an alternate definition of `scale`, which used `map`. But how would `map` be defined from first principles? It is essentially the same pattern as `scale`, except that the first argument is a function, not a number:

```
map(F, [ ] ) => [ ];

map(F, [A | X]) => [ F(A) | map(F, X) ];
```

For mapping over two lists simultaneously the rules are:

```
map(G, [ ], _) => [ ];

map(G, _, [ ] ) => [ ];

map(G, [A | X], [B | Y]) => [ G(A, B) | map(G, X, Y) ];
```

mapping a function across a pair of lists

The presence of two basis cases allows us to deal with the case where the lists are not the same `length`. As soon as one list is reduced to [ ], the recursion will stop, so the length of the result will be that of the shorter of the two argument lists.

Reducing a List

Quite often we have need to compute the result of a binary (i.e. two-argument) operator being applied to "reduce" a list to a single item. An example would be to "add up" the elements of a list of numbers. The rules specialized to the add operator might be:

```
add_up( [ ] ) => 0;
```

```
add_up( [A | X] ) => A + add_up(X);
```

The same technique could be used for multiplying the elements of a list, or to applying any binary function H to a list in the same pattern. We do need to specify a base value for the case of the empty list. The general form of the rules for reducing a list using operator `H` are:

```
reduce( _, Base, [ ] ) => Base;

reduce( H, Base, [A | X] ) => H(A, reduce(H, Base, X));
```

reducing a list by a function `H`*, together with a base value*

This set of rules "biases" the reduction to the right, i.e. the result of

$\texttt{reduce(H, Base, [}X_0\texttt{, }X_1\texttt{, …, }X_{N-1}\texttt{])}$

will be that of

$\texttt{H(}X_0\texttt{, H(}X_1\texttt{, …, H(}X_{N-1}\texttt{, Base) …))}$

Horner's Rule

Consider the requirement of evaluating polynomials

$$a_0*x^n + a_1*x^{n-1} + \ldots + a_{n-1}*x^1 + a_n*x^0$$

where we are given a list with low-order coefficient first $[a_n, a_{n-1}, \ldots, a_1, a_0]$ and a value x. A method that is commonly used to reduce the number of multiplications is to evaluate the polynomial using the following scheme, known as *Horner's Rule*:

$$(\ldots((a_0*x + a_1)*x + \ldots + a_{n-1})*x + a_n$$

This has the computational advantage of not computing the large powers separately. Instead they are folded into the multiply-add's that have to be done anyway. This elegant scheme is concisely represented by the following rex recursion:

```
horner(X, [ ]) => 0;

horner(X, [A | L]) => A + X * horner(X, L);
```

Principle of Radix Representation

This is actually an application of Horner's rule. Here we assume that a list represents a radix numeral for a number, least-significant digit first. We wish to construct rules for a function *value* that computes the number. The radix r representation of a number is a sequence of digits

$$d_{n-1}\ d_{n-2}\ \ldots\ d_2\ d_1\ d_0$$

where each d_i is in a digit in the set {0, 1, …, r-1}. The number represented by the sequence is the value of the expression

$$d_{n-1}*r^{n-1} + d_{n-2}*r^{n-2} + \ldots + d_2*r^2 + d_1*r^1 + d_0*r^0$$

For example, if r = 2, we have the binary representation, where each d_i is either 0 or 1. A numeral such as

1 1 0 1

represents the number designated by the decimal numeral 13, since

$$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 == 13$$

and

$$1*10^1 + 3*10^0 == 13$$

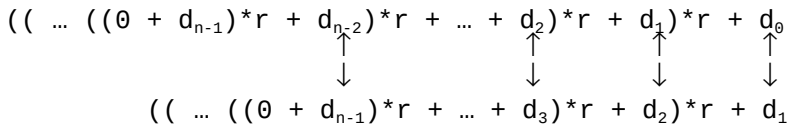
It is important to notice that the expression for the value can also be computed another way, in a nested fashion using *Horner's Rule*:

$$((\ldots ((0 + d_{n-1})*r + d_{n-2})*r + \ldots + d_2)*r + d_1)*r + d_0$$

The function `value` will accept a list of digits [d_0, d_1, d_2, …, d_{n-2}, d_{n-1}] and return the value. It will use the Horner's rule version of the expression. The idea is that we can compute values of a sequence by multiplying by r the value of all but the first element of the sequence (treated as a numeral with one fewer digit) then adding the remaining digit. In other words, notice that the sub-expression of the above

$$(\ldots ((0 + d_{n-1})*r + d_{n-2})*r + \ldots + d_2)*r + d_1$$

looks a lot like the *original* expression. The only difference is that the d subscripts have been "shifted" by one position:



That is,

value([d_0, d_1, d_2, …, d_{n-2}, d_{n-1}]) ==

d_0 + r * value([d_1, d_2, …, d_{n-2}, d_{n-1}])

In order to set things up to apply recursion, we only need to identify the sequence `[ Digit | Digits ]` with [d_0, d_1, d_2, …, d_{n-2}, d_{n-1}] to convert this equation into a rex rule, adding a new variable `Radix` for the radix:

```
value( [ Digit | Digits ], Radix ) =>

         Digit + Radix * value( Digits, Radix );

value( [ ], Radix ) => 0;
```

Radix Interpretation of a list of digits, Least-significant first

Here we have added a basis rule for the empty sequence.

Let us check this with the binary numeral 1 1 0 1, which we said has a value of 13. The list representation, least-significant digit first, will be [1, 0, 1, 1]. By the rules:

```
   value( [1, 0, 1, 1], 2 )
=> 1 + 2 * value( [0, 1, 1], 2 )
=> 1 + 2 * (0 + 2 * value( [1, 1], 2 ))
=> 1 + 2 * (0 + 2 * (1 + 2 * value( [1], 2 )))
=> 1 + 2 * (0 + 2 * (1 + 2 * (1 + 2*value( [ ], 2 ))))
=> 1 + 2 * (0 + 2 * (1 + 2 * (1 + 2*0)))
=> 1 + 2 * (0 + 2 * (1 + 2 * (1 + 0)))
=> 1 + 2 * (0 + 2 * (1 + 2 * 1))
=> 1 + 2 * (0 + 2 * (1 + 2))
=> 1 + 2 * (0 + 2 * 3)
=> 1 + 2 * (0 + 6)
=> 1 + 12
=> 13
```

Now consider the inverse function for radix conversion: Given a number, produce a list of its digits in a given radix notation. For now, we will develop the list least-significant digit first. We can either apply `reverse` to the result, or use a later technique to get the digits in most-significant digit first order. We can find the least-significant digit by using the integer remainder function *mod*: `N % M` is the remainder that occurs when `N` is divided by `M` using integer division /. We can find the remaining digits by applying the same process to the quotient of the number and the radix. We are letting recursion do the work

for us. We use an auxiliary function digits1 so that our function `digits` handles the case `0` in such a way that the result is `[0]` rather than an empty list. The rules are thus:

```
digits(0, Radix) => [0];
digits(N, Radix) => digits1(N, Radix);

digits1(0, Radix) => [ ];
digits1(N, Radix) => [ (N % Radix) | digits1(N / Radix, Radix) ];
```

Forming the digit list of a natural numbering a given radix

The "Radix" Principle

Although most of the computation with which we will be concerned is "digital" in nature, we use the term "radix principle" to connote algorithmic techniques that rely specifically on the data being representable by a series of digits, such as in radix representation. Some algorithms rely on this fact for efficiency, while others do not. The radix principle makes it possible to do arithmetic with reasonable efficiency. If, for example, all arithmetic were done using tally representation, not much useful computation would get done. We already saw how much space saving was afforded by using radix notation instead of tallies. Now consider the time saved in doing arithmetic with a radix representation instead of tallies. The addition of two *arbitrarily-long* binary numerals, represented as lists, least-significant digit first, can be expressed using the following rules:

```
add_bin(X, Y) => add_bin(X, Y, 0);

add_bin([ ], X, Carry) => add_digit(X, Carry);
add_bin(X, [ ], Carry) => add_digit(X, Carry);

add_bin([A | X], [B | Y], C) =>
  Sum_Digit = (A+B+C) % 2,
  Carry = (A+B+C) / 2,
  [Sum_Digit | add_bin(X, Y, Carry)];

add_digit([ ], 0) => [ ];
add_digit([ ], 1) => [1];
add_digit([A | X], C) =>
  Sum_Digit = (A+C) % 2,
  Carry = (A+C) / 2,
  [Sum_Digit | add_digit(X, Carry)];
```

Adding two arbitrarily-long binary numerals, least-significant digit first.

The core of these rules is the function `add_bin` of three arguments, two sequences and a carry bit. If one of the sequences is empty, then `add_digit` takes over to finish off the addition. Otherwise there is a rule for each combination of first digits in each sequence and for the carry. These rules produce a digit of the resulting sequence, followed by a recursive call to `add_bin` with a new carry value.

Notice that we could, if desired, avoid using the `%` and `/` functions by enumerating the eight different possibilities of digits for `A`, `B`, and `C`.

Examples of other techniques that we will study that make use of, or rely on, the radix principle are:

Fast multiplication by the "Russian peasants' principle"

radix sort, described in *Complexity of Computing*

multiplexors, described in *Computing Logically*

barrel shifters (described in *Finite-State Computing*)

Fast Fourier Transform

The radix principle represents an idea that should not be overlooked when developing efficient algorithms. Here we will show the Russian peasants' principle. Suppose we wish to raise a number to an integer power N. The simple way to do this is to multiply the number by itself N times. This thus requires N multiplications. A more clever way to achieve the same end is to repeatedly square the base number, and selectively multiply some of the results to achieve a final product. Repeatedly squaring the original number gives us powers of two of that number. That is:

$N, N^2, (N^2)^2, ((N^2)^2)^2, \ldots$

is the same as

$N^1, N^2, N^4, N^8, \ldots$

To achieve an arbitrary power of N, say N^k, we can represent k in binary. Then we select the powers of N to powers of 2 according to the bits in the binary representation that are 1. For example, if k were 19, its binary representation would be 10011. The corresponding powers of N are then N^{16}, N^2, N^1. When we multiply these together, we get the desired product $N^{16+2+1} = N^{19}$.

Fortunately, we do not have to put these powers into a list. We can simply multiply in the ones we need as we decompose k into binary. The Russian peasants' approach, expressed in rex, would be:

```
power(X, 0) => 1;

power(X, Y) => even(Y) ? power(X*X, Y/2);

power(X, Y) => X * power(X*X, Y/2);
```

where function `even` tests whether its argument is even or odd. The binary decomposition of the second argument is taking place by dividing it by 2 on each recursion step.

A further example of the radix principle occurs after the following brief presentation of sorting techniques.

Insertion Sorting a List

Arranging a list so that the elements appear in increasing order is known as "sorting" the list. As explained above, in functional programming, the original list is not disturbed. Instead a new list is created that has the same elements in the appropriate order. There are dozens of ways to do it. Here are a few:

Function `insertion_sort` sorts a list by repeatedly inserting an element where it belongs:

To sort an empty list, return the empty list:

```
insertion_sort([ ]) => [ ];
```

To sort a non-empty list, `insertion_sort` all but the first element (using recursion), then insert the first element into its proper place:

```
insertion_sort([F | R]) => insert(F, insertion_sort(R));
```

Overall, recursion does most of the work in `insertion_sort`. However, we still need to define `insert`.

Inserting an element into its proper place in an empty list just gives the list with one element:

```
insert(A, [ ]) => [A];
```

To insert an element into a non-empty list, compare the element with the first element of that list. The resulting list starts with one or the other, and recursion takes care of inserting the other element in the remaining list:

```
insert(A, [B | X]) =>   // note: [B | X] is assumed to be ordered
  A < B ?
    [A, B | X]
  : [B | insert(A, X)];
```

This is a fine example of letting recursion do the work for you.

Selection Sorting a List

An example of a different sort is `selection_sort`which sorts a list by repeatedly selecting the minimum of the unsorted remaining elements and putting it next.

To sort an empty list, return the empty list

```
selection_sort([ ]) => [ ];
```

To sort a non-empty list, an equational guard comes in handy. First get a list with the minimum as the first element, and the rest of the elements as the rest of that list. Call this list `[M | R]`. Return the minimum `M` followed by the result of sorting `R`, the rest of that list (letting recursion do that work):

```
selection_sort(L) =>
    [M | R] = select_min(L),
    [M | selection_sort(R)];
```

Function `select_min` is designed to work only on *non-empty* lists L It brings the minimum of the list to the first position.

The minimum of a list of one element is at the first

```
select_min([A]) => [A];
```

For a list with at least two elements, retain the first element and apply `select_min` to the remaining elements, then return a new list with the retained element and the first element of the result properly ordered:

```
select_min([A | L]) =>
    [B | R] = select_min(L),
    (A < B ? [A, B | R] : [B, A | R]);
```

Merge Sorting a List

Merge sorting is another way to sort. We will show later that it has substantially fewer rewrite steps than either of the sorts introduced prior. By "merging", we mean operation of creating, from two sequences already in order, a longer sequence containing the elements of both sequences. This can be done easily by examining only the first elements of residual unmerged sequences and choosing the smaller one for output, until both sequences have been decimated.

Our implementation of function `merge_sort` works in the following way:

To sort:

> A non-empty list to be sorted is made into a list of 1-element lists. These lists are merged together a pair at a time using `merge_pairs`. This gives us lists of length at most 2. Then the process is repeated, merging pairs of those lists to get half as many lists that are twice as long. This is done in successive stages until only one list is left. That list is the sorted list.

To merge two lists:

> If the either list to be merged is empty, return the other list.
>
> Otherwise, compare the first elements of each list. Return a new list starting with the smaller element and followed by the result of merging the remaining elements.

The `merge_sort` function, expressed in rex, is given below:

First the initial list is transformed to a list of 1-element lists then those lists are merged repeatedly.

```
merge_sort(List) = repeat_merge( map((X) => [X], List ) );
```

Function `repeat_merge` merges pairs in a list of lists until there is only one list left.

```
repeat_merge([A]) => A;              // only one list left

repeat_merge(Lists) =>               // more than one list left
  repeat_merge( merge_pairs(Lists) );
```

Function `merge_pairs` merges pairs of lists in a list until none is left. It is similar to a `map` application, except that the function being mapped (`merge`) is called on successive pairs from a single list rather than on pairs from two different lists.

```
merge_pairs([ ]) => [ ];                    // no more lists

merge_pairs([A]) => [A];                    // only one list

merge_pairs([A, B | L]) => [merge(A, B) | merge_pairs(L)];
```

Function `merge` creates a single ordered list from two ordered lists.

```
merge(L, [ ]) => L;

merge([ ], M) => M;

merge([A | L], [B | M]) =>
```

```
  A <= B ? [A | merge(L, [B | M])] : [B | merge([A | L], M)];
```

Below is a coarse trace of `merge_sort` in operation:

```
merge_sort([5, 1, 2, 7, 0, 4, 3, 6]) ==>

repeat_merge([ [5], [1], [2], [7], [0], [4], [3], [6] ]) ==>

repeat_merge(merge_pairs([ [5], [1], [2], [7], [0], [4], [3], [6]
]))

repeat_merge([ [1, 5], [2, 7], [0, 4], [3, 6] ]) ==>

repeat_merge(merge_pairs([ [1, 5], [2, 7], [0, 4], [3, 6] ])) ==>

repeat_merge([ [1, 2, 5, 7], [0, 3, 4, 6] ]) ==>

repeat_merge(merge_pairs([ [1, 2, 5, 7], [0, 3, 4, 6] ])) ==>

repeat_merge([ [0, 1, 2, 3, 4, 5, 6, 7] ]) ==>

[0, 1, 2, 3, 4, 5, 6, 7]
```

Radix Sorting a List

We conclude the set of sorting examples with a method based on the radix principle. For this method, we assume that the numbers are non-negative integers. Sorting is based on comparing bits of the numbers, from lowest to highest. As splitting and regrouping is done for each bit, the numbers remain sorted on lower-order bits. Sorting is complete after the numbers are regrouped on the highest order bit.

```
// To sort, we sort based on the number of bits,
// from lowest order to highest

radix_sort(L) = radix_sort(0, numBits(L)-1, L);


// Sort on the Ith bit, then on the remaining bits

radix_sort(I, N, L) = I > N ? L : radix_sort(I+1, N, split(I,
L));


// split the list into two based on the Ith bit,
// then append the results

split(I, L) = append(drop((X)=>bit(I, X), L),
                     keep((X)=>bit(I, X), L));


// bit(I, X) gives the I-th bit of X

bit(I, X) = I == 0 ? X%2 : bit(I-1, X/2);
```

```
// find the maximum number of bits across all numeral in list

numBits(L) = ceilLog2(reduce(max, -Infinity, L));


// find the number of bits required to represent a numeral

ceilLog2(N) = N == 0 ? 0 : 1 + ceilLog2(N/2);
```

Further discussion of sorting methods appears in the chapter on Computational Complexity.

Exercises

Wherever possible, adhere to the recursion manifesto in the following:

1 • Give a set of rules for a function that computes the list of squares of each of a list of numbers. (This could be done with `map`, but do it from scratch instead.)

2 •• Give a set of rules for computing the sum of a list of numbers; for computing the product of a list of numbers. (This could be done with `reduce`, but do it from scratch instead.)

3 •• Using your function `days_of_month` constructed in an earlier exercise, give rules for the function `total_days` that takes as an argument a list of months and returns the sum of the days in those months.

4 •• Give a set of rules for computing the average of a list of numbers (use 0 for the average of an empty list).

5 •• Indicate two different ways to compute the sum of the squares of a list of numbers.

6 •• Give rules that define the function `flip`, that operates on lists, and exchanges successive pairs of elements. If there is an odd number of elements, the last element is left as is. For example:

```
flip([1, 2, 3, 4, 5, 6, 7]) ==> [2, 1, 4, 3, 6, 5, 7]
```

Suggestion: Use a rule that matches on the first two elements, rather than just one:

```
flip([A, B | L]) => … ;
```

7 ••• Give rules for the function `at_least` that tells whether a list has at least a certain number of elements. For example:

```
at_least(3, [1, 2, 3]) ==> 1

at_least(3, [1, 2]) ==> 0
```

Avoid counting all of the elements of the list. This is unnecessarily inefficient for large lists.

8 ••• Like the previous problem, except `at_most`.

9 •• The function `select` has the property of selecting the I^{th} element of a list, $I >= 0$, or returning the value of a special parameter `Other` if there is no such element (the list is not long enough). That is,

`select(I, [X0, X1, …, XN-1], Other)` ==> X_I if $I < N$

`select(I, [X0, X1, …, XN-1], Other)` ==> Other if $I >= N$

Give a set of rules for *select*.

10 •• The function `find_index` has the property of computing the index of the first occurrence of a given element within a list. If there is no such occurrence, -1 is returned. For example,

```
find_index('d', ['a', 'b', 'c', 'd', 'e']) ==> 3
find_index('a', ['a', 'b', 'c', 'd', 'e'] ==> 0
find_index('g', ['a', 'b', 'c', 'd', 'e']) ==> -1
```

Give a complete set of rules for `find_index`.

11 ••• Give rules for a function `remove_duplicates` that removes all duplicates in a list. For example

```
remove_duplicates([1, 2, 1, 3, 1, 2, 3]) ==> [1, 2, 3]
```

12 •• Give rules for a function that gives the value of a list representing the 2-adic representation of a number, least-significant digit first, using the digits 1 and 2.

13 ••• Give rules for a function that gives the list representation of a number in 2-adic form, least-significant digit first.

14 ••• Give rules for a function that produces the list of prime factors of a natural number. For example

```
factors(72) ==> [2, 2, 2, 3, 3]
```

15 •• Using functions above, give rules for a function that produces the unique prime factors of a natural number. For example

```
unique_factors(72) ==> [2, 3]
```

16 •• Give rules for the function `subst` that makes substitutions in a list. Specifically, `subst(A, L, R)` returns a new list that is like list `L` except that whenever `A` would have occurred as a member of `L`, `R` occurs instead.

17 •• By adding an extra argument, and assuming integer functions `mod` and `div`, generalize the function `add_bin` to a function that adds in an arbitrary radix.

18 ••• Devise a function that will *multiply* two numbers represented as a list of bits, least-significant-bit first. Notice that this function has some advantage over the standard multiplication function found in most programming languages, namely that it will work for arbitrarily-large numbers.

19 ••• Sometimes we use numeral systems of mixed radix. For example, in referring to time within a given month, we could use expressions of the form D:H:M:S for days, hours, minutes, and seconds. H ranges from 0 to 24, M from 0 to 59, and S from 0 to 59. To compute the number of seconds from the start of the day corresponding to a given time, we'd compute:

```
S + 60*(M + 60*(H+24*D)).
```

Generalize this mixed radix computation by giving rules for a function *value* that takes as arguments two lists, one giving the ranges and another giving the ordinal numbers within these ranges. For example, in the current case we would call

```
value( [S, M, H, D], [1, 60, 60, 24] )
```

20 ••• Devise a function that will *divide* one number represented in binary by another, yielding a quotient and a remainder. This function should return the pair of two items as a list. Do the division digit-by-digit, don’t convert to another form first.

21 •• The function `keep` takes two arguments: a predicate and a list: `keep(P, L)` is the list of those items in `L` such that `P` is true for the item. For example,

```
keep(odd, [1,3,2,4,6,7]) ==> [1,3,7]
```

Provide rex rule definitions for `keep`.

22 •• The function `drop` is like function keep above, except that the items for which `P` is not true are kept. For example,

```
drop(odd, [1,3,2,4,6,7]) ==> [2,4,6]
```

Provide rex rule definitions for `drop`.

23 •• The function `select` takes two arguments: a list of 0's and 1's and another list, usually of the same length. `select(S, L)` is the list of those items in `L` for which the corresponding element of `S` is 1. For example,

```
select([1,0,0,1,1,0], [1,3,2,4,6,7]) ==> [1,4,6]
```

Provide rex definitions for `select`.

24 •• *Iterated function systems* are used for producing so-called "fractal" images. These entail applying a function repeatedly to an initial seed argument. Let

$F^N(X)$

denote

`F(F(F…(F(X))…))`
N applications of `F`

including the definition:

$F^0(X) = X$.

Give rewrite rules for the function `iterate` informally defined by:

`iterate(N, F, X)` ==> $F^N(X)$

25 ••• Restate the definition of *Ackermann's function* using *iterate*.

26 ••• By **indefinite iteration** we mean iteration that stops when some condition is true, rather than by iterating a pre-determined number of times. The condition for stopping is best expressed as a predicate, say P. Give the rewrite rules for a function

```
iterate(P, F, X)
```

defined to compute

$F^{\mathbf{n}}(X)$

where **n** is the least value of N such that $P(F^N(X)) == 1$.

27 ••• Give the rules for a function that transposes a matrix represented as a list of lists. Your function can assume that each "row" of the matrix has the same number of elements without checking. You might, however, wish to construct a

separate function that checks the integrity of the matrix. Check your definition carefully and show that the types match up.

28 ••• Referring to the previous problem, if you understand matrix addition and multiplication, construct functions that carry out these operations.

29 •••• If you understand matrix inversion, construct a function that carries out this operation.

30 •• Show how to use enumeration to define the function `radix` without using the `%` and `/` functions.

4.10 Accumulator-Argument Definitions

Consider the problem of specifying a function that can reverse a list, for example:

```
reverse([1, 2, 3]) ==> [3, 2, 1]
```

The newcomer will typically try to approach this problem inductively by creating something like:

```
reverse( [ ] ) => [ ];                      // not recommended

reverse( [A | L] ) => append(reverse(L), [A]);
```

While this pair of rules does achieve its purpose, it is clumsier than necessary when it comes to execution by rewriting. This clumsiness translates into taking much longer in execution. This particular rule set requires a number of rewrites proportional to $n^2/2$ to reverse a list of length n, whereas it is possible to do it in rewrites proportional to only n. Here's an illustration for a list of length 4:

```
   reverse([1, 2, 3, 4])
=> append(reverse([2, 3, 4]), [1])
=> append(append(reverse([3, 4], [2]), [1])
=> append(append(append(reverse([4]), [3]), [2]), [1])
=> append(append(append(append(reverse([ ]), [4]), [3]), [2]), [1])
=> append(append(append(append([ ], [4]), [3]), [2]), [1])
=> append(append(append([4], [3]), [2]), [1])
=> append(append([4 | append([ ], [3])], [2]), [1])
=> append(append([4 | [3]], [2]), [1])
=> append(append([4, 3], [2]), [1])
=> append([4 | append([3], [2])], [1])
=> append([4, 3 | append([ ], [2])], [1])
=> append([4, 3 | [2]], [1])
=> append([4, 3, 2], [1])
=> [4 | append([3, 2], [1])]
=> [4, 3 | append([2], [1])]
=> [4, 3, 2 | append([ ], [1])]
=> [4, 3, 2 | [1] ]
```

```
=> [4, 3, 2, 1]
```

This clumsiness can be avoided by using the technique of an *accumulator*. An accumulator is an "extra" argument that serves to accumulate the result. In the case of `reverse`, what is accumulated is a list that ends up being the answer. For the reverse function, the reversal of the list is accomplished by moving the elements from one list to another. They are thus accumulated in an order that is the reverse of the order on the original list. We use a two-argument function `reverse`, then define a one-argument version in terms of it. In the first rule, when the original list is empty, we return the accumulated last:

```
reverse( [ ], R ) => R;
                     ↑      ↑
accumulator argument        the accumulator is returned
```

In the second rule, when the list is non-empty, we continue with the rest of the list and accumulate the first of the list on an already-started list:

```
reverse( [A | L], R ) => reverse( L, [A | R]);
                          ↑                ↑
 accumulator argument                      the accumulator accumulates
```

Let us verify that this results in far fewer rewriting steps for the previous list example:

```
   reverse( [1, 2, 3, 4], [ ])
=> reverse( [2, 3, 4], [1])
=> reverse( [3, 4], [2, 1])
=> reverse( [4], [3, 2, 1])
=> reverse( [ ], [4, 3, 2, 1])
=> [4, 3, 2, 1]
```

In general, using the non-accumulator definition will require a number of steps that is about one-half the square of the number of steps in the accumulator definition. Thus using an accumulator provides a significant saving in computation time. We shall see how to perform such an analysis in more detail in the chapter on Complexity.

4.11 Interface vs. auxiliary functions

In order to make a one-argument reverse function, we may define it in terms of the two-argument version presented in the previous section. The function in terms of this one by specifying an additional argument:

```
reverse(L) = reverse(L, [ ]);
```

We can give a description of what the two-argument `reverse` does: It appends the second list to the reverse of the first. This jibes with the rule above: appending `[ ]` to the reverse

of the first list is exactly the reverse of the first list, since appending `[ ]` to anything gives that thing.

To simply reverse a list, the one-argument `reverse` function is what we should provide to the user. Thus it is called an *interface* function. The two-argument reverse is called the *auxiliary* or "helper" function. This is not to say that a user would never have need for the auxiliary itself, but this would only happen if she wanted to do a combination of reversal and appending, which seems to be a less frequent need.

In many cases, we can build reversal into our function definitions rather than call upon reverse after the fact. For example, it was natural to produce the digits of the radix representation of a number least-significant digit first. If we wanted them most-significant first instead, we could either call reverse on the result, or we could just design the function to handle it directly using an accumulator argument. Here's how it would look for the radix representation. Note that we use an interface function for two purposes: to handle the special case of argument 0, and to call the auxiliary function with `[ ]` as the accumulator value.

```
digits(0, Radix) => [0];
digits(N, Radix) => digits1(N, Radix, [ ]);

digits1(0, Radix, Tail) => Tail;
digits1(N, Radix, Tail) =>
    digits1(N / Radix, Radix, [N % Radix | Tail]);
```

Function `digits` *gives the digits of the first argument represented in the radix of the second, most-significant digit first.*

Notice that the third argument of `digits1` is an accumulator. As we divide the original number successively by `Radix`, we are determining digits of higher powers of the radix, that get tacked on to the left-end of the list. When the number is finally decimated (reduced to 0), in the basis for digits1, the accumulated list is returned.

4.12 Tail Recursion

The type of recursion displayed by reverse using an accumulator, where there are no further operations to be performed on the rewritten result, is called **tail-recursion**. Tail-recursive rules have the desirable property that they reduce storage overhead resulting from nested function calls.

Below we show the distinction using `nrev` to denote the "naive" first attempt at constructing the `reverse` function vs. `rev2` to show the version with an accumulator argument. In `rev2`, there is nothing else to be done when the right-hand side returns its result. This is tail-recursion.

```
nrev([ ]) => [ ];
nrev([A | L]) => append(nrev(L), [A]);
                 ↑
                 due to this call, this rule is not tail-recursive

reverse( L ) = rev2(L, [ ]);
rev2( [ ], M ) => M;
rev2( [A | L], M ) => rev2( L, [A | M] );
                       ↑
                       this rule is tail-recursive
```

comparative forms of list reversal, non-tail-recursive vs. tail-recursive

While tail-recursive rules are desirable for efficiency, they can be less readable unless one is on the lookout for them. Therefore it is sometimes a good idea to have a non-tail-recursive reference version of a function on hand if a tail-recursive version is being used.

Consider trying to give a tail-recursive formulation for *factorial*:

```
factorial(0) => 1;

factorial(N) => N * factorial(N-1);
```

As with the naive `reverse` example, there is a tendency to build-up unfinished work, in this case multiplies, outside the principal expression being rewritten:

```
factorial(4) ==> 4*factorial(3) ==> 4*3*factorial(2) ==> …
```

The unresolved multiplications represent work to which we will have to return when we finally get to use the first rule. How would we express this function using tail-recursion? As it turns out, we cannot do so with only one argument: We need to add an *accumulator* argument to carry the accumulated product and the original argument value as it diminishes. This can be accomplished by using an auxiliary function of two arguments and defining the interface function in terms of it:

```
factorial(N) = factorial(N, 1);

factorial(0, M) => M;

factorial(N, M) => factorial(N-1, N*M);
```

Here we have “overloaded” the name `factorial` to use it for two distinct functions, one with one argument and the other with two arguments. Now consider evaluating factorial(4):

```
factorial(4)     ==>
factorial(4, 1)  ==> factorial(3, 4*1)  ==>
factorial(3, 4)  ==> factorial(2, 3*4)  ==>
factorial(2, 12) ==> factorial(1, 2*12) ==>
factorial(1, 24) ==> factorial(0, 1*24) ==>
```

```
factorial(0, 24) ==> 24
```

While the tail-recursive factorial has a cleaner evaluation sequence, its rules are more complicated due to the introduction of a second function. These rules don't appear to be as natural as the original ones.

Exercises

1 ••• Construct an alternate set of rules for `reduce` that biases the reduction to the left, i.e.

```
reduce(H, Base, [X0, X1, …, XN-1])  ==>
      H(…H(H(Base, X0), X1), …, XN-1)
```

This function is often differentiated from the original one by calling this one `foldl` (fold-left) and the other `foldr` (fold-right).

2 •• The function `mappend` combines `map` and `append` in the following way. Suppose `f` is a function of one argument that returns a list for arguments drawn from a list `L`. Then `mappend(f, L)` is the list of the values of `f(A)`, for `A` in `L`, appended together.

For example, if `f(1) ==> [10, 11]`, `f(2) ==> [12, 13]`, and `f(3) ==> [14, 15]`, then

```
mappend(f, [1, 2, 3]) ==> [10, 11, 12, 13, 14, 15]
```

This is in contrast with `map`:

```
map(f, [1, 2, 3]) ==> [[10, 11], [12, 13], [14, 15]]
```

Give rules that define `mappend`.

3 •• Give another set of rules for the function `length` that computes the length of a list. This time, use an accumulator argument so that the rules are tail-recursive.

4 ••• Using an accumulator argument, but not using explicit list reversal, give rules for a function that converts from binary to a natural number when the binary is represented as a list *most significant digit first.*

5 ••• Give rules for the function `qsort` (abbreviation for "Quicksort") that sorts a list by the following recursive method:

> If the list has one or no element, the result is just the list itself.
>
> If the list has more than one element, use the first element to split the list into two: one list of elements less than the first element, and a list of the

remaining elements. qsort each of those lists and append the results together so that the ordering is correct.

Once you have your function working, replace the use of `append` with an appropriate accumulator argument.

4.13 Using Lists to Implement Sets

It is common to use lists in the computer to represent sets. In order to represent a set, we disregard order of the elements. We must also ensure that there are no duplicate elements. The empty list [ ] is naturally used to represent the empty set. To add a new member to a set, we only need use the list constructor [|]. However, we must be sure that the member is not already present. The function `member` is such that `member(A, S)` ==> 1 if A is in the set and 0 otherwise.

```
member(_, [ ]) => 0;          // since the empty set can have no member

member(A, [A | S]) => 1;      // A is the first member in the list

member(A, [_ | S]) => member(A, S);

                              // A is not the first member, but could come later
```

To add a member known not to be in the set:

```
add_new_member(A, S) => [A | S];
```

To add a member in general, we use a guarded rule:

```
add_member(A, S) => member(A, S) ? S;   // already a member, no change

add_member(A, S) => add_new_member(A, S);
```

To form the union of two sets, we can use a process similar to `append`. However, we must take care not to duplicate any elements. We assume that there are no duplicates in either argument set.

```
union([ ], T) => T;

union([A | S], T) => add_member(A, union(S, T));
```

Power Set Example

The power set of a set S is the set of all of subsets of S. Suppose we wished to give rules for computing the power set (as a list) from a given set (list). For example,

```
subsets([a, b, c]) ==>
```

```
[[ ], [a], [a, b], [a, c], [a, b, c], [b], [b, c], [c]]
```

The type of *power*, by the way, is $A^* \rightarrow A^{**}$, since it takes a list of arbitrary things and returns a list of lists of those things.

Below we have worked through the reasoning of this problem. Thinking inductively …

Basis: `subsets( [ ] )` is `[ [ ] ]` since the empty set has only one subset: itself. This gives the following rule:

```
subsets( [ ] ) => [ [ ] ];
```

Induction: How can we get `subsets([A | L])` from `subsets(L)`?

For one thing, `subsets(L)` is *contained in* `subsets([A | L])`, i.e. `subsets([A | L])` will be something appended to `subsets(L)`:

```
subsets( [ A | L ] ) => append(subsets(L), ???);
```

What is missing? `subsets(L)` are those subsets of `[A | L]` that don't contain `A`. We need the ones that do contain `A`. But these are just like `subsets(L)` except that `A` has been added to each set.

So for `???` above we can use

```
add_to_each(A, subsets(L))
```

Now we have to define `add_to_each`. We can give rules for it alone, or we can recognize that `add_to_each` is just a "map" application:

```
add_to_each(_, [ ]) => [ ];

add_to_each(A, [E | S]) => [ [A | E] | add_to_each(A, S)];

subsets( [ ] ) => [ [ ] ];

subsets( [ A | L ] ) =>
      append( subsets(L), add_to_each(A, subsets(L)));
```

The first alternative eliminates the function `add_to_each` by using an anonymous function in conjunction with `map`. Noting that

```
add_to_each(A, L) == map( (S) => [A | S], L);
```

we can replace the `add_to_each` expression with the `map` expression in the second rule for subsets.

A second alternative is to replace the multiple uses of `subsets(L)` with an equational guard, giving us a new second rule:

```
subsets( [ A | L ] ) =>
      SoL = subsets(L),
      append( SoL, map( (X) => [A | X], SoL));
```

Finally, we can get rid of the call to `append` by using a version of `map` called `map_tail` that has an accumulator argument:

```
subsets( [ ] ) => [ [ ] ];

subsets( [ A | L ] ) =>
      SoL = subsets(L),
      map_tail( (X) => [A | X], SoL, SoL);

map_tail( F, [ ], Acc ) => Acc;

map_tail(F, [A | X], Acc) => map_tail(F, X, [ F(A) | Acc ]);
```

Exercises

1 • Trace through the series of rewrites for `union( [1, 2, 3, 4], [2, 4, 5, 6])`.

2 •• Give a set of rules for finding the *intersection* of two sets.

3 •• The *difference* of two sets, `difference(S, T)`, is defined to be the elements that are in S but that are not in T. Give a set of rules for the function `difference`.

4 •• Give a set of rules for testing two sets for equality, recalling that the elements need not be listed in the same order. One possibility is to use the `difference` function defined above. What other ways are there?

5 •• Define rules for a function `includes` so that `includes(S, T) ==> 1` if set S includes set T and `includes(S, T) ==> 0` otherwise. [Hint: Use the function `difference`.]

6 ••• Show that definitions for set operations can be simplified if we assume that the elements of each list always occur in a specific order and we can test that order between any two elements.

7 •••• Earlier we derived a way to compute the set of all pairs of elements of two sets represented as lists. Extend this idea to a function that computes the set of all n-tuples from a *list* of n sets. Calling this function tuples, we would have

```
tuples( [ [1, 2], [3, 4, 5], [6, 7] ] ) ==>

[[1, 3, 6], [1, 3, 7], [1, 4, 6], [1, 4, 7], [1, 5, 6],
 [1, 5, 7], [2, 3, 6], [2, 3, 7], [2, 4, 6], [2, 4, 7],
 [2, 5, 6], [2, 5, 7]]
```

Note that:

```
tuples( [ ] ) ==> [ [ ] ]
```

since there is exactly one tuple of no elements, the empty tuple. Also,

```
tuples( [ [ ] ] ) ==> [ ]
```

since there are no tuples that contain an element of the empty set.

4.14 Searching Trees

Consider the problem of determining whether an element satisfying a given property occurs in a tree. The property could entail specifying the exact identity or it could specify some characteristic of the element. Both of these cases are subsumed by specifying a predicate `P` that is satisfied exactly by the elements of interest.

This type of problem arises routinely in computer science. For example, if the tree is a directory structure, we might want to search for a specific file or sub-directory name, or for a file with specific ownership or permission properties, or with a last-write date before a certain date.

Let us assume that our trees are specified as lists, with the root as the first element and its major sub-trees as the remaining elements. For example, the following tree would be specified as the list

[1, [2, [4], [5], [6]], [3, [7, [9]], [8]]]

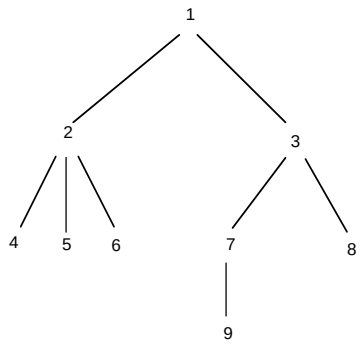


Figure 34: A tree for searching

The following simple recursive algorithm searches a tree for a node with property P:

To find a node with property P in a tree:

- If the root has property P, then return 1 (for success).
- Search the sub-trees of the root until one returns 1.
- If no sub-tree has returned success, then return 0 (for failure).

Let's cast these ideas as rex rules:

```
find_df(P, [Root | Subtrees]) => P(Root) ? 1;

find_df(P, [Root | Subtrees]) => find_in_subtrees(P, Subtrees);


find_in_subtrees(P, [ ]) => 0;

find_in_subtrees(P, [Tree | Trees]) =>
    find_df(P, Tree) ?
          1
        : find_in_subtrees(P, Trees);
```

Depth-first search of a tree for a node with property `P`.

Here `find_in_subtrees` iterates over the sub-trees performing a search on each until either there is success or until there are no more trees.

This is an example of **mutual recursion**. There are two functions, `find` and `find_in_subtrees` and each calls the other. Each function has a different set of responsibilities: `find` checks the root of its tree, while `find_in_subtrees` checks each sub-tree. The latter is essentially an iterative process. Both functions are tail-recursive.

Often mutual recursion can be replaced with use of one or more higher-order functions. For example, the following definition using the function `some` is equivalent, but more succinct:

```
find_df(P, [Root | Subtrees]) => P(Root) ? 1;

find_df(P, [Root | Subtrees]) => some((T)=>find_df(P, T), Subtrees);
```

The type of search exhibited above is called **depth-first search**. If there are several nodes in the tree satisfying the property, the first one is detected is the one that is encountered on a trajectory that plunges downward before it traverses laterally. The pattern of depth-first search in this case is shown below:

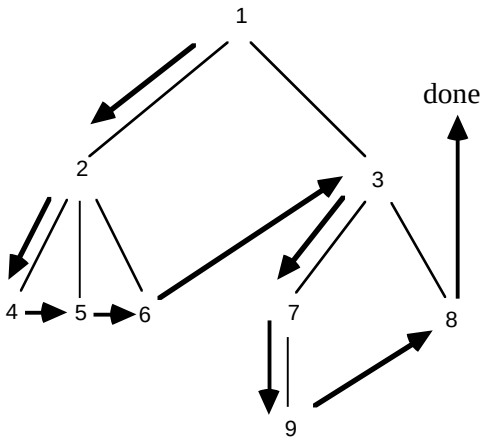


Figure 35: Depth-first search of a tree

A depth-first search establishes an ordering of the nodes, the order in which P is applied to test the node in the search. In this example, the ordering is [1, 2, 4, 5, 6, 3, 7, 9, 8]. This ordering is known as the **depth-first ordering**.

A complementary style of search is known as **breadth-first search**. Here the nodes are checked in order of increasing distance from the root. In order to accomplish this kind of search, we need to simulate a structure called a *queue*, which holds subtrees in the order the nodes are encountered until they can be revisited later. The algorithm is then as follows:

To find a node with property P in a tree, breadth-first:

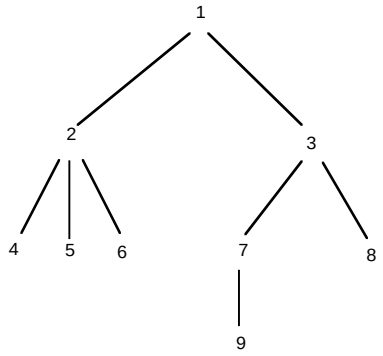
- Start with the tree as the only element in the queue.
- Repeat the following until success, or until queue is empty:
    - Consider the first tree in the queue. If the root of the tree satisfies P, then return success.
    - Add each of the sub-trees to the rear of the queue.
- (Queue is empty). Return failure.

Let’s make this algorithm more precise by presenting it in rex. Here we are using the same tree representation as before: The tree is represented as a list, with the first element of the list being the root and the rest of the list being the sub-trees.

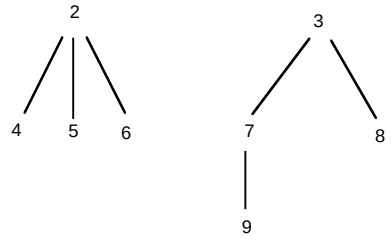
```
find_bf(P, Tree) => find_in_queue(P, [Tree]);

find_in_queue(P, [ ]) => 0;

find_in_queue (P, [[Root | Subtrees] | Trees]) =>
    P(Root) ?
      1
    : find_in_queue(P, append(Trees, Subtrees));
```

Breadth-first search of a tree for a node with property `P`.

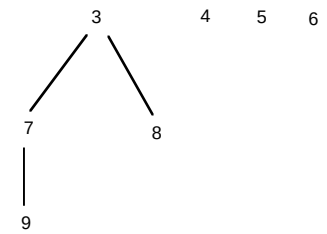
The following is a simulation of this algorithm, using the previous example tree. Suppose we are searching for a node equal to 7. Since the queue is a sequence of trees, we show that sequence at each stage. The initial queue is a sequence of one tree, the original tree:



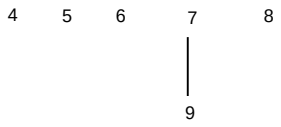
The root 1 is not equal to 7. The queue becomes the sequence of two trees:



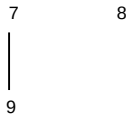
The first tree in the queue is the one with root 2, which is not equal to 7, so its sub-trees are appended to the end of the queue, resulting in:



The first tree in the queue is now the one with root 3, which is not equal to 7. So its sub-trees are added to the end of the queue, resulting in the queue:



The next three trees to be examined have roots not equal to 7 and the sub-trees are empty, so the queue becomes, in three steps:

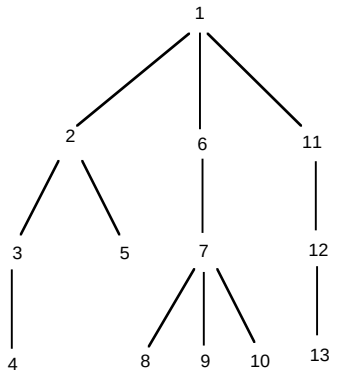


At this point, the root of the first sub-tree is equal to 7, so we stop with success.

As with depth-first search, breadth-first search also induces an ordering of the nodes, called the **breadth-first ordering**. This ordering is exactly what we would see if we read off the tree level by level, left-to-right. In the present example, this ordering is: [1, 2, 3, 4, 5, 6, 7, 8, 9].

Exercises

1 •• Consider the following tree. What are the depth-first and breadth-first numberings?



2 •• In the preceding tree, suppose `P(n)` is the property "n is a prime number greater than 5". What node would be found in a depth-first search? in a breadth-first search?

3 ••• Modify the depth-first search algorithm so that, rather than merely returning success, the algorithm returns a list representing the path from the root to the node found. For example, if we were searching the tree above for node 10, the list returned would be `[1, 6, 7, 10]`.

4 ••• Repeat the preceding problem for the breadth-first search algorithm.

5 ••• A *tree address* is a sequence of numbers indicating a path from the root of the tree to one of its nodes:

> The root of the tree has address `[ ]`.
>
> If the node occurs in the i^{th} subtree, the tree address of the node is i followed by the tree address of the node relative to the subtree.

We'll use the convention of numbering the subtrees starting at 0. For example, in the diagram above, the tree address of node 10 is

```
[1, 0, 2]
```

since the node occurs as root of subtree 2 of subtree 0 of subtree 1 of the overall tree.

Modify the depth-first search algorithm so that it returns the tree address of the node it finds, if any.

6 ••• Repeat the preceding problem for the breadth-first search algorithm.

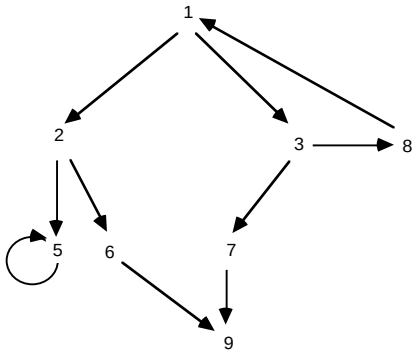
7 ••• Define in rex a function that will return the node in a tree given its tree address. It is possible that there is no node corresponding to a given address. Handle this possibility by returning a list of the node in the case the node is found, and the empty list in case of failure.

4.15 Searching Graphs

Searching a directed graph depth-first is similar to searching a tree, except that there are additional complications arising from the possibility the graph is not necessarily a tree. These are:

> A given node may be the target of more than one other node (this is sometimes called "fan-in"). We do not want to search from this node more than once. Thus we need some way of remembering that we've seen it before.
>
> A node may be in a cycle. Even though the node might be a target of just one node, unless we remember whether we've seen the node before, we could cycle forever and the search would not terminate.

The following graph is obtained by a slight modification of the previous tree. We see that fan-in occurs at nodes 5 and 9. A cycle occurs among nodes 1, 3, and 8, and also node 5 by itself.



We see that both non-tree phenomena can be handled by a common technique: refuse to search from a node from that we've already searched.

For the present, we will modify the search algorithms to keep a list of nodes that have been encountered. Loosely speaking, we check that list before searching the node a second time. There are other ways of doing this that do not involve scanning the list, but we will save them for an appropriate time later.

Another issue to be dealt with is that a general graph does not admit the representation we have been using for trees. Thus we have to use a different representation for general graphs. The one we will use now, for sake of concreteness, was introduced in *Information Structures*:

> A graph is a list. Each element is a list consisting of the name of a node followed by the targets of that node.

The list representation for the preceding graph would thus be:

```
[ [1, 2, 3],
  [2, 5, 6],
  [3, 7, 8],
  [5, 5],
  [6, 9],
  [7, 9],
  [8, 1],
  [9] ]
```

Despite this assumption, we shall try to cast the search algorithms to be relatively free of the assumption itself. We will do this by creating a function

```
get_targets(Node, Graph)
```

that, given a node and the graph, will return the list (possibly empty) of targets of the node. Only this function needs to know how the graph is represented. So if we change the representation, this is all we will need to change, not the search algorithm itself. The algorithm consists of the following rules:

Consider defining a depth-first search. The first rule is an interface function: `find_dfg(P, Node, Graph)` tries to find a node satisfying `P` in the graph, that is reachable from node `Node`.

```
find_dfg(P, Node, Graph) => find_dfg(P, [Node], Graph, [ ]);
```

The interface function calls the auxiliary function, with the set of "seen" nodes empty. If the set of nodes remaining to be searched is empty, then failure is reported.

```
find_dfg(P, [ ], Graph, Seen) => 0;
```

If there is at least one remaining node and the first node satisfies P, then success is reported.

```
find_dfg(P, [Node | Nodes], Graph, Seen) => P(Node) ? "1";
```

If the first node does not satisfy `P`, then we get the targets of the node. From those targets, we remove any that have been seen already. We add the remainder to the front of the list of nodes and continue the search, with the first node now noted as having been seen.

```
find_dfg(P, [Node | Nodes], Graph, Seen) =>
  Targets = get_targets(Node, Graph),
  New = difference(Targets, Seen),
  find_dfg(P, append(New, Nodes), Graph, [Node | Seen]);
```

For the particular graph representation described, function `get_targets` can be expressed using the built-in rex function `assoc`. Recall that this function searches a list of lists for a designated first component. If it finds one, in returns the list having that component. Otherwise it returns the empty list.

```
get_targets(Node, Graph) =>
  Found = assoc(Node, Graph),
  Found == [ ] ? [ ] : rest(Found);
```

A simple version of `difference` is as follows:

```
difference([ ], B) => [ ];

difference([A | As], B) =>
  member(A, B) ?
    difference(As, B)
  : [A | difference(As, B)];
```

For breadth-first search of a graph, we only need modify the `find` rule by changing the order of arguments to `append`:

```
find_bfg(P, [ ], Graph, Seen) => 0;

find_bfg(P, [Node | Nodes], Graph, Seen) => P(Node) ? "1";

find_bfg(P, [Node | Nodes], Graph, Seen) =>
```

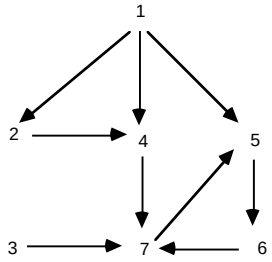
```
Targets = get_targets(Node, Graph),
New = difference(Targets, Seen),
find_bfg(P, append(Nodes, New), Graph, [Node | Seen]);
```

The effect is to put new targets behind the current queue of targets.

Breadth-first searching a graph finds the node *nearest* to the starting node, in terms of the number of arrows that need traversing. A variant of it is used in shortest-path or least-cost path problems. We shall discuss this further in a later section. The concepts of breadth-first numbering and depth-first numbering apply to graphs as well as trees, except that a specific starting node must be specified. Also, the numbering depends on the order in which the targets of each node are listed.

Exercises

1 •• Consider the following graph. What are valid depth-first and breadth-first numberings relative to node 1 as a starting node?



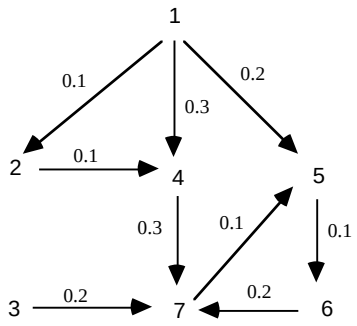
2 •• In the preceding graph, suppose that P(n) is the property "n is a prime number greater than 3". What node would be found in a breadth-first search?

3 ••• Modify the depth-first search algorithm so that, rather than merely returning success, the algorithm returns a list representing the path from the root to the node found. For example, if we were searching the graph above for node 6, the list returned might be `[1, 4, 7, 5, 6]`.

4 ••• Repeat the preceding problem for the breadth-first search algorithm.

5 ••• A third form of search is known as *iterative deepening*. As with breadth-first search, it finds nodes in order of increasing distance from the root, but it does not require storage for a queue. It is effectively a series of depth-first searches with increasing depth bounds. Construct a function for performing this form of search.

6 •••• Consider the following modification of breadth-first search: The arcs on a directed graph each have a positive numeric cost (representing, say, distance or travel time) associated with them. Devise an algorithm that, given a node, called the *source* node, computes the least-cost path between this node and all nodes. The cost of a path is defined to be the sum of the costs on the arcs in the path.



The graph in this case can be represented as follows:

```
[ [1, [0.1, 2], [0.2, 5], [0.3, 4]],
  [2, [0.1, 4]],
  [3, [0.2, 7]],
  [4, [0.3, 7]],
  [5, [0.1, 6]],
  [6, [0.2, 7]],
  [7, [0.1, 5]] ]
```

The result of the algorithm would be a list of [Cost, Node] pairs. For source node 1 this would be:

```
[[0, 1], [0.1, 2], [0.2, 4], [0.2, 5], [0.3, 6], [0.5, 7],
[Infinity, 3]]
```

4.16 Argument Evaluation Disciplines

It is often the case that there is more than one sub-expression to which rules can be applied to a term. For example, consider a rule set for `add`:

```
add(0, M) => M;
add(N+1, M) => add(N, M)+1;
```

Suppose we want to evaluate the term

```
add(0, add(0, 5))
```

Here we could apply the rule for `add(0, M)` to the outer term to get

```
add(0, 5)
```

or to the inner term `add(0, 5)`, to get the same thing. However, we will not *always* rewrite to the same thing immediately. To see this, consider a term of the form

```
add(N+1, add(K+1, M))
```

Applying a rule to the outer term gives us

```
add(N, add(K+1, M)) + 1
```

while applying to the inner term gives

```
add(N+1, add(K, M) + 1)
```

These two are obviously different, although by another rule application, we could convert both of them to

```
add(N, add(K, M) + 1) +1
```

Applicative Order

Most programming languages adopt a specific discipline about where a rule will be applied when there is a choice. The most common discipline is known as

> **applicative-order argument evaluation**:
>
> Give priority to applying a rule to an *argument* of a term before applying a rule to the entire term.
>
> Example: In `f(g(0), h(1))`, apply a rule for `g` or `h` before applying any rule for `f`.

Even this is not without ambiguity, however, since there could be several arguments to which rules apply. By **leftmost applicative-order,** we give priority to the leftmost argument term first, and similarly for *rightmost applicative-order*.

Examples

In `add(N+1, add(K+1, M))`, we have an argument `add(K+1, M)` to which a rule is applicable. Moreover, this is the leftmost such argument, so the rewritten term under leftmost applicative order is `add(N+1, add(K, M)+1)`.

In `add(N+1, add(K+1, add(M+1, 2)))`, under leftmost applicative order, a rule is applicable to the second argument, `add(K+1, add(M+1, 2))`. However, this argument also has an argument to which a rule is applicable, so we must give priority to that argument rather than the outer argument. Thus, the rewritten term would be `add(N+1, add(K+1, add(M, 2)+1))`.

Normal Order

Another evaluation-order discipline with important uses is known as

> **normal order argument evaluation**:
>
> Give priority of applying a rule to the *entire* term over applying a rule to an argument of the term.
>
> Example: In `f(g(0), h(1))`, apply a rule for *f* before applying any rule for *g* or *h*.

Examples

Under normal order we would have the following series of rewrites:

```
add(N+1, add(K+1, add(M+1, 2))) ==>
add(N, add(K+1, add(M+1, 2)))+1 ==>
add(N, add(K, add(M+1, 2))+1)+1 ==>
add(N, add(K, add(M, 2)+1)+1)+1
```

Contrast this with applicative order, where the rewrite series would be:

```
add(N+1, add(K+1, add(M+1, 2))) ==>
add(N+1, add(K+1, add(M, 2)+1)) ==>
add(N+1, add(K, add(M, 2)+1)+1) ==>
add(N, add(K, add(M, 2)+1)+1)+1
```

The end results are the same, but the intermediate details differ.

Even though applicative order is the most common, normal order has the advantage of terminating in some cases where applicative order does not. As an example, consider the following rules:

```
if(1, A, B) => A;
if(0, A, B) => B;

foo(N) => foo(N+1);
```

Consider the term

```
if(0, foo(0), 1)
```

Applicative order would require that `foo(0)` be evaluated before using the definition of `if`. However, `foo(0)` will never converge. Therefore applicative order will never use the definition of *if* and the entire computation *diverges*. On the other hand, with normal order, the second rule for if will be used immediately and there will be no call for the evaluation of `foo(0)`.

It can be shown that normal order is strictly more general, in the sense that there will never be a case where applicative order gives an answer but normal order fails to give one. Unfortunately, applicative order is the norm in most programming languages, not normal order. One might remember this by the following quip:

Normal order isn't.

The reason that normal order is not "normal" has to do with normal order being more complicated to implement, not that it is undesirable for mathematical reasons.

Normal Order in rex and Delayed Evaluation

As with most languages, the rex evaluator uses applicative order for all functions, except for a few "special forms" such as the conditional form `__ ? __ : __` , logical conjunction `&&`, and logical disjunction `||`, that evaluate arguments left to right as they need them.

It is possible to achieve a custom normal order effect through what we will call the defer operator. Any expression, including an argument to a function, can be "wrapped" as the argument to an operator `$` (read "defer"). The expression is not evaluated until it is absolutely necessary. Thus, if we have an actual argument wrapped in $:

```
h($f(X, Y), Z)
```

this argument will effectively be treated as if a normal-order argument, while others will be treated as applicative order. Only when, if ever, it becomes necessary for `h` to know the value of `f(X, Y)` will the latter be evaluated. For example, in a conditional expression

```
p(X) ? $f(X, Y) : g(Y, Z)
```

even if `p(X)` evaluates to 1, we do not need to know the value of `f(X, Y)`. The value of this expression is just `$f(X, Y)`. If, on the other hand, we used `f(X, Y)` in a numeric expression, such as

```
Z + $f(X, Y)
```

it becomes necessary to know what the value of `$f(X, Y)` is. At this point the expression would be evaluated.

One of the key uses of `$` in rex will be explained in the section *Infinite Lists*.

Using Function Arguments to Achieve Delay

A traditional device for achieving the effect of delaying the evaluation of an argument expression (i.e. the *defer* operator, as discussed with normal order evaluation) is to embed the expression in question into the body of an additional function with no arguments. Then, when we want to evaluate this expression, we apply the function (to no arguments). For example, suppose that the expression we want to delay is

```
X + g(X, Y)
```

To pass this expression unevaluated, we actually pass the 0-argument function

```
() => X + g(X, Y)
```

Suppose that this function is bound to a variable `D`. Then when we want the evaluation of the original expression to take place, we would compute

```
D()
```

(`D` applied to no arguments). This scheme differs slightly from the *defer* scheme in rex. In the scheme being discussed, the program must know to expect the 0-argument function and arrange for its application (to no arguments). Thus a function cannot be written that will take *either* a delayed argument or an ordinary argument, unless some sort of *tag* is also passed along to indicate which.

4.17 Infinite Lists (Advanced)

This topic describes a programming paradigm that is available in very few languages (rex is one, of course!). It can be "engineered" in others, but sometimes with great difficulty. However, due to the substantial power that this approach provides, it will likely be in many high-level languages at some point in the future (how distant we hesitate to speculate).

The rewriting approach provides an ideal way to describe and implement an unusual feature that is available in some languages: the ability to manipulate lists as if they were infinite. This requires delaying the computation of the tail of the list, as in `[ A | $ L ]`, until the tail is needed. Otherwise an attempt would be made to evaluate the tail, resulting in divergence.

The List of All Natural Numbers

The simplest non-trivial example of an infinite list is the list of all natural numbers, conceptually shown as

```
[0, 1, 2, 3, …]
```

We want *from* to be a function that, with argument `N`, will generate the infinite list of numbers from `N` on. The list above is the special case `from(0)`. The definition of `from` must use a normal-order list constructor. As discussed earlier, this can be achieved by the delay wrapper `$`, as in:

```
from(N) => [ N | $ from(N+1)];
```

The idea here is that the recursive call to `from(N+1)` is not evaluated until needed. So if we are showing the result of `from(0)`, we would do these evaluations as it comes time to print each successive element. Let us check this by giving a few rewrites of `from(0)`:

```
from(0) ==>
[ 0 | $ from(1) ] ==>
[ 0 | [ 1 | $ from(2) ] ] ==>
[ 0 | [ 1 | [ 2 | $ from(3) ] ] ] ==>
[ 0 | [ 1 | [ 2 | [ 3 | $ from(4) ] ] ] ]
```

which is the same as

```
[0, 1, 2, 3 | $ from(4) ]
```

When applying a rule for a function that has such a list as an argument, the usual rules apply: a formal argument `[A | L]` matches the actual argument so that `A` is the first element of the infinite list and `L` is the rest of the infinite list. For example, define functions *first* and *rest* by

```
first( [A | L] ) => A;

rest( [A | L] ) => L;
```

Then `rest` would force the evaluation of the delayed expression as necessary:

```
rest(from(0)) ==> rest( [0 | $ from(1)] ) ==> $ from(1)
```

If the result of *rest* were used, e.g. in evaluating

```
5 + first(rest(from(0)))
```

then `$from(1)` would be further expanded to get `[1 | $ from(2)]` and `first` would extract the 1, rewriting to 5 +1, then to 6.

Using this idea, we can construct functions that have infinite lists as arguments and results. For example, the function partial_sums produces a list of the sum of the first, first two, first three, and so on, elements of its argument:

```
partial_sums( [1, 3, 5, 7, …] ) ==>
                                    [1, 4, 9, 16, …]
```

The rules are, using an auxiliary function `partial_sums2`:

```
partial_sums(X) => partial_sums2(0, X);   // 0 is initial accumulator

partial_sums2(Acc, [ ]) => Acc;

partial_sums2(Acc, [A | X]) => [(Acc + A) | $ partial_sums2(Acc+A, X)];
```

Unzipping an Infinite List

The following function "unzips" a finite or infinite list into two lists.

```
unzip(X) = [evens(X), evens(rest(X))];

evens([ ]) => [ ];
evens([A]) => [A];
evens([A, _ | X]) => [A |$ evens(X)];
```

Unzipping a list

Pipe Composition

A very attractive aspect of functions on infinite lists is that they can be composed as with pipe composition discussed earlier. An example of pipe composition for infinite lists occurs in the next example, and is previewed here.

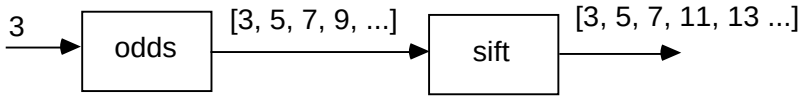


Figure 36: Piping an infinite stream through a function

This type of composition gives infinite lists value for certain computing applications, such as digital signal processing, where the application is typically structured as a set of interconnected stream-processing functions: integrators, filters, scalars, and the like.

Prime Number Sieve

The function `primes` below produces the infinite list of prime numbers beginning with 3. It does this using the technique of "sieving". Consider the infinite list of odd numbers:

```
3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, …
```

From this list, drop all those, other than the first (i.e. 3), that are multiples of the first:

```
3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, …
```

Now do the same for the *next* number that is left (5), i.e. drop all multiples of it:

```
3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, …
```

Continue this way, dropping multiples of 7, then 11, … The numbers that survive the drops are the primes. At every major step, the first number survives, so this insures that every prime will eventually be produced.

The program is:

```
primes() = sift(odds(3));

odds(N) = [N | $ odds(N+2)];

sift([A | X]) => [A | $ drop((X) => divides(A, X), sift(X))];
```

Function `primes` *generates the infinite list of primes*.

To gain maximum utility from this paradigm, it is helpful to be able to compose programs with loops, as will be discussed in the next section.

Functional Programs with Loops

Another technique that can be used to generate infinite lists is to have "loops" in the defining equations. For example, the following equation:

```
Ones = [ 1 |$ Ones ];
```

defines `Ones` to be the infinite list [1, 1, 1, 1, …]. The figure below shows how this works, by piping the output back into the `|` (followed-by) function.

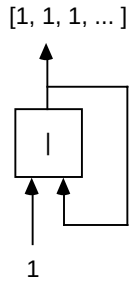


Figure 37: A simple functional program with a loop.
Here | represents the "followed-by" function used to construct lists.

Example - Another way to get the partial sums of an infinite sequence X is to use:

```
Psums = map(+, X, [0 | $ Psums]);
```

Here + is applied to two sequences, so this is the `map` form of +, rather than the simple arithmetic form. The definition of `Psums` has a "loop" in the sense that the definition

itself uses the quantity `Psums`. The two programs with loops for `Ones` and `Psums` can be shown as follows:

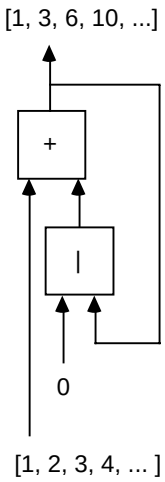


Figure 38 A functional program with a loop showing result for an example input.

Here + represents `map(+, . , .)`

Here is an example of how this works in rex:

```
rex > X = from(1);
1

rex > X;
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, …

rex > Psums = map(+, X, [0 | $ Psums]);
1

rex > Psums;
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, …
```

Fibonacci sequence using a recursive group

The examples above both defined fixed infinite sequences using loops. If we want to define *functions* using loops, we need something like an equational guard, yet slightly different. Consider the following attempt to define the function fib that generates the Fibonacci sequence:

```
fib() = Result = [1, 1 | $ map(+, Result, rest(Result)) ],
        Result;
```

This definition first defines the quantity represented by variable `Result` using an equational guard, then gives that value as the result of the function. Syntactically this definition is well-formed. However, the value of `Result` used on the right-hand side of

the equation is *not* the same as the one on the left; the value is, by definition, the value of `Result` in the ambient environment (it may or may not be defined). What we want is an environment where both uses of `Result` mean the same thing. We had this in the global environment in earlier examples. But how do we get it inside the function `fib`? The answer is that we need a special construct called a *recursive group* that creates a recursive environment. In rex this is shown by giving a series of equations inside braces {…}. Each variable defined in that environment has the same meaning on the left- and right-hand sides of the equations. The last thing inside the braces is an expression, the value of which is the value of the group. The correct version of `fib()` is as follows:

```
fib() = { Result = [1, 1 | $ map(+, Result, rest(Result)) ];
          Result};
```

Here the first equation defines the variable `Result` to be a list starting with [1, 1, …]. The rest of the list is the pairwise sum of the list itself with the rest of the `Result` , [1, …]. Thus the first element in this sum is 2, the next element is therefore 1+2 ==> 3, the next 2+3 ==> 5, and so on. A rex dialog show this:

```
rex > fib();
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …
```

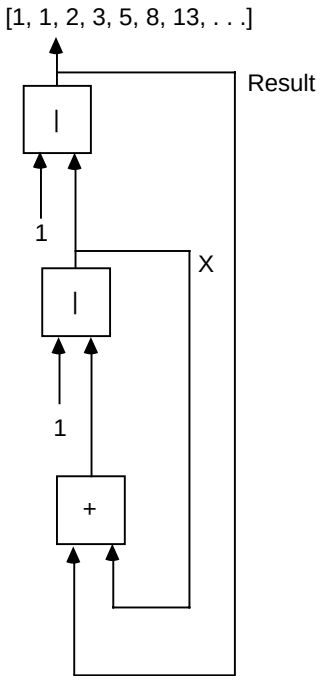


Figure 39: A functional program generating the Fibonacci sequence.

The use of `rest` in the program was eliminated by using the fact that `rest([A | X]) == X`

A definition that conforms directly to the diagram, using an additional variable `X`, is:

```
fib() = { Result = [1 | $ X];
          X = [1 | $ map(+,Result, X)];
          Result};
```

Simulating Differential Equations

Differential equations are equations that are to be solved for functions, rather than numbers, as unknowns. These equations are constructed using differential, as well as algebraic, operators. A typical type of differential equation involves real-valued functions of one variable. Often that variable is identified as a time parameter. We can simulate such equations by using a discrete approximation to time. In this case, a function of time can be represented by the sequence of values sampled at discrete time instants. With this in mind, it is possible to use our infinite lists as these sequences, i.e. to represent real-valued functions of time.

As an example, the derivative operator can be simulated by taking differences of adjacent argument values. The definition is:

```
deriv([A, B | X]) = [(B - A) | $ deriv([B | X]) ];
```

As it turns out, however, we do not use this operator directly in the solution method to be presented.

The usual algebraic operators +, -, *, etc. have to be mapped as pairwise operators on infinite lists. Thus to add the values of two "functions" `F` and `G` represented as sequences, we would use `map(+, F, G)`.

First-Order Equation

Suppose that we wish to solve a first-order (meaning that the highest-order derivative is 1) linear homogenous (meaning that the *rhs* is 0) equation:

$$\frac{dX}{dt} + a*X(t) = 0$$

subject to an initial value X(0) = X0. A solution entails finding a function X that satisfies this equation. We will represent the function X by a sequence. The sequence corresponds to the values of the true function X at points 0, 0 + dt, 0 + 2dt, …, treating dt as if it were an actual interval. This interval is known as the “step size”. It will become implicit in our solution method. Solving the equation for dX:

$$dX = -a*X(t)*dt$$

But also

$$dX = X(t+dt) - X(t)$$

Combining these two and solving for X(t+dt):

$$X(t+dt) = X(t) - a*X(t)*dt$$

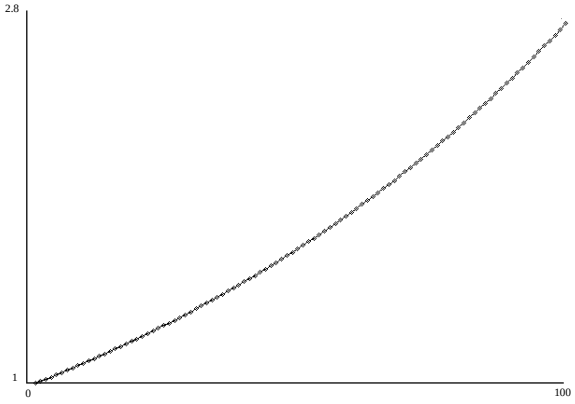
Taking dt to be 1, we have

$$X(t+1) = X(t) - a*X(t)$$

Now we have the approximation X(t+1) expressed in terms of X(t). Combining that with the known initial value `X0`, we can write in rex:

```
X = [X0 |$ map(-, X, scale(a, X))];
```

(Note that this equation has a "loop".) As before, we are using the map version of operator - that works on two sequences pairwise. For a given values of `X0` and `a`, the sequence `X` is thus determined. For example, the following figure shows the points in sequence `X` when `X0`== 1 and `a` == -0.01.



Graph of the solution to a first-order differential equation.

Analytically, we know that the solution to the equation is $X(t) = e^{0.01t}$, which jibes with the numerical solution obtained; at t = 100, we have X(100) == 2.70481, which is approximately equal to e == 2.71828.

The solution method represented above is effectively what is called **Euler's method**. It is not the most accurate method, but it is believed that the same solution technique using infinite lists can also be applied to more refined methods, such as the Runge-Kutta method.

Second-Order Equation

To show that the method presented above is general, we apply it to a second-order equation, of general form:

$$\frac{d^2X}{dt^2} + a*\frac{dX}{dt} + b*X(t) = 0$$

Where initial values are given for both X and $\frac{dX}{dt}$. It is common to introduce a second variable Y to represent $\frac{dX}{dt}$, transforming the original single equation to a system of equations:

$$\frac{dY}{dt} + a*Y(t) + b*X(t) = 0$$

$$Y(t) = \frac{dX}{dt}$$

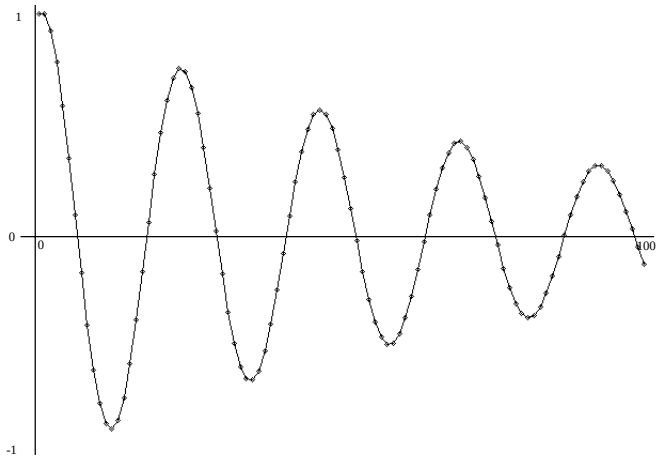
As before, we treat dt as if it were a discrete interval. As before, we solve for dX and dY, and equate these to X(t+1) - X(t) and Y(t+1) - Y(t) respectively. This gives:

```
X(t+1) = X(t) + Y(t)*dt
Y(t+1) = (1 - a)*Y(t) - b*X(t)
```

Translating into rex, using infinite lists:

```
X = [X0 | $ map(+, X, Y)];
Y = [Y0 | $ map(-, scale((1-a), Y), scale(b, X))];
```

Here when a scalar is multiplied by a sequence, the result is that of multiplying each element of the sequence by the scalar. The diagram below shows the first 100 values of `X` when `a` == 0.1, `b` == 0.075, `X0` == 1, and `Y0` == 0.



Graph of the solution to a second-order differential equation.

Exercises

1 • Trace the first few rewrites of `partial_sums(from(0))` to verify that the partial sums of the integers are [0, 1, 3, 6, 10, 15, …]

2 • Give rewrite rules for a function `odds` such that

```
odds(1) ==> [1, 3, 5, 7, 9, …]
```

3 •• Certain sets of rules on lists also make sense on infinite lists. An example is `map`, as introduced earlier. For example,

```
map(square, odds(1)) == [1, 9, 25, 49, 81, …]
```

Review the previous examples we have presented to determine which do and which don't make sense for infinite lists. Indicate where $ needs to be introduced to make the definitions effective.

4 •• Give rules for a function that takes a function, say *f*, as an argument, and produces the infinite sequence of values

```
[f(0), f(1), f(2), f(3), …]
```

5 ••• Give rules for a function that take a function, say *f*, and an argument to f, say x, as arguments, and produces the sequence of values

```
[f0(x), f1(x), f2(x), f3(x), …]
```

where $f^i(x)$ means f(f(…f(x)…)) (i times).

6 •• Suppose we use infinite lists to represent the coefficients of Taylor's series. That is, $a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$ is represented by the infinite list $[a_0, a_1, a_2, a_3, \ldots]$. Present rex functions that add two series and that multiply them by a constant.

7 ••• Continuing the above representation of series, construct a rex function that multiplies two series. The corresponding operation on infinite lists is called the *convolution* of the lists.

8 •••• Continuing the above thread, construct a rex function that derives the coefficients of

$$\frac{1}{1 - s}$$

where *s* is a series.

9 •••• Derive rex functions that generate the series of coefficients for your favorite analytic functions (*exp*, *sin*, *cos*, *sinh*, etc.).

10 ••• ["Hamming's problem"] Develop a function that generates, in order, the infinite list of numbers of the form $2^i 3^j 5^k$, where i, j, and k are natural numbers, i.e.

```
[2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, …]
```

11 •••• Referring to the earlier problem regarding transposing a matrix, construct a function that will transpose an *infinite* matrix, represented as an infinite list of infinite lists. For example, if the function's argument is:

```
[[0, 1, 3, 6, 10, …], [2, 4, 7, 11, …], [5, 8, 12, …],
 [9, 13, …], [14, …], …]
```

the transpose is

```
[[0, 2, 5, 9, 14, …], [1, 4, 8, 13, …], [3, 7, 12, …],
 [6, 11, …], [10, …], …]
```

12 ••• Referring to the previous problem, construct a function that will linearize an *infinite* matrix by "zig-zagging" through it. For example, zig-zagging through the first matrix above would give us:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, …]
```

13 ••• Construct a function that is the inverse of the zig-zag function in the previous problem.

14 ••• Define a version of 'pairs' that will work in the case that either or both argument lists are infinite, e.g.

```
pairs(from(0), from(0)) ==>

    [[0, 0], [1, 0], [0, 1], [2, 0], [0, 2], [1, 1], [0, 3], …
    ]
```

Thus, such a definition demonstrates the mathematical result that the Cartesian product of a countable set is countable.

15 ••• Derive solutions for cases in which the right-hand side of the above equations are replaced by “forcing functions” of t, which in turn are represented as sequences.

16 ••• Derive solutions for cases in which the coefficients of the equation are functions of t rather than constants..

17 •••• Explore the adaptation of more refined solution methods, such as *Runge-Kutta* (if you know this method) to the above approach.

4.18 Perspective: Drawbacks of Functional Programming

Functional programming is important for a number of reasons:

- It is one of the fundamental models of computability.
- It provides succinct and elegant means of manipulating potentially very large information structures without deleterious side-effects on data used by some models.
- Consequently, it is a useful model for parallel computation, which can be prone to anomalous behavior if side-effects are not managed carefully.

Functional programming can also fit well with other models, such as object-oriented and logic programming, as will be seen. Despite these desirable traits, we hesitate to recommend it as the only model one consider for software development. Instead we would prefer to see its use where it fits best.

An example of where functional seems less than ideal is computations that need to repeatedly re-assign to large arrays destructively. Here "need" is used subjectively; there is no widely-accepted theoretical definition of what it means to *require* destructive modification. Intuitively however, the following sort of computation is a canonical example: Consider the problem of maintaining a *histogram* of a set of integer data. In other words, we have an incoming stream of integers in some range, say 0 to N-1, in no particular order. We want to know: for each integer in the range, how many times does it appear in the stream. The natural way to solve this problem is to use linear addressing: for each data item in the stream, use the item to index an array of counts, adding 1 each time that integer is encountered. This method is straightforward to implement using destructive assignment to the array elements. However, a functional computation on arrays would create a new array for every element in the stream, which will obviously be costly in comparison to using destructive modification. Some functional programming languages are able to get around this problem by using clever compilation techniques that only *apparently* create a new array at each step but that actually re-use the same array at each step. However, it does not appear that such techniques generalize to all possible problems.

A place where functional programming seems to yield to object-oriented programming techniques is in programming with structures that seem to inherently require modification because there is only one of them. An example is in graphical user interface programming. Here there is only one of each widget showing on the screen, which contains the state of that widget. It does not make sense to speak of creating a new widget each time a modification of the state is made.

4.19 Chapter Review

Define the following concepts or terms:

accumulator argument
append
applicative order
auxiliary function
beta reduction
breadth-first search
copy rule
delayed evaluation
depth-first search
equational guard
Euclid's algorithm
Euler's method
Horner's rule
insertion sorting
interface function
guarded rule
inductive definition
radix representation
merge sorting
mutual recursion
normal order
radix principle
recursion
selection sorting
sieve
tail recursion

4.20 Further Reading

L.C. Paulson, *ML for the working programmer*, Cambridge University Press, Cambridge, MA, 1991.

Simon Thompson, *Haskell - The craft of functional programming,* Addison-Wesley, Reading, MA,1999.

5. Implementing Information Structures

5.1 Introduction

This chapter discusses some of the key principles for constructing information structures, such as lists and trees, and discusses primitive implementation in Java as an example. Such structures provide a foundation for the understanding of algorithm design considerations that play a central role in computer science, some of which will be presented in later chapters.

We have already discussed arrays extensively. Arrays are one of the key components of structural computing. The other components are *records* (as they are called in Pascal) or *structs* (as they are called in C). In Java, the class concept is used as an extension of this notion, in the sense that a class provides methods for accessing the data as well as a way to represent the data internally. Classes, coupled with arrays, are the key building blocks for constructing a wide variety of "data structures". Further discussion of the object concept and its uses appears in the following chapter.

5.2 Linked Lists

Linked lists are one of the key structuring devices in computer software. Generally speaking, lists are used to build sequences of data items *incrementally*, especially when we have no advanced notion of how large the sequence will ultimately be. Instead of having to estimate an appropriate initial size, and possible make wholesale adjustments during population of an array, lists allocate item by item, using only as much storage as is needed to hold the items plus a per-item overhead. We describe how linked lists provide a way of implementing the list abstraction found in rex, as well as implementing other list abstractions.

As an example of where linked lists are useful, consider implementing a text editor application. Suppose that the text is organized as a series of paragraphs. The editor provides a way of cutting a paragraph from one part of the document and pasting it in another. In order to make this operation fast, we would avoid storing the paragraphs as a linear array, since this cutting and pasting would entail shifting the elements of the array each time we perform an operation. Instead we would have each paragraph remember the paragraph after it by a *reference* to that paragraph. This kind of use of references is seen whenever we read a newspaper. The blocks of text for an article (which don't coincide with paragraphs necessarily) are scattered on different pages. At the end of a block is a "reference" message "continued on page ...". In a computer, references are not simply pieces of text. Instead they are implemented as memory references or pointers to the next block. Thus the process of finding the target of a reference is very fast, as it can exploit the linear addressing principle.

> The process of going from a reference to its target is called *dereferencing*.

We could effect the cut and paste operation simply by changing references rather than doing a physical cut and paste. Note that some newspapers also provide reverse references "continued from page ...". These would be called "doubly-linked lists" and are mentioned further in the exercises.

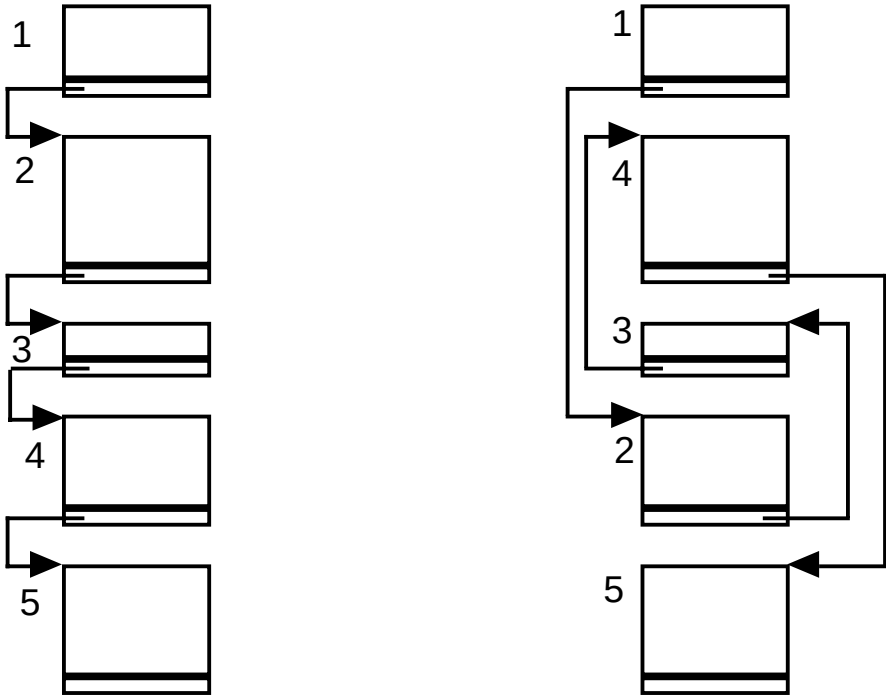


Figure 40: Exchanging paragraphs 2 and 4 by changing references

The key idea of linked lists is to provide a flexible way of connecting units of information together even though they reside in non-contiguous units in computer memory. This is accomplished by constructing a list out of objects, often called **cells**, where each cell contains both a data item and a reference to the next cell in the list. In the final cell of the list, the reference is a special *null reference* that indicates there are no further cells. **The null reference cannot be dereferenced.** In Java, attempting to dereference a null reference will result in a run-time error. In some languages, attempting to do so may produce an unpredictable result. Thus one should always make sure, one way or another, that a reference is not null before dereferencing it.

> The same test for a null reference, which tells whether we are at the end of the list, is also the one that tells whether we have the ability to dereference the reference, that is, whether there is any target.

The figure below shows how a linked-list cell is viewed. The Java code for declaring this type of cell might be:

```
class Cell
{
Item data;
Cell next;
}
```

Here `Item` refers to the type of the data item itself. This can be either a basic type or a defined type. The field `next` is the reference to the next cell. The reason it is not necessary to make any special mention that `next` is a reference is that it is implicit: *In Java, all variables representing objects are implicitly references.* Because the type of object being defined is named *cell* and *cell* is mentioned in the definition, this can also be viewed as **recursive type definition**:

```
Cell_reference = Item x Cell_reference | null
```

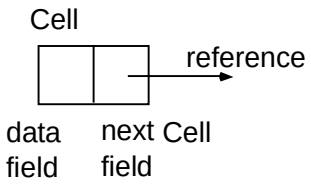


Figure 41: List cell structure

The figure below shows how we depict the case where the value of next is the null reference. This form is used because the next field doesn't point to anything.

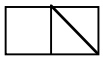


Figure 42: Representing the case of the last element, i.e. the next reference does not point to anything

The following is an example of a linked list with data elements a, b, c, d.

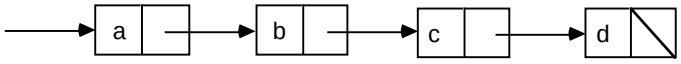


Figure 43: A linked list of four elements

Of course, such structures have already been mentioned in Chapter 2. It is also possible for the elements in the list to be *references* to the actual data elements. This is especially useful in the case that the elements are of non-uniform size, as we might have with a list of strings.

We distinguish between two varieties of lists, but many variations in between are also possible:

Closed lists:
A closed list is a linked list, the cells of which are not normally shared by other lists.

Open lists:
An open list is a linked list in which the cells are shareable by other open lists.

5.3 Open Lists

In an open list, sharing is encouraged, to economize on storage space and to reduce overhead from copying. However, open lists complicate matters of reclaiming unused storage, since we cannot simply delete a cell just because we are done with it in one: such a cell might also be one or more other lists. Java takes care of such reclamation automatically by computing whether each cell is accessible or not when storage becomes in short supply. Cells that are no longer accessible are recycled (deallocated and made available for other uses).

Part of the reason we emphasize open lists here is that they correspond in a natural way to the implementation of lists in rex and related languages. *In simplest terms, a list can be viewed as a reference to a cell.* The empty list is identified with the null reference. For this reason, we could simply rename the `Cell` class previously presented to be a list class. The distinction between list and cell in this simple implementation is purely one of viewpoint. In more complex closed-list implementations to be described later, it will be important to distinguish between lists and cells.

```
class List
{
Item First;       // data in the first cell
List Rest;        // reference to next cell and rest of the list
}
```

To make this more convincing, we show how to implement the rex functions `cons`, `first`, and `rest`.

Function cons constructs a new list from an existing list and a new first element. Were it to be defined anew in rex, the definition would be:

```
cons(E, L) = [E | L];
```

With the definition of List used previously, this function would be defined by including it as a "static method" within the class definition, since Java does not have functions as

such. A static method is one that applies to all objects in the class in general, rather than a particular object.

```
class List
{
Item First;
List Rest;

// return a new list (reference to cell) created from an existing
// list (referenced by Rest) and a data item

static List cons(Item First, List Rest)
  {
  List result = new List;
  result.First = First;
  result.Rest = Rest;
  return result;
  }

}
```

A more elegant way to accomplish this same effect is to introduce a constructor for a List that takes the First and Rest values as arguments. A constructor is called in the context of a new operator, which creates a new List. Adding the constructor, we could rewrite cons:

```
class List
{
Item First;
List Rest;

// construct a List from First and Rest

List(Item First, List Rest)
  {
  this.First = First;
  this.Rest = Rest;
  }


// return a new list (reference to a cell) created from an item
// First and an existing list Rest

static List cons(Item First, List Rest)
  {
  return new List(First, Rest);
  }
}
```

The functions `first` and `rest` would be defined in rex as follows:

```
first( [E | L] ) = E;    // return the first element in a list

rest( [E | L] ) = L;     // return the list of all but the first
```

We now add corresponding functions to the Java implementation:

```
class List
{
Item First;
List Rest;

// construct a List from First and Rest

List(Item First, List Rest)
  {
  this.First = First;
  this.Rest = Rest;
  }


// return a new list (reference to a cell) created from an item
// First and an existing list Rest

static List cons(Item First, List Rest)
  {
  return new List(First, Rest);
  }


// return the first element of a non-empty list

static Item first(List L)
  {
  return L.First;
  }


// return all but the first element of a non-empty list

static List rest(List L)
  {
  return L.Rest;
  }


// return indication of whether list is empty

static boolean isEmpty(List L)
  {
  return L == null;
  }

static boolean nonEmpty(List L)
  {
  return L != null;
  }
}
```

We took the liberty of also adding the functions `isEmpty` and `nonEmpty` to the set of functions being developed, as they will be useful in the following discussion.

Now let's use these definitions by presenting the implementation of some typical rex functions. Consider the definition of function `length` that, as we recall, returns the length of its list argument. The rex definition is:

```
length( [ ] ) => 0;

length( [F | R] ) => length(R) + 1;
```

The translation into Java, which could go inside the class definition above, is:

```
static int length(List L)
  {
  if( isEmpty(L) )  return 0;

  return length(rest(L)) + 1;
  }
```

Notice that each rex rule corresponds to a section of Java code. First we check whether the first rule applies by seeing if the list is empty. If it is not empty, we apply the function recursively and add 1.

As an alternate to implementation of the `length` function, we could use an iterative, non-recursive, solution:

```
static int length(List L)
  {
  int result = 0;

  while( nonEmpty(L) )
    {
    L = rest(L);         // "peel" the first element from the list
    result++;            // record that element in the length
    }

  return result;
  }
```

Although this version is non-recursive, it is perhaps more difficult to understand at a glance, as it introduces another variable to worry about. Depending on the compiler, however, this might well be the preferred way of doing things.

Note that the `length` function should not, and does not, modify its argument list. It merely changes the value of the local variable `L` which is a *reference* to a cell.

Now let's try another example, the function `append`. First in rex:

```
append( [ ], M ) => M;

append( [A | L], M ) => [A | append(L, M)];
```

then in Java:

```
static List append(List L, List M)
  {
  if( isEmpty(L) ) return M;

  return cons(first(L), append(rest(L), M));
  }
```

Notice that the pattern is very similar to the recursive implementation of length. In the case of `append` however, there is no clear and clean way in which the function could be implemented iteratively rather than recursively.

Finally, let's look at a function which was implemented with an accumulator argument earlier: `reverse`. In rex we employed an auxiliary function with two arguments, one of which was the accumulator.

```
reverse( L ) = reverse( L, [ ] );

reverse( [ ], R ) => R;

reverse( [A | L], R ) => reverse( L, [A | R]);
```

A literal translation into Java would be to have two functions corresponding to the two rex functions:

```
static List reverse(List L)
  {
  return reverse(L, null);
  }

static List reverse(List L, List R)
  {
  if( isEmpty(L) ) return R;

  return reverse(rest(L), cons(first(L), R));
  }
```

In the case of `reverse`, we can get rid of the need for the auxiliary function by using iteration. An alternate Java definition is:

```
static List reverse(List L)
  {
  List result = null;
  while( nonEmpty(L) )
    {
    result = cons(first(L), result);
    L = rest(L);
    }
  return result;
  }
```

This version is probably the preferred one, despite it being slightly removed from the original rex definition, since it does not introduce the complication of an auxiliary function.

Exercises

(You may wish to develop rex versions of these solutions first, then translate them to Java.)

1 •• Construct a Java function which will test whether an element occurs in an argument list.

2 •• Construct a Java function which will add an element onto the end of a list, returning a totally new list (leaving the original intact).

3 •• Construct a Java function which will produce as an open list the digits of an argument number in a given radix.

4 •• Construct a Java function which will produce a number given the list of digits in a given radix as an argument.

5 ••• Construct a Java function which will produce a sorted list of the elements from its argument list.

6 ••• Construct a Java function which will produce the list of all subsets of an argument list viewed as a set.

7 ••• Construct a Java function which will return, from a sorted list, a sorted list of the same elements with no duplicates.

5.4 Closed Lists

Some of the techniques for open lists can be used to implement closed lists. Recall that while open lists generally encouraging tail-sharing, closed lists provide a way to prevent. While open lists provide a nice mathematical programming style, dealing with closed lists, e.g. using destructive modification, should also be part of our repertoire. In some cases we use closed lists to save space. Rather than create a new list: we modify the elements or references directly in place. Closed lists can also save time: To append one list to another, we can get by just by modifying references rather than recreating the first list as function `append` does. Because modifying lists is more error prone than creating new ones, we must be more careful if we decide to do any form of sharing. Usually it is best to avoid sharing whenever lists are being modified destructively.

In a sense, a closed list can be implemented by putting a *wrapper* around an open list for which no sharing is to take place. In the absence of sharing, it makes sense to do things which we wouldn't wish to do with open lists, such as keep track of the *last* cell in the list and modify it destructively.

The usual way to provide a wrapper is through a list **header**, a particular object which identifies the list and through which initial accesses are made. Auxiliary information, such as the length of the list, can also be maintained in the header.

With open lists, we may or may not have a header. Our initial primitive exposition was without, and corresponds to implementations in rex and related languages.

The figure below shows a closed list, where auxiliary information, namely a reference to the last cell in the list, is maintained. A type definition for the header might be:

```
class closedList
{
Cell head;
Cell tail;
}
```

where `Cell` is as previously declared.

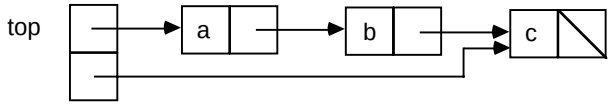


Figure 44: Example of a closed list with 3 elements: a, b, c.

Common uses of closed lists are data containers which maintain objects in a certain order and allow addition or removal only according to a pre-specified discipline:

stack - data are removed in the opposite order of insertion

queue - data are removed in the same order of insertion

We will say more about such containers in later chapters. The figures below depict these uses for linked lists. We leave it to the reader to provide code for the appropriate data operations for these abstractions.

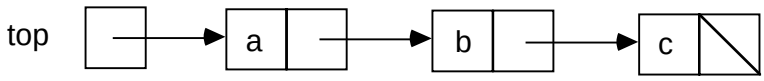


Figure 45: A stack implemented as a closed list.
The header contains a reference to the top cell.

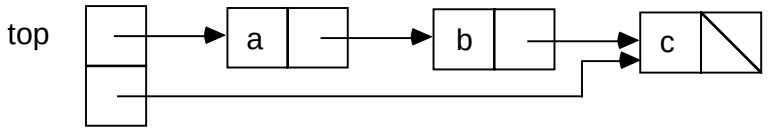


Figure 46: A queue implemented as a closed list.
The oldest cell is removed first.
Insertions take place after the youngest cell.

A more complete presentation of a closed list implementation will come once we have introduced object-oriented concepts in *Object-Oriented Computing*. For now, we will be content with a simple example of what can be done.

Appending Closed Lists

We will use the form of closed list described earlier, with a header that points to both the first and last element in the list. If the list is empty, we will assume that both of these references are null. Before writing any code, it is helpful to draw of picture of what is to happen. Then a series of statements can be constructed to carry out the plan. Finally, special cases, such as empty lists, must be dealt with to make sure the code works for them.

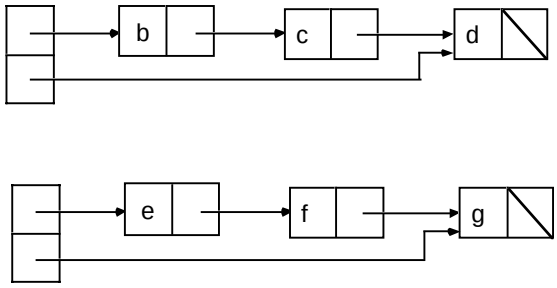


Figure 47: Two closed lists before appending

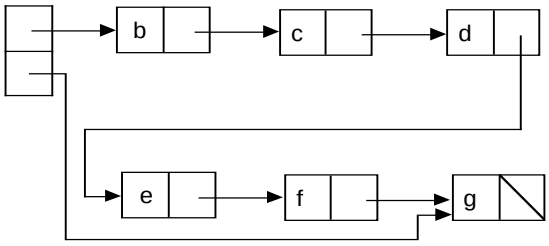


**Figure 48: Closed list after appending second to first.
The second list is no longer shown as a separate entity,
as it would not be advisable to use it once its cells are implicitly shared.**

Assume the following structural definition for the types closedList and cell:

```
class closedList
{
Cell head;
Cell tail;
}

class Cell
{
Item data;
Cell next;
}
```

In order to effect the appending of list `M` to list `L`, we need to do the following:

```
L.tail.next = M.head;  // connect the tail of L to the head of M

L.tail = M.tail;       // install the tail of M as the new tail of L
```

We also have to deal with the null cases. If `L` is empty, then `L.tail` is `null`, so we certainly don't want to dereference it. However, in this case we need to set `L.head` to `M.head`. On the other hand, if `M` is empty, then `M.head` is `null`, so setting `L.tail.next` to `M.head` does no harm. But in this case, `M.tail` will also be `null`. We want to leave `L.tail` as it was, pointing to the tail of `L`. So the final code, packaged as a procedure which modifies `L`, is:

```
void append(closedList L, closedList M)
{
if( L.tail == null )
  L.head = M.head;         // L is null, make L's head be M's
else
  L.tail.next = M.head;    // L is not null, connect L to M

if( M.head != null )
  L.tail = M.tail;         // M is not null, make L's tail be M's
}
```

Exercises

1 •• Construct a procedure *find* which takes a closed list and an argument of type `Item` and returns a reference to the first occurrence of a cell containing that element, or null if the element does not occur.

2 ••• Construct a procedure *reverse* which reverses a closed list in place. Be sure to handle the empty list case.

3 ••• Construct a procedure *insert* which destructively inserts an item into a closed list given a reference to the cell before which it is to be inserted. Assume that if this reference is null, the intention is to insert it at the end of the list.

4 ••• Construct a procedure *delete* which destructively removes an item in a closed list given a reference to the cell to be deleted.

5 ••• A *doubly-linked list* (DLL) is a form of closed list in which each cell has two references, pointing to both the next cell in the list and the previous cell in the list (the latter reference is 0 if there is no previous cell).

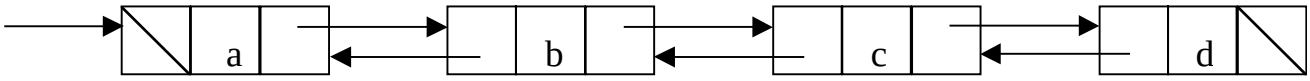


Figure 49: A doubly-linked list of four elements

Give a Java definition for a) the cell of a DLL, and b) a DLL. (Take into account the possibility of a DLL with no elements.)

Develop a set of procedures that do each of the following:

6 •• Find an item in a DLL based on its value. The result is a reference to the cell, or 0 if no such value was found.

7 •• Delete the cell pointed to by a given reference.

8 •• Insert a new cell with a given value following the cell identified by a reference.

9 •• Insert a new cell with a given value before the cell identified by a reference.

10 •• Concatenate two DLL's to form a new DLL.

11 •• Create a DLL with the same values as are contained in an open list.

12 •• Create an open list with the same values as are in a DLL.

13 ••• Think of some applications where a DLL is a more appropriate structure than an ordinary linked list.

14 ••• A *ring* is like a doubly-linked list in which the first and last elements are linked together, as suggested below. This type of structure is used, for example, in some text editors where searches for the next occurrence of a specified string wrap around from the end of the text to the beginning.

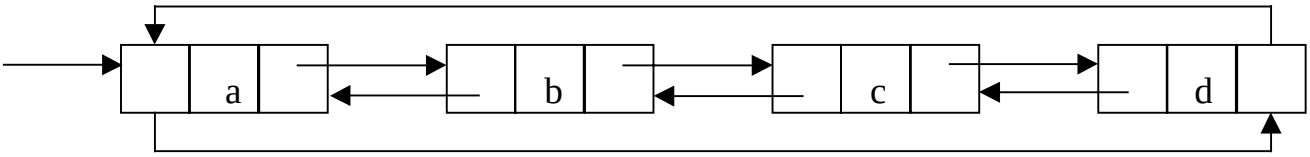


Figure 50: A ring of four elements

Repeat the previous two exercises substituting "ring" for DLL.

15 ••• A *labeled binary tree* (LBT) is structure constructed from nodes similar to those in a doubly-linked list, but the references have an entirely different use. An LBT is a branching version of an open list. Each cell has a data item (called the "label") and two references which themselves represent LBT's.

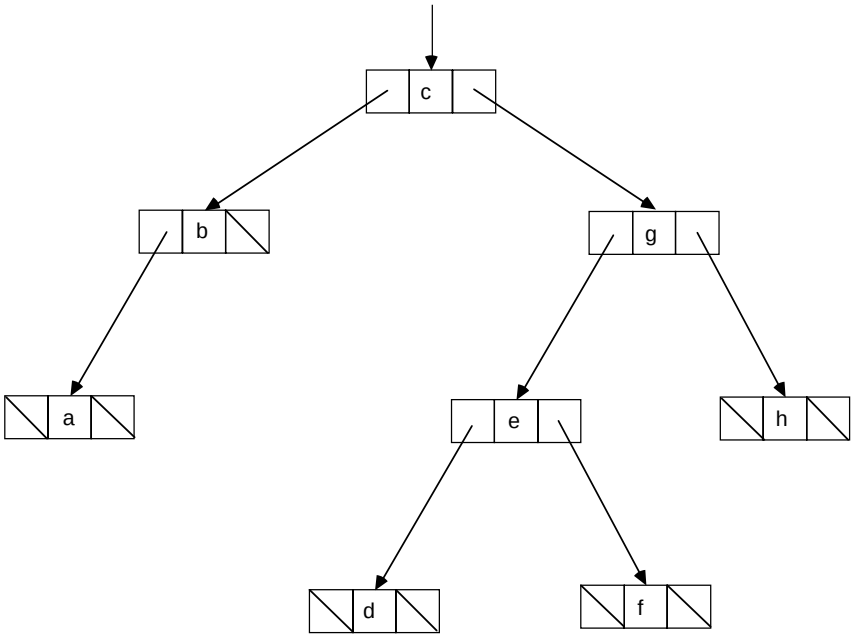


Figure 51: A labeled binary tree

16 ••• Develop a set of abstractions similar to the abstractions *cons*, *first*, *rest*, *null*, etc. for open lists.

17 ••• A *traversal* of an LBT is defined to be a linear list, the elements of which are in one-to-one correspondence with the nodes of the LBT. There are several standard types of traversals, enumerated below. Develop functions which produce each form of traversal from an LBT.

In each of the following cases, the traversal of an empty tree (represented by a null reference) is the empty list.

In an *in-order traversal*, the elements are ordered so that the root element is between an in-order traversal of the left sub-tree and the right sub-tree. An in-order traversal for the tree in the diagram is:

(a b c d e f g h)

since *c* is the root element, (a b) is an in-order traversal of the left sub-tree of the root, and (d e f g h) is an in-order traversal of the right sub-tree of the root. (These facts are established by applying the definition recursively.)

In a *pre-order traversal*, the elements are ordered so that the root element is first, followed by a pre-order traversal of the left sub-tree, then a pre-order traversal of the right sub-tree. For the example above, a pre-order traversal is

(c b a g e d f h)

In a *post-order traversal*, the elements are ordered so that the root is last, and is preceded by a post-order traversal of the left sub-tree, then a post-order traversal of the right sub-tree. For the example above, a post-order traversal is

(a b d f e h g c)

18 •••• In a *level-order* or *breadth-first traversal*, the elements are ordered so that the root is first, the roots of the two sub-trees are next, then the roots of their sub-trees, left-to-right, etc. For the example above, the level-order traversal is

(a b g a e h d f)

Develop a function that produces the level-order traversal of a LBT.

19 ••••• Show that the information in a traversal by itself is insufficient to re-establish the LBT from which it came. Is it possible to use two different traversals to re-establish the LBT? If not, demonstrate. If so, which pairs of traversals work? For those pairs, develop a function that constructs the tree given the traversals.

20 ••• Develop a formula for the number of null references in an LBT as a function of the number of nodes N. Prove your formula by induction.

5.5 Hashing

The principle of hashing combines arrays and lists to achieve an astounding effect: efficient time access to a large volume of data based on key words, numbers, or phrases stored in the data. We present here just one of many variations on the concept. The lists appear to be somewhat closed, but are essentially simple open lists with headers. Typically all addition can take place at the front end. As such, the lists are functioning as write-only stacks, the latter being discussed in more generality in the next chapter.

The problem addressed by hashing is to access "records", e.g. structs, according to some "key" value. The keys could be large numbers or strings for example. If a large number of such records are stored in an array, it can take considerable time to search the array to find the one corresponding to a given key. On the other extreme, we could use linear addressing to access an array by using the key as an index. However, for many such indices there will typically be no record, so much memory space could be wasted with unused locations. It would not be feasible to create such an array for more than a few hundred million keys given current computer technology.

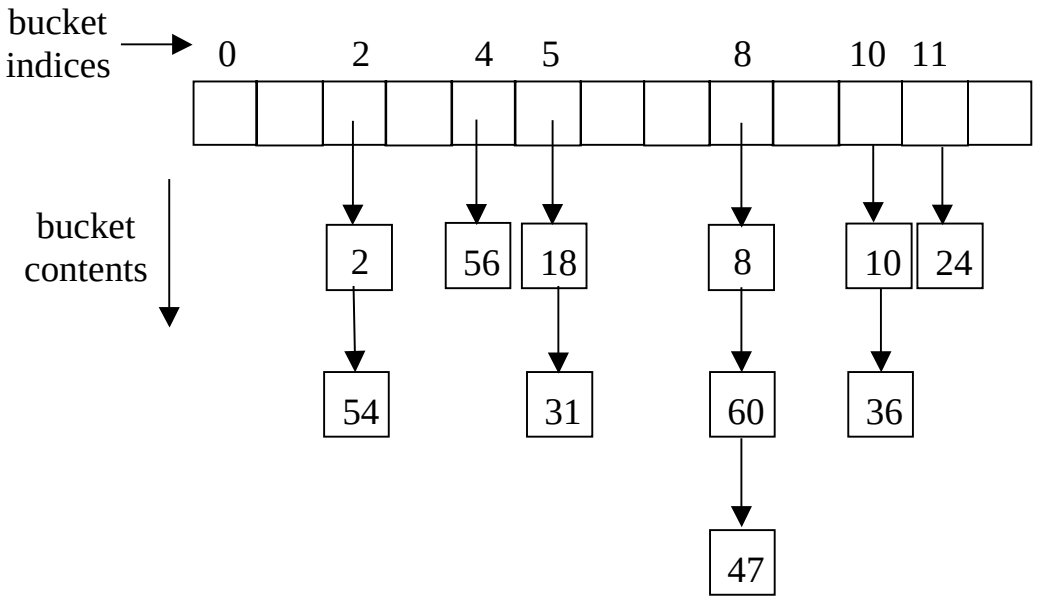


Figure 52: Array of 13 buckets, each a linked list, used for hashing. The numbers in the buckets represent key values.

Hashing "folds" the indexing array so that the same location is used for multiple records. These records are linked together in a list. The records corresponding to any one location are called a "bucket". The bucket is searched linearly. The trick to fast access is to keep the buckets small. This can be done by keeping the index array nominally large and having a way of distributing the records more-or-less "randomly" into the buckets. Based on only the key, we have to know how to find the bucket, both to insert the record in the bucket in the first place and to find out if a record with a given key is in the bucket. The overall structure, as illustrated in the figure, is typically called a *hash table*.

For the example above, we simply took the key value modulo the table size, 13, and used the result as an index. Thus the bucket for key 18 is 18 % 13 ==> 5, while the bucket for key 47 is 47 % 13 ==> 8. Typically such a simple bucket computation will not assure very random distributions. So rather than taking the raw key value mod the table size, we agree in advance on a function

h: key_values → integers

and use

```
h(k) % table_size
```

as our index of the bucket for key *k*. This kind of function is called a *hash function*. By careful choice of *h*, we can get very random distributions and handle arbitrarily large key values. We can even use strings or other structures as key values, by considering those structures to be numerals in a certain radix.

Example Hash Function

The following hash function, *hash_pdg* (for "pretty darn good") works effectively on strings, producing an unsigned long. Before using the resulting value to index the hash table, the value produced by the function is taken modulo the table size. This insures that indices are within range. The function works by using the integer values of successive characters in the string. An accumulator *h* is initialized to 0. Each character is added to *h* multiplied by a constant to obtain a new value of *h*. The multiplier has been chosen to randomize the result as much as possible.

```
unsigned long hash_pdg(char str[ ])
{
int multiplier = 131;
unsigned long h = 0;
int N = str.length();
for( int i = 0; i < N; i++ )
  {
  h = h*multiplier + str[i];
  }
return h;
}
```

The origin of the function is G. H. Gonnet and R. Baeza-Yates, 1991.

5.6 Principle of Virtual Contiguity

We conclude this chapter with a reference-based structure quite different from linked lists. This is an array-like structure for simulating large arrays from smaller ones. The key idea here is to approach the performance availed by the linear addressing principle, without the need for having a single contiguous array.

This principle is used in the structure of so-called virtual memory computers, which are now commonplace. We explained above how we need to have data stored in contiguous memory locations if we are to exploit the linear addressing principle. This requirement can present a problem when very large arrays are involved: it could happen that, at the time a request for a large array is made, the memory has become temporarily "fragmented". That is, there is enough total memory available in terms of the number of storage locations, but no contiguous block that is large enough to hold an array of desired size. The principle of virtual contiguity can be used to "piece together" smaller blocks, with a slight penalty in access time.

Suppose we need to allocate an array requiring 10^6 bytes of memory but there is no block available of that amount. Suppose that there are 100 blocks of 10^4 bytes each available in various blocks. The principle of virtual contiguity allows us to piece these blocks together to give us our 10^6 blocks. This piecing is done by adding a second level of indexing, as implemented by an index array 100 addresses in length. Call the virtual array A and the index array T (for "table"). The values T[0] .... T[99] hold the base addresses of our 100 blocks of 10^4 bytes each. Thus, to access A[i], we first compute i / 100 (using integer division) to find out which block to use, then use i % 100 to access within this block. In equations:

$$\&A[i] \equiv T[i / 100] + i \% 100$$

where &A[i] means the *address* of A[i] in memory.

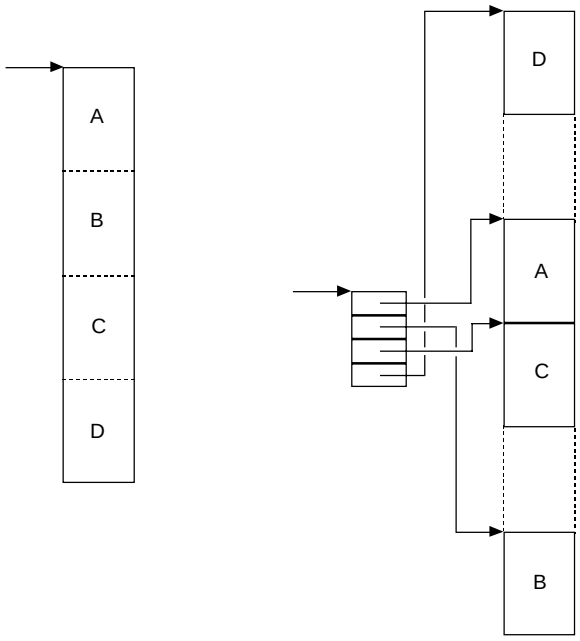


Figure 53: Virtual contiguity:
left: array as perceived by program;
right: array as implemented in linear address-space memory

Virtual Memory

In a true virtual memory system, an additional twist is used along with the principle of virtual contiguity: a table entry T[i] can contain either a memory address or a disk address (as determined by an additional bit in each word). The block being referenced need not be in memory at the time the reference is attempted; instead it is on disk and is brought in on demand. This allows us to "time-share" a relatively small amount of main memory by swapping blocks to and from the disk, giving the illusion of a very large amount of memory. The cost paid for this is a slightly slower overall access time, plus a large penalty if the desired block has to be brought in from disk.

In a virtual memory system, blocks are referred to as **pages** and the array T is called a **page table**. Systems are designed so that they try to keep the most-likely-to-be referenced pages in memory whenever possible. The workability of such schemes relies on what is called the **principle of locality**: programs tend to refer to the same pages over and over again in a nominal time interval. Obviously a virtual memory system does not strictly follow the linear addressing principle of uniform access time when a page is not present on disk. Nonetheless, most people design algorithms as if the linear addressing principle still held, relying on the principle of locality to make linear addressing a good

approximation. Fortunately for the applications programmer, the mechanisms implementing virtual memory are carried out transparently by the system.

Exercises

1. •• Write a program which will do a fast spelling check by using a dictionary stored as a hash table. Populate the table from a dictionary file, such as `/usr/dict/words` which is available in most UNIX® systems. Compare the speed of your program to one that searches the dictionary sequentially.

2. •• Implement a system of arrays that uses the principle of virtual contiguity.

5.7 Chapter Review

Define the following terms:

append
bucket
cell
class
closed list
dereferencing
doubly-linked
hash function
hashing
header
labeled binary tree
level-order
linear addressing principle
linked list
null reference
open list
page
pre-order traversal
post-order traversal
queue
recursive type
ring
virtual memory

5.8 Further Reading

G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures, 2nd ed.*, Addison-Wesley, 1991. [Concise reference on a wide range of algorithmic techniques, with code. Moderate to Difficult]

6. States and Transitions

6.1 Introduction

This chapter introduces the idea of states and transitions. It talks about representing transitions by rules, noting the difference between deterministic and non-deterministic systems. It indicates how programs can be thought of in terms of states and transitions, and how ordinary imperative programs can be transformed into functional ones. It also discusses Turing machines, and a variety of related ideas. Most of these ideas are tied together with the thread of "functional programming" already introduced. Then in the next chapter we will see how these ideas play into "object-oriented programming", which is at the other end of the spectrum.

Among the most pervasive notions in computing is that of "state". We define the *state* of a computation as a set of information sufficient to determine the future possible behaviors. In other words, the state tells us how the system can change. It also can tell us something about what has happened in the past, if we know it to have been started in a particular initial state. It doesn't usually tell exactly what has happened, but rather conveys some *abstraction* of what has happened.

A typical computational system starts with the initial state that embodies the input to the computation, and continues until termination, at which point the output can be extracted from the final state.

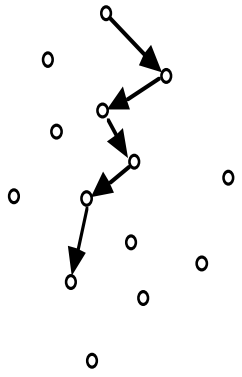


Figure 54: The progression of a system from state to state

In the context of a set of rewrite rules, the state determines the possible *end results* of the computation, if any. We can stop the computation at any point and resume it, so long as we are careful to record the state at the stopping point. This is of major importance, for example, in computer *operating systems*, which are sets of programs that control the usage of major resources, such as input-output devices, available on the computer. Operating systems frequently switch from one computation in progress to another, for

purposes of best exploiting the available resources. The state of a program is saved in memory, enabling the system to restart the program later on. For example, if a program needs to wait for a critical resource to become available, the operating system will suspend that program and run another program meanwhile. The state concept is also useful when we wish to simulate a program's execution mentally or with pencil and paper. If we get tired, we can record the state and resume the activity later.

States in digital computation are a lot like the states we encounter in solving certain kinds of puzzles. Here the configuration of the puzzle completely determines the state. Many such puzzles can be solved using the techniques of graph searching, such as breadth-first search. These entail being able to detect whether we encountered the current state earlier, in which case we would not want to repeat the same set of moves, for that would lead to no progress.

Towers of Hanoi Example

In this famous puzzle, N discs of decreasing sizes are stacked on one of three spindles, the other two of which are initially empty. The problem is to move all N discs, one at a time, from the first spindle to the second, maintaining the constraint that at no time is a larger disc placed atop a small one.

The state in the case of this puzzle is the way that the disks are stacked on the spindles. The initial state for N = 4 would be as shown below.

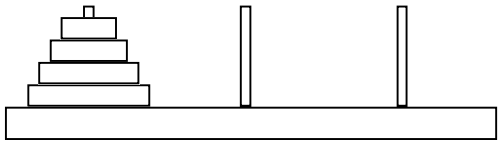


Figure 55: ***The Towers of Hanoi puzzle for N = 4, initial state.***

The desired final state is:

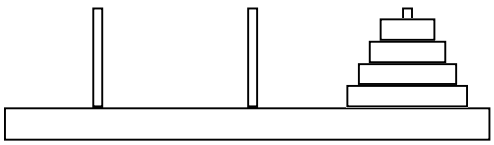


Figure 56: The Towers of Hanoi puzzle for N = 4, final state.

The following is an example of an illegal state, one that cannot occur during a solution, since a larger disk is atop a smaller one.

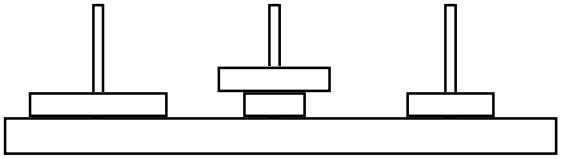


Figure 57: An illegal state in the Towers of Hanoi puzzle

To provide a tool for discussion of the solution to such puzzles, as well as for computational systems in general, we can exploit the mathematical concept of "binary relation" as described in the next section.

6.2 Transition Relations

To apply relations to the discussion of states, the set of pairs of states (s, s') such that one move will take the system from s to s' is called the **transition relation**. Typically we will represent a transition relation by =>. A single pair (s, s') such that s => s' is called a **transition**. We sometimes say that s' is a **successor** of s when there is a transition (s, s') and that s is a **predecessor** of s'.

Below we show some possible transitions between states for the Towers of Hanoi puzzle with only 2 disks. With this bird's-eye view of the states and transitions, it is possible to find a sequence of transitions that lead from any state to any other, and, in particular, to solve the puzzle. While many solutions are possible, there is a unique *shortest* solution represented by the path from top center to bottom left. Quite evidently, any deviation from this path adds more steps than are necessary.

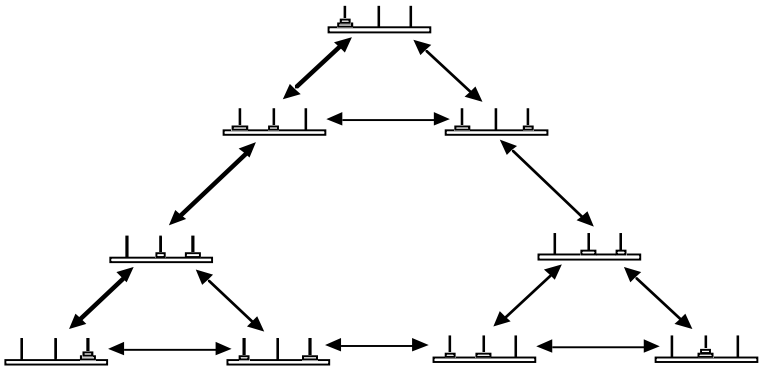


Figure 58: Bird's-eye view of possible state transitions in the Towers of Hanoi puzzle with 2 disks. The shortest solution is the sequence of transitions from the top state downward to the lower left.

Notice that in this particular puzzle, the transition relation happens to be *symmetric*, in that for each move there is an opposite move that returns the puzzle to the previous state. Accordingly, the state-transition graph can be represented as an undirected graph. Of course, this is not the case for puzzles in general. Many puzzles have one-way transition relations.

The scheme of constructing the state-transition diagram can be used to solve the puzzle for larger values of N as well. The only problem here is that we have a "combinatorial explosion" of states as we consider larger N. Fortunately, we can get an understanding of the state-transition structure of this particular puzzle without going to very large N. Examine the state-transition diagram above. Notice that there are three triangular patterns embedded within an overall triangular pattern. Consider the top three states. This triangular pattern shows the motions of the top disk with the bottom disk remaining fixed on the first spindle. Similarly, the lower-left triangle has the bottom disk fixed to the third spindle, and finally, the lower-right triangle has the bottom disk fixed to the second spindle. We could, therefore, have constructed this diagram as follows:

1. Diagram the states for the puzzle with a single disk. This diagram is just a triangle, with each vertex representing one of the three positions for the single disk.

2. For two disks, consider a similar triangle with the vertices representing the positions of the bottom disk. For each of these vertices, embed a single-disk triangle.

This construction is suggested by the following pattern:

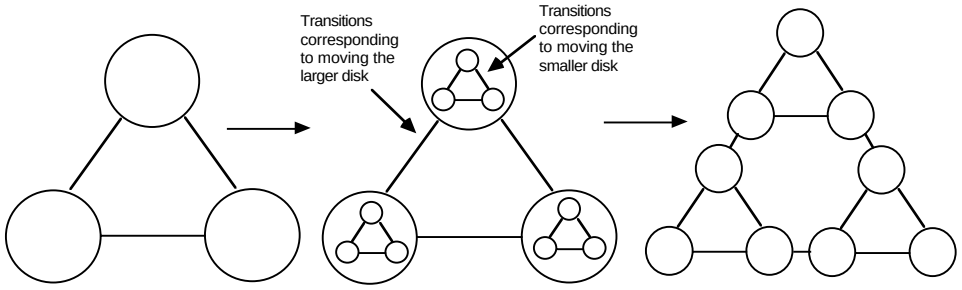


Figure 59: Embedding pattern for constructing a 2-disk transition diagram from 1-disk transition diagrams

In this analysis, we are starting to think "recursively", an important skill we will continue to hone throughout this book. The same triangular pattern can be used to construct a diagram for N+1 disks from three N disk diagrams for any N. For example, the following diagram for 3 disks was obtained by taking three of the diagrams on the right-hand side above and placing them inside a 1-disk diagram.

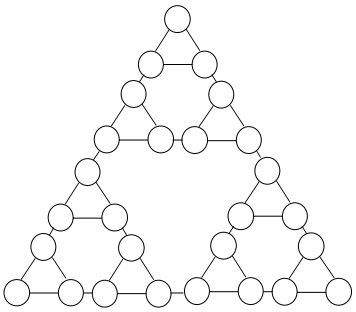


Figure 60: The state-transition diagram corresponding to the 3-disk structure

One thing this construction tells us is that every time we add a new disk, we triple the number of states that have to be considered. For an N-disk puzzle, there are thus 3^N states. While we are at it, we can make some other observations about these diagrams:

- The start and end points of the puzzle are always at the extreme vertices of the triangle.

- The shortest path from one of these extremities to another will always entail 2^N nodes, i.e. it takes 2^N - 1 moves to solve an N-disk puzzle.

Both of these assertions can be proved by induction. We could also continue this construction *ad infinitum* by repeatedly placing the 1-disk triangle inside the inner-most vertices of the diagram. In the limit, we would have a "self-similar system", sometimes called a *fractal*. The limiting case is self-similar because inside each of the outermost vertices we have contained an exact replica of the entire system. A related geometric construction is known as the *Sierpinski Gasket*.

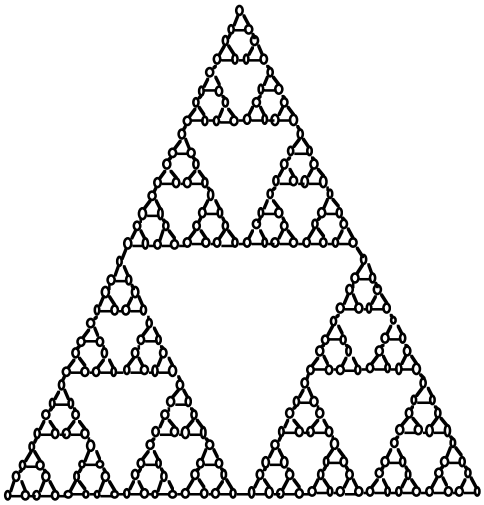


Figure 61: Approximation to a Sierpinski Gasket

Of course, we don't expect that all state-transition diagrams will generate comparable artwork.

Reachability Relations

From a transition relation ➪, we will have need for the accompanying **reachability relation** (also called the **reflexive transitive closure** of ➪), which will be represented ➡. The reachability relation ➡ means that we can get from one state to another by a series of zero or more intervening states. More precisely:

$$T_0 \Rightarrow T_n$$

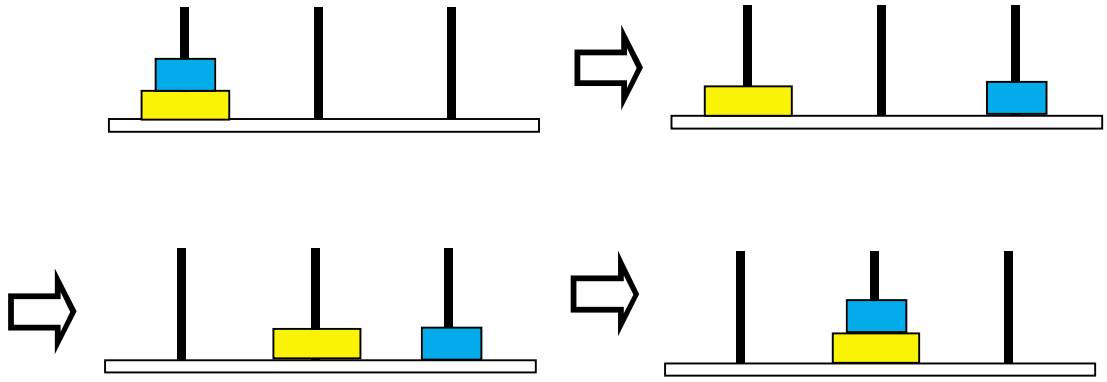
means that there are terms $T_1, T_2, \ldots, T_{n-1}$ $(n \geq 0)$ such that

$$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_n$$

Example – Towers of Hanoi Reachability Relation

In the 2-disk Towers of Hanoi Puzzle, we can assert

based on the following series of individual transitions:



In later sections, we will have further occasion to use this notation in connection with computational systems.

6.3 Transition Rule Notation

The transitions for the Towers of Hanoi puzzle are determined by very explicit rules. The most succinct way we can state these is:

- A transition can involve moving only one disk
- A transition must result in a legal state.

These rules, then, characterize an infinite set of possible transitions (when we consider the puzzle for all values of N, the number of disks). In like fashion, a computational system will have transition rules. Unlike puzzles, computational rules will often be **deterministic**: that is, there is at most one successor to any state. A system, such as many puzzles, where for some state there are at least two possible successors, is called **non-deterministic**. Non-deterministic systems have some important technical uses outside of puzzles, as we shall see later. One of these uses is as *grammars*, ways of representing the syntax of programming languages.

A state-transition system is called

deterministic if every state has at most one successor.

non-deterministic if a state may more than one successor.

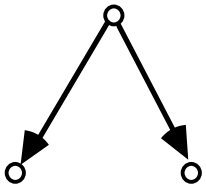


Figure 62: Scenario not occurring in a deterministic system

To codify the rules of the Towers of Hanoi puzzle, we can represent a state as a pattern

towers(L, M, N)

where L, M, and N represent the disks stacked on each tower. We will then be able to express rules for changing from one state to another in the form

towers(L, M, N) => towers(L', M', N').

where the primed towers are derived from the unprimed ones according to a particular pattern, or by

towers(L, M, N) => *Condition* ? towers(L', M', N').

in the case that the transition holds only if *Condition* on towers is satisfied. Note that these *look* a lot like rex rules. However, they end with a period rather than a semi-colon. This is because the rules are not actually in rex, but are implemented in a language called *Prolog*. A Prolog program is provided that can determine from such a set of rules whether the puzzle has any solutions, or a shortest solution. It does this by enumerating reachable states, and therefore will only work in the case that the number of such states is not too large.

To represent the disks on the towers themselves, we could number the disks 1, 2, 3, 4 and record the numbers on each of the three spindles as lists from top to bottom. Thus the initial state is

```
towers([1, 2, 3, 4],  [ ], [ ])
```

and the desired final state is

```
towers([ ], [ ], [1, 2, 3, 4])
```

(Note that this is not the state representation that we used in counting the number of states. There is no requirement that it be the same.) For the 2-disk puzzle, the state-transitions would be shown as follows, where we have omitted the outer towers(…) to keep the diagram from getting too cluttered:

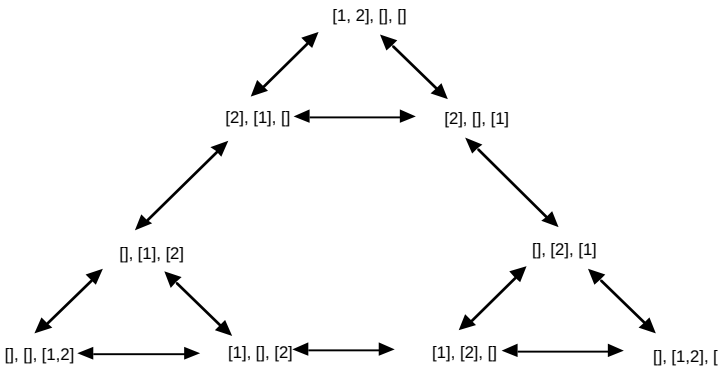


Figure 63: Coded state transitions for the Towers of Hanoi puzzle

For a fixed value of N, one could enumerate all of the states and transitions possible for an N-disk Towers of Hanoi problem. However, none of these enumerations expresses the general rules for transitions, because we don't yet have a way to describe manipulations on *arbitrary* lists. We now digress to describe a notation for presenting such rules. Then we will give a complete set of rules for the puzzle.

The transition rules are represented in one of the following forms:

S => S'.

or

S => C ? S'.

where S and S' are state forms and C is a *guard condition*. The condition states additional constraints governing when the use of the rule is possible. So the rules for moving a disk from the first spindle are:

```
towers([A | L], [], N) => towers(L, [A], N).                    % 12e

towers([A | L], M, []) => towers(L, M, [A]).                    % 13e

towers([A | L], [B | M], N) => A < B ? towers(L, [A, B | M], N). % 12n

towers([A | L], M, [B | N]) => A < B ? towers(L, M, [A, B | N]). % 13n
```

The characters on lines following `%` are comments. The first two rules govern movement of a disk from the first spindle to an empty spindle. The third and fourth rules govern movement of a disk from the first to a non-empty spindle, and require that the disk being moved is smaller than the disk already on top of that spindle. Here we are assuming that the disks are identified with numbers, so that `A < B` means that disk `A` is smaller than disk `B`. The naming of the rules is of the form *i j S*, where *i* and *j* are spindle numbers (1, 2, or 3) giving the number of the "from" and "to" spindles, and S is either `e` or `n`, designating whether the target spindle is empty or non-empty. For example, rule `12n` designates moves from spindle 1 to spindle 2, where spindle 2 is non-empty. The reason we have to separate the empty and non-empty cases is so that we can make the necessary comparison

to determine whether a disk is being put atop a larger disk or not, in the case of a non-empty spindle.

There are eight more rules in two sets paralleling these, showing how disks can be moved from the second spindle and from the third spindle, respectively.

```
towers(N, [A | L], []) => towers(N, L, [A]).                    % 23e

towers([], [A | L], M) => towers([A], L, M).                    % 21e

towers(N, [A | L], [B | M]) => A < B ? towers(N, L, [A, B | M]). % 23n

towers([B | N], [A | L], M) => A < B ? towers([A, B | N], L, M). % 21n

towers([], N, [A | L]) => towers([A], N, L).                    % 31e

towers(M, [], [A | L]) => towers(M, [A], L).                    % 32e

towers([B | M], N, [A | L]) => A < B ? towers([A, B | M], N, L). % 31n

towers(M, [B | N], [A | L]) => A < B ? towers(M, [A, B | N], L). % 32n
```

There are thus twelve rules in all, six for each combination of ij to an empty spindle, and six to a non-empty spindle. Because there is no limit on the size of the lists, each transition rule represents an infinite set of transitions.

For example, in the previous figure, one series of state transitions (going downward and to the left) is the following:

```
towers([1, 2, 3, 4], [], [])
towers([2, 3, 4], [1], [])
towers([3, 4], [1], [2])
towers([3, 4], [], [1, 2])
towers([4], [3], [1, 2])
```

To see that the rules justify these transitions,

```
towers([1, 2, 3, 4], [], []) => towers([2, 3, 4], [1], [])  by 12e

towers([2, 3, 4], [1], []) => towers([3, 4], [1], [2])      by 23e

towers([3, 4], [1], [2]) => towers([3, 4], [], [1, 2])      by 23n

towers([3, 4], [], [1, 2]) => towers([4], [3], [1, 2])      by 12e
```

It should be emphasized that these rules by themselves do not *solve* the puzzle. They only articulate the legal transitions. However, these rules can be the input of a relatively simple program that does solve the puzzle without additional intellectual effort. Such a program will be further discussed in chapter *Computing Logically*. Later on in the current chapter, we will give a different set of rules for solving the puzzle directly, i.e. rules that *program* the solution.

Deterministic Solution of the Towers of Hanoi

This example makes use of recursive rules to solve the Towers of Hanoi puzzle. Let us try to discover a general method for solving this problem for N discs. We want to create a function that takes the number of discs N as an argument and returns a list of "instructions" for moving the discs. Each instruction will itself be a list, of the form:

```
["move disk ", D, " from ", A, " to ", B]
```

where D, A, and B are numbers of disks and spindles.

It is reasonable to try to get recursion to work for us. This would entail breaking an N disc problem down into lesser problems, e.g. N-1 disc problems. In order to move N discs from a spindle A to a spindle B, we might try the following:

> Move top disc from spindle A to spindle C.
>
> Move N-1 discs from spindle A to spindle B.
>
> Move top disc from spindle C to spindle B.

Unfortunately, this doesn't work. When the top disc has been moved to spindle C, it blocks the use of spindle C for subsequent moves from A, since in a legal state the top disc must be smaller than the other discs below it on A.

A different attempt would be to recursively move the top N-1 discs from A to C, move the bottom one disc to B, then move N-1 discs from C to B. This approach has the virtue that the bottom disc can be ignored while all of the other motion is taking place, since it must be larger than all of the other N-1 discs: no illegal states will be introduced.

This second attempt can be converted into a method, expressed recursively as follows:

> To move N discs from a spindle A to a spindle B:
>
> > If N = 0, there is nothing to do.
> >
> > If N > 0, then:
> >
> > > move (N-1) discs from A to C, where C is the third spindle other than A and B;
> > >
> > > move one disc from A to B;
> > >
> > > move (N-1) discs from C to B.

Since the first and third steps of the recursion can be done with the one disc in the second step unaffected, the desired constraint is maintained.

Obviously we are letting recursion work for us here, in fact with *two* recursive task calls to solve one task. We wish to give a set of rules that will solve this problem, in the sense that the ultimate result will be a list of pairs indicating single moves from one spindle to another. The spindles will be numbered 1, 2, 3. The discs will be numbered 1, 2, …, N to smallest to largest. We will represent a stack of discs to be moved as a list $[d_0, d_1, \ldots, d_N]$, smallest first.

We transcribe our rules into *rex* as follows, where `towers(L, A, B, C)` means move the stack of discs `L` from spindle `A` to `B`, with `C` as the other spindle (`A`, `B`, and `C` will vary from term to term).

```
towers([ ], A, B, C) => [ ];

towers([ D | L ], A, B, C ) =>
  append( towers(L, A, C, B), [move(D,A,B) | towers(L, C, B, A)] );
```

programmed rex solution rules for the Towers of Hanoi problem

Here `move(D, A, B)` rewrites to some term depending on how we wish to represent the move, e.g. we could use the rule

```
move(D, A, B ) => ["move disk ", D, " from ", A, " to ", B];
```

Since the exact form of move is not important, we shall not rewrite terms of the form `move(D, A, B)` further in the answers but just leave them as is. To complete the solution to our problem's specification, we need an interface function that takes the number of discs as an argument and calls *solve*. Given the number `N`, we need to create a stack of discs numbered [1, 2, …, N]. This can be accomplished by the function *range* given earlier. So the interface function, which we call tower, is

```
tower(N) => towers(range(1, N), a, b, c);
```

where a, b, and c are the names of the three spindles.

A different version of *solve* that eliminates the append use in favor of an accumulator argument (the last argument of `towers1` in this case) can be expressed as:

```
towers(L, A, B, C) => towers1(L, A, B, C, [ ]);

towers1( [ ], A, B, C, Moves) => Moves;

towers1( [D | L], A, B, C, Moves) =>
  towers1(L, A, C, B, [ move(D,A,B) | towers1(L, C, B, A, Moves)] );
```

programmed solution rules for the Towers of Hanoi problem using an accumulator

A Basic Counting Principle

In order to get an idea of the magnitude of certain problems such as those involving states, it is helpful to be able to count the sizes of sets without actually enumerating their members. One intuitive way to count is to describe a thought experiment that would, if actually conducted, enumerate the members and use it to determine the number of members, or at least an upper bound on the number of members.

Perhaps the simplest counting principle involves the notion of the **Cartesian product** of sets, designated by x. If A and B are two sets, then

$$A \times B$$

stands for the set of all *ordered pairs*, the first element drawn from A and the second drawn from B. Similarly, if N sets are mentioned, then

$$A_1 \times A_2 \times \ldots \times A_N$$

designates the set of all *ordered N-tuples*, the i^{th} component of which is drawn from A_i. For example, if A = {1, 2} and B = {a, b, c}, then A x B = {(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)}.

Let's use |S| to denote the size of S, i.e. the number of elements in S. Then the counting principle is:

$$| A_1 \times A_2 \times \ldots \times A_N | = | A_1 | \ | A_2 | \ldots | A_N |$$

basic counting principle

where the juxtaposition on the right is ordinary numeric product. To describe this as a thought experiment, consider the case N = 2. Each element of $| A_1 \times A_2 |$ consists of a pair, the first element of which is drawn from A_1 and the second element of which is drawn from A_2. The experiment consists of enumerating all possible ways to construct such a pair. There are $| A_1 |$ ways to choose the first element in the pair. For each of those choices, there are $| A_2 |$ ways to choose the second element in the pair. Therefore we have a total of $| A_1 | \ | A_2 |$ ways to choose pairs.

The extension of this argument to the general case is straightforward. We would use induction on N. For N = 1, there is nothing to prove. For N > 1, we can adopt the inductive hypothesis that

$$| A_1 \times A_2 \times \ldots \times A_{N-1} | = | A_1 | \ | A_2 | \ldots | A_{N-1} |$$

Now consider this as a single set and add on A_N. That is, $A_1 \times A_2 \times \ldots \times A_N$ has the same number of elements as

$$(A_1 \times A_2 \times \ldots \times A_{N-1}) \times A_N$$

(although these two sets are technically slightly different; why?). So perform the same argument that we did for N = 2 to conclude that

$$| A_1 \times A_2 \times \ldots \times A_{N-1} \times A_N | = | A_1 | \, | A_2 | \ldots | A_{N-1} | \, | A_N |$$

As a special case, consider the situation where each A_i is the same, say A. Then we have

$$| A \times A \times \ldots \times A | = |A|^N$$

where the superscript denotes raising |A| to the N^{th} power.

Example

The number of (legal) states in the N-disk Towers of Hanoi puzzle is 3^N. To see this, consider that each state is representable by an N-tuple over the set {1, 2, 3}. The i^{th} element of the N-tuple is the tower on which the i^{th} disk resides.

Exercises

The following are various puzzles used as test cases in computer science. For each puzzle, describe the legal states. Devise a linear notation for the states. Describe the possible state-transitions informally (you need not use the formal rule notation we presented, although it is worth trying to see if it will work. When necessary, just state guard conditions informally.). Sketch a portion of the state-transition diagrams.

Note that in some cases (e.g. peg solitaire) it is possible to give a single set of rules that fit many sizes of puzzle. You should do this whenever possible. In others (e.g. the N^2-1 puzzles), formulating a general rule for many sizes of puzzle appears more difficult; the rules seem to need to be tailored to the size of the puzzle. This is not to say it can't be done. In such cases, it is acceptable to use a small instance of the puzzle, e.g. the 8-puzzle.

1 •• The linear peg solitaire(N) puzzle (a different puzzle for each N). This puzzle is played on a board of 2N-1 holes in which N pegs of each of two colors are placed, as shown below. The objective is to interchange all the pegs of the two colors. A legal transition involves either (a) moving a peg forward into an adjacent hole in the direction of its intended home; (b) jumping a peg forward over another peg and into an adjacent hole.

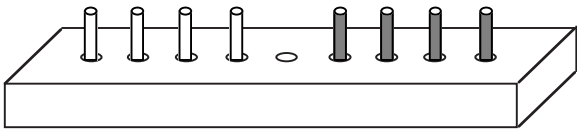


Figure 64: Peg solitaire: white pegs move or jump a peg only to the right. Colored pegs move or jump a peg only to the left.

The following shows another possible state of the puzzle:

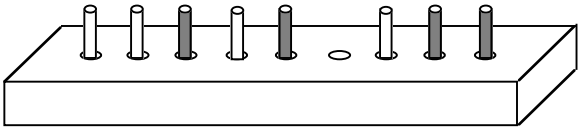


Figure 65: Peg solitaire after a few moves and jumps

The desired final state of the puzzle is as shown below:

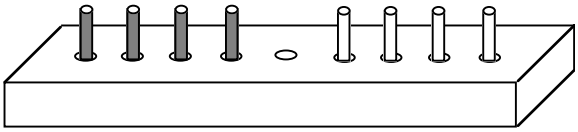


Figure 66: Desired final state of peg solitaire

[Hint: Model the sequence of pegs on either side of the hole as separate lists, constructing the left-hand list in the right-to-left peg direction.]

2 ••• Water jugs puzzles. There are several jugs of known capacities, each an integer number of units. One of the jugs is filled with water. The problem is to get a specified number of units of water into one of the jugs. A common example is that the jugs have capacities of 3, 5, and 8 liters, the 8 liter jug is the full one, and the objective is to obtain exactly 4 liters.

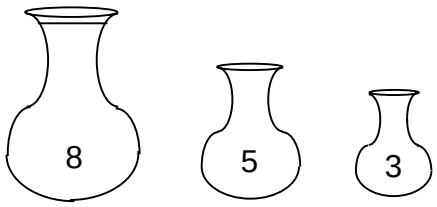


Figure 67: A water jug puzzle. The 8 liter jug is full. What transitions give us 4 liters in one jug?

3 ••• Give a rex program for generating a solution to the Towers of Hanoi puzzle given an arbitrary (legal) starting state.

4 ••• Suppose we wished to construct the state-diagram for a puzzle similar to the Towers of Hanoi, but with four towers instead of three. Describe the structure of the state-transition diagram.

5 ••• The N^2-1 puzzles (e.g. the 15 puzzle, the 8 puzzle, the 3 puzzle, etc.) This puzzle is played on an N-by-N board. There are N^2-1 tiles, numbered 1 through N^2-1. This leaves one space for a tile blank. The objective is to get from a given state of the puzzle to the state in which the tiles are all in order, by sliding blocks vertically or horizontally (not diagonally) onto the adjacent blank space.

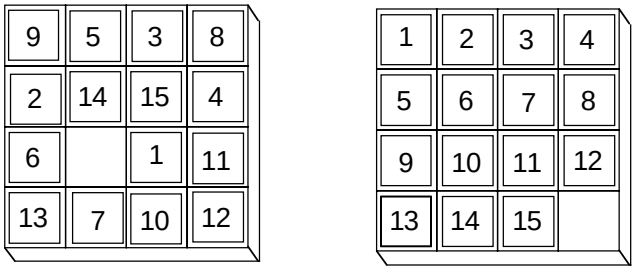


Figure 68: Two states of the 15 puzzle

Draw the complete state transition diagram for the 3 puzzle (played on a 2-by-2 board).

6 ••• The previous exercise introduced the N^2-1 puzzles. Express the 8-puzzle in the rule notation.

7 •• A puzzle (or its underlying transition relation) is called *strongly connected* if for any two states T_0 and T_1, it happens to be the case that $T_0 \Rightarrow T_1$. Is the 3-puzzle (described in the preceding problem, with N = 2) strongly connected?

8 ••• Give an argument that shows that the Towers of Hanoi puzzle, for any fixed number of disks, is strongly connected. (Hint: Use induction on the number of disks.)

9 ••• The Chinese rings puzzle. N rings are wired in a particular way through an elongated loop. Each loop is permanently attached to a wire, the opposite end of which is permanently connected to a bar. On its way from the bar to the ring, the wire passes through the ring on the immediate right. The objective is to remove the loop from all of the wires and their rings, so that the loop can be completely

separated from the rest of the puzzle. Give a set of rules that produce a list of moves for solving the puzzle deterministically.

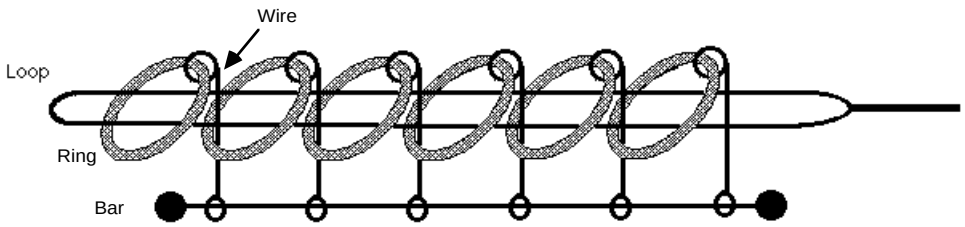


Figure 69: The Chinese rings with N = 6, before the loop has been removed from any ring

[Hint: Pursue an analysis similar to the deterministic solution to the Towers of Hanoi. This can be done purely based on the following analysis of the constraints of the puzzle: (a) The loop can be removed from the *leftmost* wire, or put back through it, in any state. (b) The loop can be removed from a non-leftmost wire w, or put back through w, provided that the wire w' to the immediate left of w goes through the loop and none of the wires to the left of w' go through the loop. So, for example, the first few states in the solution of the puzzle might be (numbering the wires 1, 2, 3, … left-to-right):

Take loop off wire 1.
Take loop off wire 3.
Put loop on wire 1.
Take loop off wire 2.
Take loop off wire 1.
Take loop off wire 5.
…

10 ••• What is the minimum number of moves required to solve the 6-ring puzzle?

11 ••• How many states are there in an N^2-1 puzzle (e.g. 15-puzzle) as described earlier), for a given N?

12 ••• How many states are there in an N-ring Chinese ring puzzle?

6.4 Rules for Assignment-Based Programs

In this section, we demonstrate how programs constructed in terms of sequences of assignment statements, loops, etc. can be recast as rule sets in rex, even though rex itself does not have an assignment construct. This demonstration is important in understanding the relationship between iteration and recursion.

Assignment-based programs use assignment to (i.e. changing the value of) various **program variables** to do their work. A program variable is an object, the value of which can change through an appropriate command. The form of an assignment will be assumed to be (using Java syntax)

Variable = *Expression*

meaning that the value of *Expression* is computed, then the value of *Variable* becomes that computed value. To give an equivalent rule-based program, we can use rule argument variables to play the role of assignable variables.

The *state* of an assignment-based program is a mapping from the names of variables to their values, in combination with the value of the instruction pointer (or "program counter") indicating the next instruction to be executed. To convert an assignment-based program to a rule-based one, associate a function name with each point before an instruction. Associate the arguments of the function with the program variables. We will then show how to derive appropriate rules.

Factorial Flowchart Example

Consider the following flowchart for a program that computes N-factorial. There are 6 places at which the instruction pointer can be, labeled 0..5. There are three program variables: `F`, `K`, and `N`. `N` is unchanging and represents the value for which the factorial is to be computed. `F` is supposed to contain the value of `N` factorial upon exit. `K` is used as an internal counter.

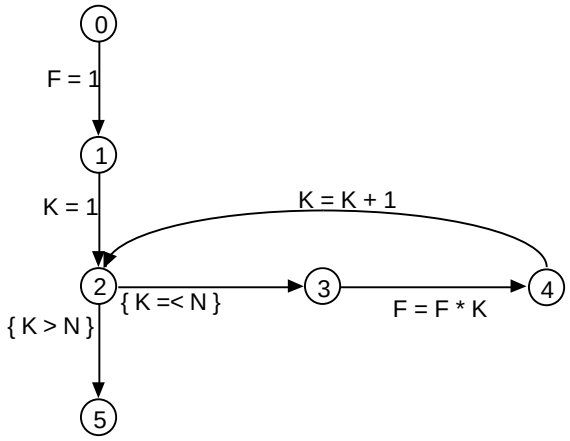


Figure 70: Flowchart for a factorial program.
The labels in braces {…} represent test conditions.
Labels of the form Var = Expression represent assignments.

We will first present the rules corresponding to the flowchart program, then indicate how they were derived. We let `fac` be the function being computed. We introduce symbols

f_0, …, f_5 as auxiliary functions. We use `arb` to indicate an *arbitrary* value (used for example to denote the value of an uninitialized assignable variable). The rules corresponding to the factorial flowchart are:

```
fac(N) => f0(N, arb, arb);

f0(N, K, F) => f1(N, K, 1);    // corresponds to assignment F = 1;

f1(N, K, F) => f2(N, 1, F);    // corresponds to assignment K = 1;

f2(N, K, F) => (K <= N) ? f3(N, K, F);

f2(N, K, F) => (K > N) ? f5(N, K, F);

f3(N, K, F) => f4(N, K, F*K);  // corresponds to assignment F = F*K;

f4(N, K, F) => f2(N, K+1, F);  // corresponds to assignment K = K+1;

f5(N, K, F) => F;
```

Let us verify that these rules give the correct computation of, say, `fac(4)`:

```
fac(4)            =>
f0(4, arb, arb)   =>
f1(4, arb, 1)     =>
f2(4, 1, 1)       =>
f3(4, 1, 1)       =>
f4(4, 1, 1)       =>
f2(4, 2, 1)       =>
f3(4, 2, 1)       =>
f4(4, 2, 2)       =>
f2(4, 3, 2)       =>
f3(4, 3, 2)       =>
f4(4, 3, 6)       =>
f2(4, 4, 6)       =>
f3(4, 4, 6)       =>
f4(4, 4, 24)      =>
f2(4, 5, 24)      =>
f5(4, 5, 24)      =>
24
```

Notice that the successive rewritten expressions correspond precisely to the *states* of the factorial flowchart in execution. A term of the form

$$f_i(\textit{N-value}, \textit{K-value}, \textit{F-value})$$

corresponds to the state in which the instruction pointer is at node i in the flowchart, and the values of variables *N*, *K*, and *F* are *N-value*, *K-value*, and *F-value* respectively.

McCarthy's Transformation Principle

Now we articulate the method for deriving rules from the flowchart. This method was first presented by Professor John McCarthy, so we call it "McCarthy's Transformation Principle". We have already indicated that there is one function name for each point in the flowchart.

> *McCarthy's Transformation Principle***:**
>
> Every assignment-based program can be represented as a set of mutually-recursive functions without using assignment.

First we give the transformation rule when two points are connected by an assignment:

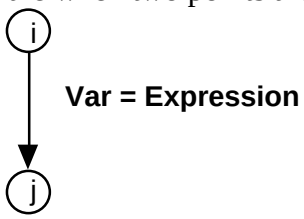


Figure 71: Assignment in the flowchart model

Corresponding to this connection we introduce a rule that rewrites f_i in terms of f_j. We identify the argument of f_i corresponding to the variable *Var* on the *lhs*. On the *rhs*, we replace that variable with *Expression*, to give the rule:

$f_i(\ldots, Var, \ldots) \Rightarrow f_j(\ldots, Expression, \ldots);$

The justification for this rule is as follows: The computation from point i will take the same course as the computation from point j would have taken with the value of *Var* changed to be the value of *Expression*.

Next we give the transformation rule when two points are connected by a test condition:

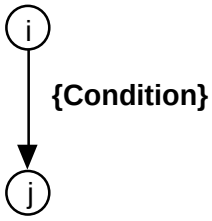


Figure 72: Conditional test in the flowchart model

In this case, the rule will be guarded, of the form

$f_i(\ldots)$ => Condition ? $f_j(\ldots)$;

The justification for this rule is: The computation from point i, in the case *Condition* is true, will take the same course as the computation from j. In most flowcharts, conditions come paired with their negations. Such a rule would usually be accompanied by a rule for the complimentary condition, recalling that ! means *not*

$f_i(\ldots)$ => !Condition ? $f_k(\ldots)$;

where f_k will usually correspond to a different point than f_j. Alternatively, we could use the sequential listing convention to represent the negation implicitly:

$f_i(\ldots)$ => Condition ? $f_j(\ldots)$;
$f_i(\ldots)$ => $f_k(\ldots)$;

Finally, to package the set of rules for external use, we introduce a rule that expresses the overall function in terms of the function at the entry point and one that expresses the function at the exit point to give the overall result. In the factorial example, these were:

```
fac(N) => f0(N, arb, arb);

f5(N, K, F) => F;
```

The rule-based model allows some other constructions not found in some flowcharts:

Parallel Assignment

With a single rule, we can represent the result of assigning to more than one variable at a time. Consider a rule such as

$f_i(\ldots, Var_1, \ldots Var_2, \ldots)$ => $f_j(\ldots, Expression_1, \ldots Expression_2, \ldots)$;

The corresponding assignment statement would evaluate both $Expression_1$ and $Expression_2$, then assign the respective values to Var_1 and Var_2. This is not generally equivalent to two assignments in sequence, since each expression could entail use of the other variable. Some languages include a parallel assignment construction to represent this concept. For two variables a parallel assignment might appear as

$(Var_1, Var_2) = (Expression_1, Expression_2)$

Combined Condition and Assignment

With a single rule, we can represent both a conditional test and an assignment. Consider a rule such as

$f_i(\ldots, Var, \ldots) \Rightarrow Condition\ ?\ f_j(\ldots, Expression, \ldots);$

Here the computation of f_i is expressed in terms of f_j with a substituted value of *Expression* for *Var* only in case that *Condition* holds This is depicted in the flowchart as:

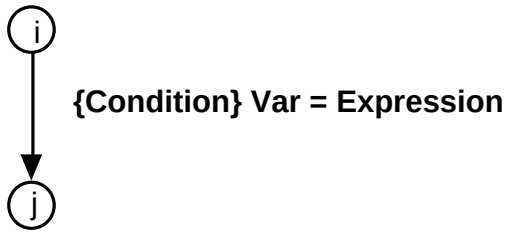


Figure 73: Combined condition and assignment in the flowchart model

List Reverse Example

One way to approach the problem of reversing a list by a recursive rule is to first give an assignment-based program for doing this, then use the McCarthy transformation. The idea of our assignment-based program is to loop repeatedly, in each iteration decomposing the remaining list to be reversed, moving the first symbol of that list to the result, then repeating on the remainder of the first list. This could be described by the following program, which is intentionally more "verbose" than necessary. An assignment of the form `[A | M] = L` decomposes the list L (assumed to be non-empty) into a first element `A` and a list of the rest of the elements `M`. This is pseudo-code, not a specific language.

```
L = list to be reversed;

R = [ ];
while( L != [ ] )
  {
  [A | M] = L;    // decompose L into first element A and rest M
  L = M;
  R = [A | R];    // compose list R
  }
assert R is reverse of original list
```

For example, here is a trace of the program through successive loop bodies:

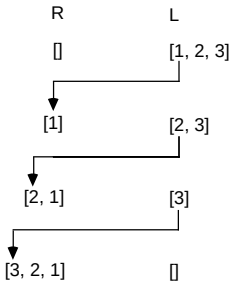


Figure 74: Flow of data in reversing a list

The flowchart version of this program could be expressed as follows:

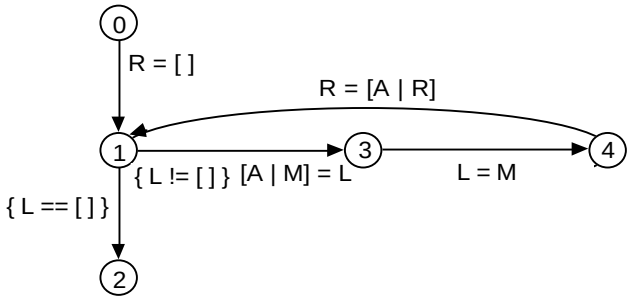


Figure 75: Flowchart for reversing a list

According to the McCarthy Transformation Principle, we can convert this to rewrite rules using the four program variables `L`, `R`, `M`, `A`.

```
f0(L) => f1(L, [ ], arb, arb);

f1(L, R, M, A) =>  L == [ ] ? f2(L, R, M, A);

f1(L, R, _, _) => L != [ ] ? L = [A | M], f3(L, R, M, A);

f3(L, R, M, A) => f4(M, R, M, A);

f4(L, R, M, A) => f1(L, [A | R], M, A);

f2(L, R, M, A) => R;
```

In the second rule for f_1,

```
f1(L, R, _, _) => L != [ ] ? L = [A | M], f3(L, R, M, A);
```

we use _ arguments to avoid confusion of the third and fourth arguments with the element `A` and list `M` decomposed from list `L`. We could simplify this further by using matching on the first argument to:

```
f1([A | M], R, _, _) => f3([A | M], R, M, A);
```

In the following section we will show why this set of rules is equivalent to:

```
f0(L) => f1(L, [ ]);

f1([ ], R) => R;

f1([A | L], R) => f1(L, [A | R]);
```

We can see that f_1 is the same function as the two-argument `reverse` function described in *Low-Level Functional Programming*.

Program Compaction Principle

The rule-based presentation gives us a way to derive more compact representations of programs. When there is a single rule for an auxiliary function name, we can often combine the effect of that rule with any use of the name and eliminate the rule itself. For example, return to our rules for factorial:

```
fac(N) => f0(N, arb, arb);

f0(N, K, F) => f1(N, K, 1);          // corresponds to assignment F = 1;

f1(N, K, F) => f2(N, 1, F);          // corresponds to assignment K = 1;

f2(N, K, F) => (K <= N) ? f3(N, K, F);

f2(N, K, F) => (K > N) ? f5(N, K, F);

f3(N, K, F) => f4(N, K, F*K);        // corresponds to assignment F = F*K;

f4(N, K, F) => f2(N, K+1, F);        // (corresponds to assignment K = K+1;

f5(N, K, F) => F;
```

We see that `f3` is immediately rewritten in terms of `f4`. Thus anywhere that `f3` occurs could be translated into a corresponding occurrence of `f4` by first making a corresponding substitution of expressions for variables. In our rules, `f3` occurs only once, in the second rule for `f2`. We can substitute the *rhs* of the `f3` rule in the second rule for `f2` to get a replacement for the latter:

```
f2(N, K, F) => (K <= N) ? f4(N, K, F*K);
```

We also see that `f4` is immediately rewritten in terms of `f2`, so the rule could be further changed to:

```
f2(N, K, F) => (K <= N) ? f2(N, K+1, F*K);
```

We can similarly re-express `f0` in terms of `f2`, as follows:

```
f0(N, K, F) => f1(N, K, 1);     // original
f0(N, K, F) => f2(N, 1, 1);     // using rule f1(N, K, F) => f2(N, 1, F);
```

Finally, we can get rid of `f5` since `f5(N, K, F)` is immediately rewritten as `F`

```
f2(N, K, F) => (K > N) ? F;
```

and we can re-express `fac` directly in terms of `f2`. The resulting set of rules is:

```
fac(N) => f2(N, 1, 1);
f2(N, K, F) => (K <= N) ? f2(N, K+1, F*K);
f2(N, K, F) => (K > N) ? F;
```

This is obviously much more compact than the original, since it contains only one auxiliary function, `f2`. The corresponding compacted flowchart, using combined conditions and assignment and parallel assignment, could be shown as follows:

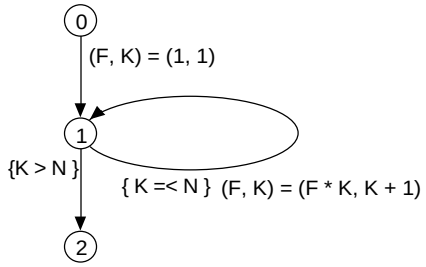


Figure 76: A compact factorial flowchart

List Reverse Compaction Example

Compact the simplified rules for the reverse program:

```
f0(L) => f1(L, [ ], arb, arb);
f1(L, R, M, A) =>  L == [ ] ? f2(L, R, M, A);
f1([A | M], R, _, _) => f3([A | M], R, M, A);
f3(L, R, M, A) => f4(M, R, M, A);
f4(L, R, M, A) => f1(L, [A | R], M, A);
f2(L, R, M, A) => R;
```

We can get rid of `f4` by compaction, giving us:

```
f3(L, R, M, A) => f1(M, [A | R], M, A);
```

We can get rid of `f3` by compaction, giving us:

```
f1([A | M], R, _, _) =>f1(M, [A | R], M, A);
```

We can get rid of `f2` by compaction, and move the guard into the rule, giving

```
f1([], R, M, A) =>  R;
```

The resulting system is:

```
f0(L) => f1(L, [ ], arb, arb);

f1([], R, M, A) =>  R;

f1([A | M], R, _, _) =>f1(M, [A | R], M, A);
```

We notice that the third and fourth arguments of `f1` after compaction never get used, so we simplify to:

```
f0(L) => f1(L, [ ]);

f1([], R) =>  R;

f1([A | M], R) =>f1(M, [A | R]);
```

Now note that `f1` is our earlier 2-argument `reverse` by a different name. So McCarthy's transformation in this case took us to an efficient functional program.

Exercises

1 •• Consider the following Java program fragment for computing `k` to the `n`th power, giving the result in `p`. Give a corresponding set of rules in uncompacted and compacted form.

```
p = 1;
c = 1;
while( c <= n )
  {
  p = p * k;
  c++;
  }
```

2 •• The Fibonacci function can be presented by the following rules:

```
fib(0) => 1;

fib(1) => 1;
```

```
fib(N) => fib(N-1) + fib(N-2);
```

However, using straight rewriting on these rules entails a lot of recomputation. [See for yourself by computing fib(5) using the rules.] We mentioned this earlier in a discussion of "caching". Give an assignment-based program that computes fib(N) "bottom up", then translate it to a corresponding set of rules for more efficient rewrite computation. [Hint: fib(N) is the Nth number in the series 1, 1, 2, 3, 5, 8, 13, 21, … wherein each number after the second is obtained by adding together the two preceding numbers.] Then give an equivalent set of rules in compacted form.

The use of a bottom-up computation to represent a computation that is most naturally expressed top-down and recursively is sometimes called the **"dynamic programming principle"**, following the term introduced by Richard Bellman, who explored this concept in various optimization problems.

3 ••• Try to devise a set of rules for computing the integer part of the square root of a number, using only addition and <= comparison, not multiplication or division. If you have trouble, try constructing an assignment-based solution first, then translating it. [Hint: Each square can be represented as the sum of consecutive odd integers. For example, 25 = 1 + 3 + 5 + 7 + 9.]

6.5 Turing Machines – Simple Rewriting Systems

We conclude this chapter with a different example that can be represented by rewriting rules, one of major interest in theoretical computer science. The main definitions, however, should be part of the cultural knowledge of every scientist, since they relate to certain problems having an unsolvable character, about which we shall say more in *Limitations of Computing*.

Turing machines **(TMs)** are named after the British mathematician Alan M. Turing, 1912-1954†, who first proposed them as a model for universal computability. By "universal" we mean a model general enough to enable the representation of any computable function. Although they share this property with the general recursive functions, Turing machines are much more basic. For example, they do not assume a general matching capability. For that matter, they do not assume anything about numbers or arithmetic. Instead, everything, including matching, is done in terms of a finite set of symbols. Any numeric interpretation of those symbols is up to the user.

† For a biography, see the book by Hodges, *Alan Turing: The Enigma*. The title is a pun. Not only is Turing regarded as enigmatic, but also one of his principal contributions was a machine which helped break codes of the German encoding machine known as the Enigma during World War II.

Each Turing machine has a fixed or "wired-in" program, although through an encoding technique, it is possible to have a single machine that mimics the behavior of any other TM. The components of a TM are:

Finite-state controller (sequential circuit)

Movable read/write head

Unbounded storage tape:

On each tape cell, one symbol from a **fixed finite alphabet** can be written. One symbol in the alphabet is distinguished as "**blank**". The distinction of blank is that, in any given state, at most a finite set of the cells will contain other than blank.

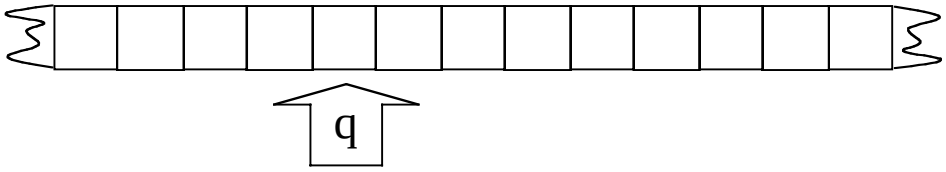


Figure 77: Depiction of a TM with controller in state q.

Since only a finite set of cells can be non-blank at any time, beyond a certain cell on either side of the tape, the cells contain only blanks. However, this boundary will generally change, and the non-blank area may widen.

The controller, in a given state (which includes the controller state and the tape state), will:

- read the symbol under the head
- write a new symbol (possibly the same as the one read)
- change state (possibly to the same state)
- move one cell left or right, or stay put

Program or Transition Function for a Turing Machine

A description of, or program for, a TM is a partial function, called the **transition function**:

States x Symbols → Symbols x Motions x States

where Motions = {left, right, none}

The symbol x here denotes the Cartesian product of sets.

In the present case, we are saying that the transition function of a TM is a partial function having a domain consisting of all pairs of the form

(State, Symbol)

and a co-domain consisting of all 3-tuples (triples) of the form

(Symbol, Motion, States)

Such a program could be given by a *state transition table*, with each row of the table listing one combination of five items (i.e. a 5-tuple or quintuple)

State, *Read*, *Written*, *Motion*, *NewState*

Such a 5-tuple means the following:

> If the machine's control is in *State* and the head is over symbol *Read*, then machine will write symbol *Written* over the current symbol, move the head as specified by *Motion*, and the next state of the controller will be *NewState*.

If no transition is specified for a given state/symbol pair, then the machine is said to *halt* in that state. The requirement that the program be a *partial function* is equivalent to saying the following:

> There is at most one row of the table corresponding to any given combination (*State*, *Read*).

The Partial Function Computed by a Turing Machine

The partial function computed by a TM is the tape transformation that takes place when the machine starts in a given initial state and eventually halts. Alternatively, the *partial function computed* by a TM is the transformation from initial tape to one of a finite set of halting combinations (state-symbol pairs). If the machine does not halt on a given input, then we say the partial function is *undefined* for this input.

Turing Machine Addition Example

We show the construction of a machine that adds 1 to its binary input numeral. It is assumed that the machine starts with the head to the immediate right of the least significant bit, which is on the right. The alphabet of symbols on the tape will be {0, 1, _}, where _ represents a blank cell.

```
_ _ _ _ _ 1 0 1 1 0 1 1 1 1 1 _ _ _ _ _ _
          ^
```

The state transitions for such a machine could be specified as:

State	Read	Written	Motion	NewState
start	_	_	left	add1
add1	0	1	right	end
add1	_	1	right	end
add1	1	0	left	add1
end	0	0	right	end
end	1	1	right	end

Below we give a trace of this Turing machine's action when started on a binary number 100111 representing decimal 39. The result is 101000, representing decimal 40. The first set of square brackets are around the symbol under the tape head. The second set of brackets shows the control state.

```
1 0 0 1 1 1 [_]  [start]
1 0 0 1 1 [1] _  [add1]
1 0 0 1 [1] 0 _  [add1]
1 0 0 [1] 0 0 _  [add1]
1 0 [0] 0 0 0 _  [add1]
1 0 1 [0] 0 0 _  [end]
1 0 1 0 [0] 0 _  [end]
1 0 1 0 0 [0] _  [end]
1 0 1 0 0 0 [_]  [end]
```

Note the importance of every Turing machine rule-set to be finite. Here we have an example of an infinite-state system with a finite representation.

The *partial function computed* by this machine can be thought of as a representation of the function f where f(x) = x+1, at least as long as the tape is *interpreted* as containing a binary-encoded number.

Universal Turing Machines

A universal Turing Machine (UTM) is a Turing Machine that, given an encoding of *any* Turing machine (into a fixed size alphabet) and an encoding of a tape for that machine, both on the tape of the UTM, simulates the behavior of the encoded machine and tape, transforming the tape as the encoded machine would and halting if, and only if, the encoded machine halts.

The set of all TMs has no bound to the number of symbols used as names of control states nor of tape symbols. However, we can encode an infinity of such symbols using only two symbols as follows: suppose that we name the control states q_0, q_1, q_2, … without loss of generality. Then we could use something like a series of i 1's to represent q_i. However, we need more than this simplistic encoding, since we will need to be able to mark certain of these symbols in the process of simulating the machine being presented. We will not go into detail here.

Exercises

1 •• Develop the rules for a Turing machine that erases its input, assuming that all of the input occurs between two blank cells and the head is initially positioned in this region.

2 •• Develop the rules for a Turing machine that detects whether its input is a palindrome (reads the same from left-to-right as from right-to-left). Assume the input string is a series of 0's and 1's, with blank cells on either end.

3 ••• Develop the rules for a Turing machine that detects whether one input string (called the "search string") is contained within a second input string. The two strings are separated by a single blank and the head is initially positioned at the right end of the search string.

4 ••• Develop a Turing machine that determines if one number is evenly divisible by a second. Assume the numbers are represented as a series of 1's (i.e. in 1-adic notation). The two numbers are separated by a single blank. Assume the head starts at the right end of the divisor.

5 •••• Develop a Turing machine that determines if a number is prime. Assume the number is represented in 1-adic notation. Use the machine in the previous problem as a subroutine.

6 ••• Present an argument that demonstrates that the composition of two functions that are computable by a Turing machine must itself be computable by a Turing machine. Assume that each function takes one argument string.

7 ••••• Develop the rules for a universal Turing machine.

6.6 Turing's Thesis

Turing's thesis, also called *Church's Thesis,* the *Church/Turing Thesis,* the *Turing Hypothesis, etc.,* is the following important assertion.

Every function that is intuitively computable is computable by some TM.

The reason for the assertion's importance is that it is one way of establishing a link between a formal rule-based computation model and what is intuitively computable. That is, we can describe a process in informal terms and as long the description is reasonably clear as a computational method, we know that it can be expressed more formally without actually doing so.

Turing's argument in support of this thesis follows†. Note that “computer” should be read as “person doing computation”. The relation to a machine comes later in the passage.

> “Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child’s arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.
>
> The behavior of the computer at any moment is determined by the symbols which he is observing, and his “state of mind” at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be ‘arbitrarily close’ and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.
>
> Let us imagine the operations performed by the computer to be split up into “simple operations” which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the

† from Turing's 1937 paper. I have made minor substitutions in extracting this selection from the context of the paper.

same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always 'observed' squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognizable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed square does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

In connection with ‘immediate recognizability’, it may be thought that there are other kinds of squares which are immediately recognizable. In particular, squares marked by special symbols might be taken as immediately recognizable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognize a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find ‘… hence (applying Theorem 157767733443477) we have …'. In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other ‘immediately recognizable’ squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable.

The simple operations must therefore include:

(a) Changes of the symbol on one of the observed squares.

(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(A) A possible change (a) of symbol together with a possible change of state of mind.

(B) A possible change (b) of observed squares, together with a possible change of state of mind.

The operation actually performed is determined, as has been suggested above, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an internal state of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares. The move which is done, and the succeeding configuration, are determined by the scanned symbol and the internal state. The machines as defined here can be constructed to compute the same sequence computed by the computer."

Our definition is the now-customary one with B = 1 and L = 1. That is, there is one scanned square and the head changes position by at most one in a single move.

Evidence in Support of Turing's Thesis

Turing's thesis has withstood the test of time and many examinations. It is generally accepted as true, even though it can't ever be proven. To do so would require a formalization of an informal concept, computability. Then a similar argument would have to be presented to argue the correctness of this new definition, and so forth. The evidence for its validity include:

- Turing's argument itself
- No counter-examples to the thesis have ever been found, despite considerable effort.
- Many other models were developed independently, some of which were argued similarly to characterize computability, then later shown to be equivalent to the Turing model.

Some of these models are enumerated below:

Specific universal computing models

Each of these models is known to have computational power equivalent to the Turing machine model:

Turing machines (Turing)

We have discussed this model above.

general recursive functions (Kleene)

This model is very similar to the rex rewriting system, except that it concerned itself only with natural numbers, not lists.

Partial Recursive Functions (Goedel, Kleene)

This model is more rigidly structured than the general recursive function model. It also deals with computing on the natural numbers. The set of functions is defined using induction, a principle to be discussed in a subsequent section. Even though the defining scheme is a little different, the set of functions defined can be shown to be the same as the general recursive functions.

register machines (Shepherdson and Sturgis)

A register machine is a simple computer with a fixed number of registers, each of which can hold an arbitrary natural number. The program for a register machine is a sequence of primitive operations specifying incrementation and decrementation of a register and conditionally branching on whether a register is 0.

lambda-calculus (Church)

The lambda calculus is a calculus of functional forms based on Church's lambda notation. This was introduced earlier when we were discussing functions that can return functions as values.

phrase-structure grammars (Chomsky)

This model will be discussed in Computing Grammatically.

Markov algorithms (Markov)

Markov algorithms are simple rewriting systems similar to phrase-structure grammars.

tag systems (Post)

Tag systems are a type of rewriting system. They are equivalent to a kind of Turing machine with separate read- and write- heads that move in one direction only, with the tape in between. The name of the system apparently derives from using the model to analyze questions of whether the read-head ever "tags" (catches up with) the write-head.

Practical uses of Turing's Thesis

The "Turing Tarpit"

Suppose that you have invented a new computing model or programming language. One test that is usually applied to such a model is whether or not every computable partial function can be expressed in the model. One way to establish this is to show that any Turing machine can be simulated within the model. Henry Ledgard calls this scheme the "Turing Tarpit".

This principle relies on the acceptance of the Turing machine notion being universally powerful for computability. For most programming languages, another model, known as a **register machine**, a machine with two registers, each of which holds an arbitrary natural number, is simpler to simulate. In turn, a register machine can simulate a Turing machine. We indicate how this is done in a subsequent section.

The Informal-Description Principle

This principle relies on an informal acceptance of Turing's argument. In order to show a certain function is computable by a Turing machine, it suffices to give a verbal description of a computational process, even without the use of a formal model. If challenged, the purveyor of the argument can usually describe in sufficient detail how to carry out the computation on a Turing machine, although such details can quickly reach a laborious level. For many examples of the use of this principle, see a book such as that of Rogers, 1967.

6.7 Expressing Turing Machines as Rewrite Rules

Given two models that are both claimed to be universal with respect to computability, it is important to be able to express every partial function expressible in one model in terms of the other model and vice-versa. Here we show how Turing machine computations can be expressed in terms of rex rules. We generate rules that manipulate lists, and leave to the reader the small remaining task of showing that the lists can be encoded as numbers. In other words, we show in this section:

Every Turing-computable partial function is a general recursive function.

It suffices to give a method for creating rex rules from Turing machine rules. Thus, at the same time, we are using the Turing Tarpit principle to show:

rex is a universal computing language.

We must first show how the state of a Turing machine will be encoded. Recall that the complete state of a Turing machine consists of a control state, a tape, and a head position. We represent the complete state by the following four components:

- The control state
- A list of symbols to the left of the head, reading left-to-right.
- A list of symbols to the right of the head, reading right-to-left.
- The symbol under the head.

In particular, the list of symbols to the left of the head reads in the opposite direction from the list to the right of the head. The reason for this is so that we can use the list-constructing facilities of rex on the symbols around the head.

Recall that a Turing machine is specifiable by a set of 5-tuples:

State, *Read*, *Written*, *Motion*, *NewState*

where *State* and *NewState* refer to control states. For each 5-tuple, there will be one rex rule. The structure of the rule depends on whether *Motion* is left, right, or none. We describe each of these cases. Note that in the rewrite rules, the variables *State*, *Read*, *Written*, *Motion*, and *NewState* will be replaced by literal symbols according to the TM rules, whereas the variables *Left*, *Right*, *FirstLeft*, *FirstRight* will be left as rex variables. The partial function `tm` defined below mimics the state-transitions of the machine.

In the case that *Motion* is *none*:

```
          tm(State, Left, Read, Right)
      => tm(NewState, Left, Written, Right);
```

In the case that *Motion* is *right*:

```
            tm(State, Left, Read, [FirstRight | RestRight])
      => tm(NewState, [Written | Left], FirstRight, RestRight);
```

In the case that *Motion* is *left*:

```
          tm(State, [FirstLeft | RestLeft], Read, Right)
      => tm(NewState, RestLeft, FirstLeft, [Written | Right]);
```

Next, we need one pair of rules to handle the case where the head tries to move beyond the symbols at either end of the tape. In either case, we introduce a blank symbol into the tape. We will use ‘ ‘ to designate the blank symbol.

```
tm(State, [ ], Read, Right) => tm(State, [‘ ‘], Read, Right);

tm(State, Left, Read, [ ]) => tm(State, Left, Read, [‘ ‘]);
```

Finally, we need to provide a rule that will give an answer when the machine reaches a halting state. Recall that these are identified by combinations of states and control symbols for which no transition is specified. We want a rule that will return the tape contents for such a configuration. In order to this, we need to reverse the list representing the left tape and append it to the list representing the right. The two-argument `reverse` function, which appends the second argument to the reverse of the first, is ideally suited for this purpose:

For each *halting* combination: *State*, *Read*, include a rule:

```
tm(State, Left, Read, Right) => reverse(Left, [Read | Right]);
```

where, as before,

```
reverse([ ], M) => M;

reverse([A | L], M) => reverse(L, [A | M]);
```

Turing Machine Example in rex

We give the rex rules for the machine presented earlier that adds 1 to its binary input numeral. We show the original TM rules on the lines within /* … */ and follow each with the corresponding rex rule. Here we use {‘0‘,‘1‘, ' '} for tape symbols (single quotes may be used for single characters) and {“start”, “add1”, “end”} for control states.

```
/* start        ' '       ' '       left    add1 */

tm(“start”, [FirstLeft | RestLeft], ' ', Right)
=> tm(“add1”, RestLeft, FirstLeft, [' ' | Right]);


/* add1         0       1       right   end */

tm(“add1”, Left, ‘0‘, [FirstRight | RestRight])
=> tm(“end”, [‘1‘ | Left], FirstRight, RestRight);


/* add1         ' '       1       right   end */

tm(“add1”, Left, ' ', [FirstRight | RestRight])
=> tm(“end”, [‘1‘ | Left], FirstRight, RestRight);


/* add1         1       0       left    add1 */
```

```
tm(“add1”, [FirstLeft | RestLeft], ‘1‘, Right)
=> tm(“add1”, RestLeft, FirstLeft, [‘0‘ | Right]);


/* end  0       0       right   end */

tm(“end”, Left, ‘0‘, [FirstRight | RestRight])
=> tm(“end”, [‘0‘ | Left], FirstRight, RestRight);


/* end  1       1       right   end */

tm(“end”, Left, ‘1‘, [FirstRight | RestRight])
=> tm(“end”, [‘1‘ | Left], FirstRight, RestRight);
```

As specified in the general scheme, we also add the following rules:

```
tm(end, Left, ' ', Right) => reverse(Left, [' ' | Right]);

reverse([ ], M) => M;

reverse([A | L], M) => reverse(L, [A | M]);
```

The following is a trace of the rewrites made by rex on the input tape `100111` representing decimal 39. As discussed in the text, the left list is the reverse of the tape, so the corresponding initial argument is `[‘1’, ‘1’, ‘1’, ‘0’, ‘0’, ‘1’]`.

```
tm(“start”,[‘1’,’1’,’1’,’0’,’0’,’1’], ' ', [ ])    =>
tm(“start”,[‘1’,’1’,’1’,’0’,’0’,’1’], ' ', [' ']) =>
tm(add1,   [‘1’,’1’,’0’,’0’,’1’], ‘1’, [' ',' ']) =>
tm(add1,   [‘1’,’0’,’0’,’1’], ’1’, [‘0’,' ',' ']) =>
tm(add1,   [‘0’,’0’,’1’], ’1’, [‘0’,’0’,' ',' ']) =>
tm(add1,   [‘0’,’1’], ’0’, [‘0’,’0’,’0’,' ',' ']) =>
tm(“end”,[‘1’,’0’,’1’], ’0’, [‘0’,’0’,' ',' '])  =>
tm(“end”,[‘0’,’1’,’0’,’1’], ’0’, [‘0’,' ',' '])  =>
tm(“end”,[‘0’,’0’,’1’,’0’,’1’], ’0’, [' ',' '])  =>
tm(“end”,[‘0’,’0’,’0’,’1’,’0’,’1’], ' ', [' '])  =>
reverse([‘0’,’0’,’0’,’1’,’0’,’1’], [' ',' '])    =>
[‘1’,’0’,’1’,’0’,’0’,’0’,' ',' ']
```

The result `101000` is indeed the representation of the number that would be 40 decimal.

Exercises

1 ••• Show that we can directly convert the function constructed above into a numeric one. [Hint: If the TM has an N-symbol tape alphabet, then lists can be viewed simply as N-adic numerals. Show how to compute, as general recursive functions, the *numeric* functions that extract the first symbol and the remaining symbols from a list.] Then show how each list rewriting rule can be recast as a numeric rewriting rule.

2 ••••• Show that every general recursive function is computable by a Turing machine**.**

3 ••• The following is a list of possible ideas for extending the "power" of the Turing machine notion. Give constructions that show that each can be reduced to the basic Turing machine definition as presented here.

1. Multiple heads on one tape. The transitions generally depend on the symbol under each tape head, and can write on all heads in one move. [Show that this model can be simulated by a single head machine by using "markers" to simulate the head positions.]

2. Multiple tapes, each with its own head.

3. Two dimensional tape. Head can move left, right, up, or down.

4. An infinite set of one-dimensional tapes, with the head being able to alternately select the next or previous tape in the series.

5. N-dimensional tape for N > 2.

6. Adding multiple registers to the control of the machine.

7. Adding some number of "counters", each of which can hold any natural number.

6.8 Chapter Review

• Define the following terms:

Cartesian product
deterministic
dynamic programming
McCarthy's transformation
parallel assignment
reachability
state
transition
transitive closure
Turing machine
Turing's thesis

• Demonstrate the application of McCarthy's transformation principle.

• Demonstrate how to convert a Turing machine program to a rex program.

6.9 Further Reading

•• Jon Barwise and John Etchemendy, *Turing's World 3.0*, CSLI Publications, Stanford California, 1993. [Includes Macintosh disk with visual Turing machine simulator.]

••• Alonzo Church, *The Calculi of Lambda Conversion*, Annals of Mathematical Studies, 6, Princeton University Press, Princeton, 1941. [Introduces the lambda calculus as a universal computing model. Expresses "Church's thesis".]

•• W.F. Clocksin and C. S. Mellish, *Programming in Prolog*, Third edition, Springer-Verlag, Berlin, 1987. [A readable introduction to Prolog.]

••• Rolf Herken (ed.), *The Universal Turing Machine, A Half-Century Survey*, Oxford University Press, Oxford, 1988. [A survey of some of the uses of Turing machines and related models.]

• A. Hodges, *Alan Turing: The Enigma*. Simon & Schuster, New York, 1983. [Turing's biography.]

•• R.M. Keller. *Formal verification of parallel programs*. Communications of the ACM, **19**, 7, 371-384 (July 1976). [Introduces the transition-system concept and applies it to parallel computing.]

••• S.C. Kleene, *Introduction to Metamathematics*, Van Nostrand, Princeton, 1952. [Introduces general recursive functions and related models, including partial recursive functions and Turing machines.]

••• Benoit Mandelbrot. *The Fractal Geometry of Nature*, W.H. Freeman and Co., San Francisco, 1982. [Discusses initial explorations of the idea of fractals.]

•• Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974. [Examples using McCarthy's transformation and other models.]

•• J. McCarthy, *Towards a mathematical science of computation*, in C.M. Popplewell (ed.), Information Processing, Proceedings of IFIP Congress '62, pp. 21-28, North-Holland, Amsterdam, 1962. [The original presentation of "McCarthy's Transformation.]

•• Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, New Jersey, 1967. [Discusses Turing machines, including universal Turing machines, and other computational models, such as register machines.]

•••• H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967. [More abstract and advanced treatment of partial recursive functions. Employs the informal description principle quite liberally.]

•• J.C. Shepherdson and H.E. Sturgis, *Computability of recursive functions*, Journal of the ACM, **10**, 217-255 (1963).

••• A.M. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Mathematical Society, ser. 2, vol. 42, pp. 230-265 (1936-37), corrections, ibid., vol. 43, pp. 544-546 (1937). [The presentation of Turing's thesis. Moderate to difficult.]

7. Object-Oriented Programming

7.1 Introduction

This chapter describes object-oriented computing and its use in data abstraction, particularly as it is understood in the Java™ language, including various forms of inheritance.

The late 1980's saw a major paradigm shift in the computing industry toward "object-oriented programming". The reason behind the surge in popularity of this paradigm is its relevance to organizing the design and construction of large software systems, its ability to support user-defined data abstractions, and the ability to provide a facility for reuse of program code. These are aside from the fact that many of the entities that have to be dealt with in a computer system are naturally modeled as "objects". Ubiquitous examples of objects are the images and windows that appear on the screen of a typical personal computer or workstation environment. These have the characteristics that they (i) maintain their own state information; (ii) can be created dynamically by the program; (iii) need to interact with other objects as a manner of course.

The paradigm shift is suggested by comparing the two cases in the following figures. In the first figure, we have the conventional view of data processing:

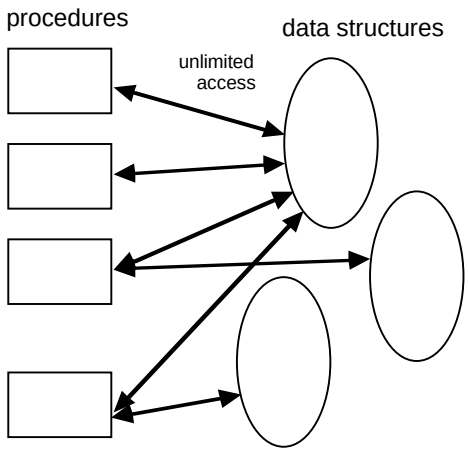


Figure 78: Conventional (pre-object-oriented) paradigm

Data structures are considered to be separate from the procedures. This introduces management problems of how to ensure that the data are operated on only by appropriate procedures, and indeed, problems of how to define what *appropriate* means for particular data. In the second figure, many of the procedures have been folded inside the data, the result being called "objects".

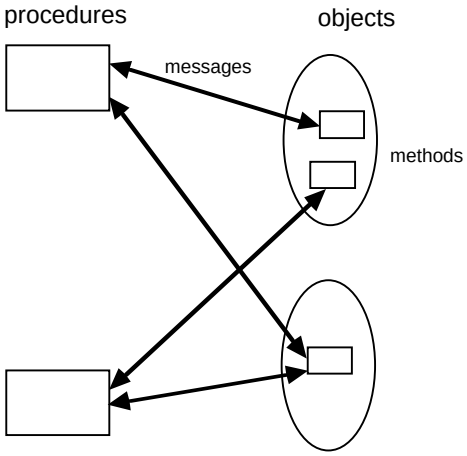


Figure 79: Object-oriented paradigm

Thus each object can be accessed only through its accompanying procedures (called ***methods***). Sometimes the access is referred to as "sending a message" and "receiving a reply" rather than calling a procedure, and sometimes it is implemented quite differently.

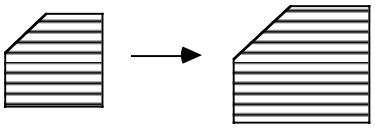


Figure 80: State transition in a graphical object as the result of sending it a resize message

In any case, an enforced connection between procedures and data solves the issue of what procedures are appropriate to what data and the issue of controlling access to the data. Languages differ on how much can be object-oriented vs. conventional: In Smalltalk, "everything is an object", whereas in Java and C++, primitive data types such as integers are not.

7.2 Classes

The concept of **object** relates to both data abstraction and to procedural abstraction. An object is a data abstraction in that it contains data elements that retain their values or state between references to the object. An object is a *procedural* abstraction, in that the principal means of getting information from it, or of changing its state, is through the invocation of procedures. Rather than attempting to access an object through arbitrary procedures, however, the procedures that access the object are associated directly with

the object, or more precisely, with a natural grouping of the objects known as their **class**. In many languages, the syntactic declaration of a class is the central focus of object definition. The class provides the specification of how objects behave and the language permits arbitrarily-many objects to be created from the class mold.

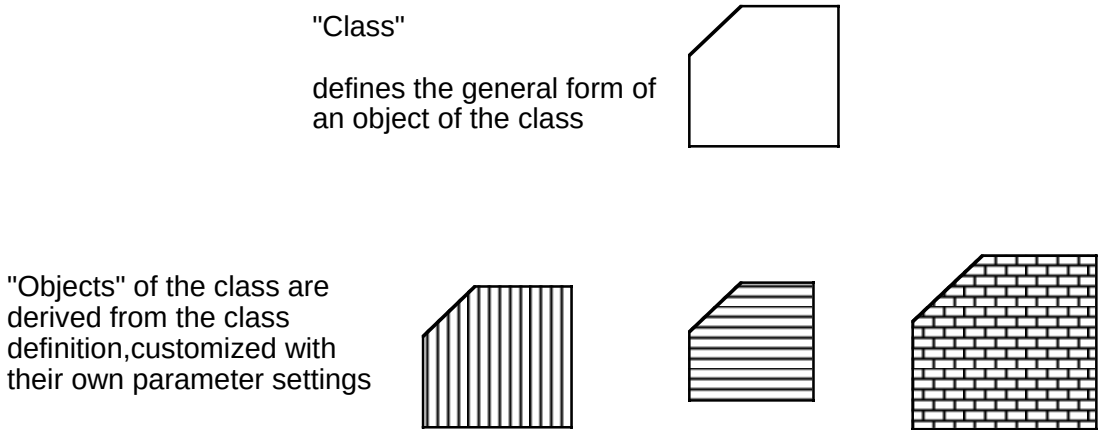


Figure 81: Class vs. Object

7.3 Attributes

Some of the types of information kept in objects may be thought of as *attributes* of the object. Each attribute typically has a value from a set associated with the attribute. Examples of attributes and possible value sets include:

size	{small, medium, large, ...}
shape	{polygonal, elliptical, ...}
color	{red, blue, green, ...}
border	{none, thin, thick, ...}
fill	{vertical, horizontal, diagonal, brick, ...}

Objects are the principal vehicles for providing data abstractions in Java: Each object can contain data values, such as those of attributes, that define its state. An object may also provide access to those values and the provide ability to change them. These things are preferably done by the methods associated with the object, rather than through direct access to the state values themselves, although Java does not prevent the latter type of access. By accessing attribute values only through methods, the representation of the state of the object can be changed while leaving the procedural interface intact. There are numerous benefits of providing a methods-only barrier between the object and its users or **clients**:

- **Principle of Modularity** ("separation of concerns"): This principle asserts that it is easier to develop complex programs if we have techniques for separating the functionality into pieces of more primitive functionality. Objects provide one way of achieving this separation.

- **Evolutionary Development**: The process of developing a program might begin with simple implementations of objects for testing purposes, then evolve to more efficient or robust implementations concurrently with testing and further development.

- **Flexibility in Data Representations**: An object might admit several different data representations, which could change in the interest of efficiency while the program is running. The object notion provides a uniform means of providing access to such changing objects.

If, on the other hand, the client were permitted direct access to the attributes of an object without using methods, the representation could never be changed without invalidating the code that accessed those attributes.

The simplest type of method for setting the value of an attribute is called a *setter*, while the simplest type for getting the value of an attribute is called a *getter*. For example, if we had some kind of shape class, with fill represented by an int, we would expect to see within our class declaration method headers as follows:

```
setFill(int Fill)

int getFill()
```

7.4 Object Allocation

Objects in Java are always dynamically *allocated* (created). It is also possible for the object to reallocate dynamically memory used for its own variables. The origin of the term "class" is to think of a collection of objects with related behaviors as a *class*, a mathematical notion similar to a *set*, of those objects. Rather than defining the behavior for each object individually, a class definition gives the behaviors that will be possessed by *all* objects in the class. The objects themselves are sometimes called *members* of the class, again alluding to the set-like qualities of a class. It is also sometimes said that a particular object is an *instance* of the class.

Using Java as an example language, these are the aspects of objects, as defined by a common class declaration, that will be of interest:

Name Each class has a name, a Java identifier. The name effectively becomes a **type name**, so is usable anywhere a type would be usable.

Constructor The *constructor* identifies the parameters and code for initializing an object. Syntactically it looks like a procedure and uses the name of the class as the constructor name. The constructor is called when an object is created dynamically or automatically. The constructor

does not return a value in its syntactic structure. A constructor is always called by using the Java `new` operator. The result of this operator is a *reference* to the object of the class thus created.

Methods Methods are like procedures that provide the interface between the object and the program using the object. As with ordinary procedures, each method has an explicit return type, or specifies the return type **void**.

Variables Each time an object is created as a "member" of a class, the system allocates a separate set of variables internal to it. These are accessible to each of the methods associated with the object without explicit declaration inside the method. **That is, the variables local to the object appear as if global to the methods, without necessitating re-declaration.**

The reason for the emphasis above is that use of objects can provide a convenience when the number of variables would otherwise become too large to be treated as procedure parameters and use of global variables might be a temptation.

7.5 Static vs. Instance Variables

An exception to having a separate copy of a variable for each object occurs with the concept of **static variable**. When a variable in a class is declared `static`, there is only *one* copy shared by all objects in the class. For example, a static variable could keep a *count* of the number of objects allocated in the class, if that were desired. For that matter, a static variable could maintain a list of references to all objects created within the class.

When it is necessary to distinguish a variable from a static variable, the term **instance variable** is often used, in accordance with the variable being associated with a particular object instance. Sometimes static variables are called **class variables**.

Similarly, a **static method** is one that does not depend on instance variables, and thus not on the state of the object. A static method may depend on static variables, however. Static methods are thus analogous to procedures, or possibly functions, in ordinary languages.

A final note on classes concerns a thread in algorithm development. It has become common to present algorithms using **abstract data types** (**ADT**s), which are viewed as mathematical structures accompanied by procedures and functions for dealing expressly with these structures. For example, a typical ADT might be a *set*, accompanied by functions for constructing sets, testing membership, forming unions, etc. Such structures are characterized by the behavioral interaction of these functions, rather than by the internal representation used to implement the structure.

> Classes seem to be a very appropriate tool for defining ADTs and enforcing the disciplined use of their functions and procedures.

7.6 Example – A Stack Class

We now illustrate these points on a specific ADT or class, a class `Stack`. A stack is simply an ordered sequence of items with a particular discipline for accessing the items:

> The order in which items in a **stack** are removed is the reverse from the order in which they were entered.
>
> This is sometimes called the LIFO (last-in, first-out) property.

Regardless of how the stack is implemented, into the stack is not part of the discipline. The `Stack` class will be specified in Java:

```
class Stack
{

// .... all variables, constructors, and methods
//      used for a Stack are declared here ....

}
```

We postulate methods `push`, `pop`, `and empty` for putting items into the stack, removing them, and for testing whether the stack is empty. Let's suppose that the items are integers, for concreteness.

```
class Stack
{
void push(int x)
  {
  // .... defines how to push x ....
  }

int pop()
  {
  // .... defines how to return top of the stack ....
  return .... value returned ....;
  }

boolean isEmpty()
  {
  // .... defines how to determine emptiness ....
  return ....;
  }
}
```

Method `push` does not return any value, hence the `void`. Method `pop` does not take any argument. Method `empty` returns a boolean (`true` or `false`) value indicating emptiness:

Java allows inclusion of a static method called main with each class. For one class, this will serve as the main program that is called at the outset of execution. For any class, the main method may be used as a test program, which is what we will do here.

```
class Stack
{
// .... other methods defined here

public static void main(String arg[])
  {
  int limit = new Integer(arg[0]).intValue();
  Stack s = new Stack(limit);
  for( int i = 0; i < limit; i++ )
    {
    s.push(i);
    }
  while( !s.isEmpty() )
    {
    System.out.println(s.pop());
    }
  }
}
```

The keyword `static` defines main as a static method. The keyword `public` means that it can be called externally. The argument type `String .... []` designates an array of strings indicating what is typed on the command line; each string separated by space will be a separate element of the array. The line

```
int limit = new Integer(arg[0]).intValue();
```

converts the first command-line argument to an `int` and assigns it to the variable `limit`. The line

```
Stack s = new Stack(limit);
```

uses the Stack constructor, which we have not yet defined, to create the stack. Our first approach will be to use an array for the stack, and the argument to the constructor will say how large to make that array, at least initially.

A second use of limit in the test program is to provide a number of times that information is pushed onto the stack. We see this number being used to control the first `for` loop. The actual pushing is done with

```
s.push(i);
```

Here `s` is the focal stack object, `push` is the method being called, and `i` is the actual argument.

The while loop is used to pop the elements from the stack as long as the stack is not empty. Since by definition of the stack discipline items are popped in reverse order from the order in which they are pushed, the output of this program, if called with a command-line argument of 5, should be

```
4
3
2
1
0
```

indicating the five argument values 0 through 4 that are pushed in the `for` loop.

7.7 Stack Implementation

Now we devote our attention to *implementing* the stack. Having decided to use an array to hold the integers, we will need some way of indicating how many integers are on the stack at a given time. This will be done with a variable `number`. The value of `number-1` then gives the index within the array of the last item pushed on the stack. Likewise, the value of `number` indicates the first available position onto which the next integer will be pushed, if one is indeed pushed before the next pop. The figure below shows the general idea.

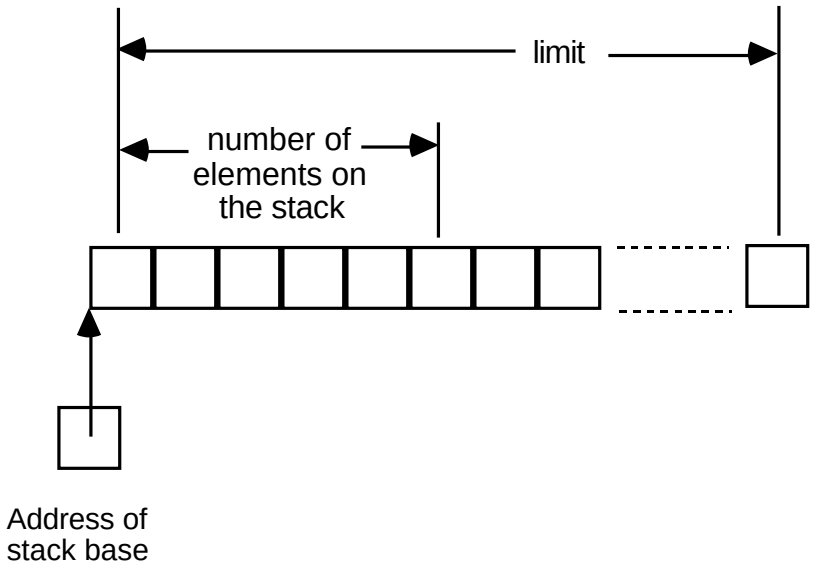


Figure 82: Array implementation of a stack

Here is how our `Stack` class definition now looks, after defining the variables `array` and `number` and adding the constructor `Stack`:

```
class Stack
{
int number;               // number of items in stack
int array[ ];             // stack contents

Stack(int limit)
  {
  array = new int[limit]; // create array
  number = 0;             // stack contains no items yet
  }
....
}
```

Note that the `new` operator is used to create an array inside the constructor. The array as declared is only a type declaration; there is no actual space allocated to it until the `new` operator is used. Although an array is technically not an object, arrays have a behavior that is very much object-like. If the `new` were not used, then any attempt to use the array would result in a terminal execution error.

Now we can fill in the methods `push`, `pop`, and `empty`:

```
void push(int x)
  {
  array[number++] = x;     // put element at position number
  }                        // and increment

int pop()
  {
  return array[--number];  // decrement number and take element
  }

boolean isEmpty()
  {
  return number == 0;      // see if number is 0
  }
```

Note that `number++` means that we increment number *after* using the value, and `--number` means that we decrement number *before* using the value. These are the correct actions, in view of our explanation above.

We can now test the complete program by

```
java -cs Stack 5
```

The argument `-cs` means to compile the source first. A command line is treated as a sequence of strings. The string `"5"`, the first after the name of class whose main is to be invoked, is the first command-line argument, which becomes `arg[0]` of the program.

7.8 Improved Stack Implementation

This first sketch of a Stack class leaves something to be desired. For one thing, if we try to push more items onto the stack than limit specified, there will be a run-time error, or *stack overflow*. It is an annoyance for a program to have to be aware of such a limit. So our first enhancement might be to allow the stack array to grow if more space is needed.

> A program should not abort an application due to having allocated a fixed array size. The program should make provisions for extending arrays and other structures as needed, unless absolutely no memory is left.

In other words, programs that preset the size of internal arrays arbitrarily are less robust than ones that are able to expand those arrays. In terms of object-oriented programming, each object should manage its own storage requirements to preclude premature failure due to lack of space.

To design our class with a growing array, we will add a new method `ensure` that ensures there is enough space before insertion into the array is attempted. If there is not enough space, then `ensure` will replace the array with a larger one.

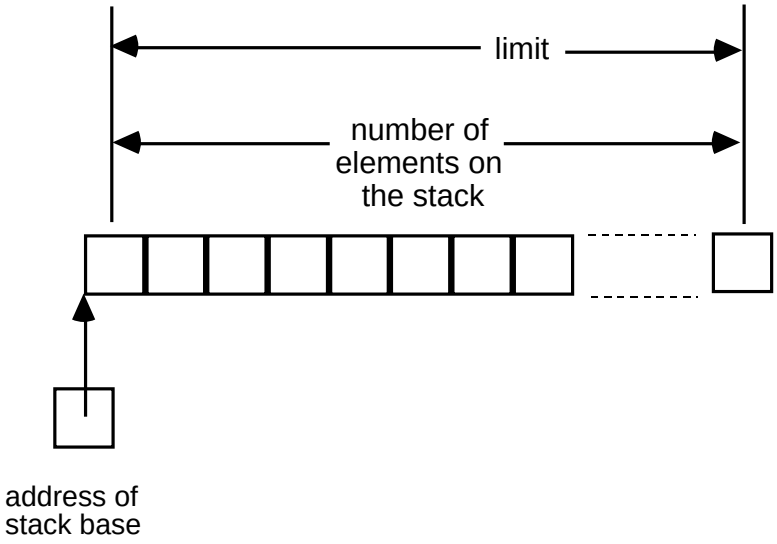


Figure 83: Full stack before extension

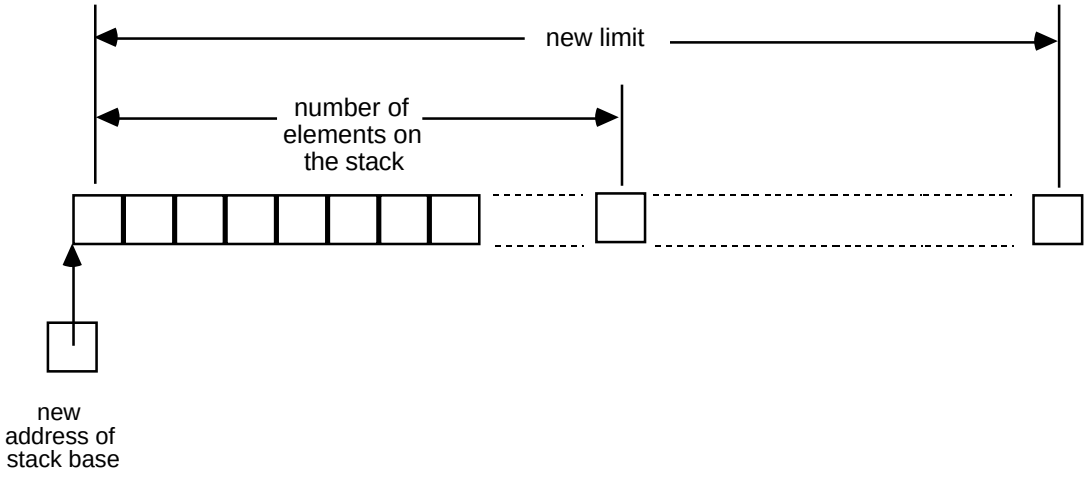


Figure 84: Stack after extension

The incremental size will be controlled by a variable `increment` in the object. For convenience, we will initialize the increment to be the same as the initial limit. Note that we also must keep track of what the current limit is, which adds another variable. We will call this variable `limit`. The value of the variable `limit` in each object will be distinguished from the constructor argument of the same name by qualifying it with `this`, which is a Java keyword meaning *the current object.*

```
class Stack
{
int number;                // number of items in the stack
int limit;                 // limit on number of items in the stack
int increment;             // incremental number to be added
int array[ ];              // stack contents

Stack(int limit)
  {
  this.limit = limit;      // set instance variable to argument value
  increment = limit;       // use increment for limit
  array = new int[limit];  // create array
  number = 0;              // stack contains no items initially
  }

void ensure()              // make sure push is possible
  {
  if( number >= limit )
    {
    int newArray[] = new int[limit+increment];  // create new array
    for( int i = 0; i < limit; i++ )
      {
      newArray[i] = array[i]; // copy elements in stack
      }
    array = newArray;         // replace array with new one
    limit += increment;       // augment the limit
    }
  }
```

```
void push(int x)
  {
  ensure();
  array[number++] = x; // put element at position number, increment
  }
....                   // other methods remain unchanged
}
```

Exercises

1 • Add to the code a method that returns the number of items currently on the stack.

2 • Add to the stack class a second constructor of two arguments, one giving the initial stack size and one giving the increment.

3 •• Suppose we wished for the stack to reduce the size of the allocated array whenever number is less than limit - increment. Modify the code accordingly.

4 ••• Change the policy for incrementing stack size to one that increases the size by a factor, such as 1.5, rather than simply adding an increment. Do you see any advantages or disadvantages of this method vs. the fixed increment method?

5 ••• For the methods presented here, there is no requirement that the items in a stack be in a contiguous array. Instead a linked list could be used. Although a linked list will require extra space for pointers, the amount of space allocated is exactly proportional to the number of items on the stack. Implement a stack based on linked lists.

6 •••• Along the lines of the preceding linked list idea, but rather than linking individual items, link together *chunks* of items. Each chunk is an array. Thus the overhead for the links themselves can be made as small as we wish by making the chunks large enough. This stack gives the incremental allocation of our example, but does not require copying the array on each extension. As such, it is superior, although its implementation is more complicated. Implement such a stack.

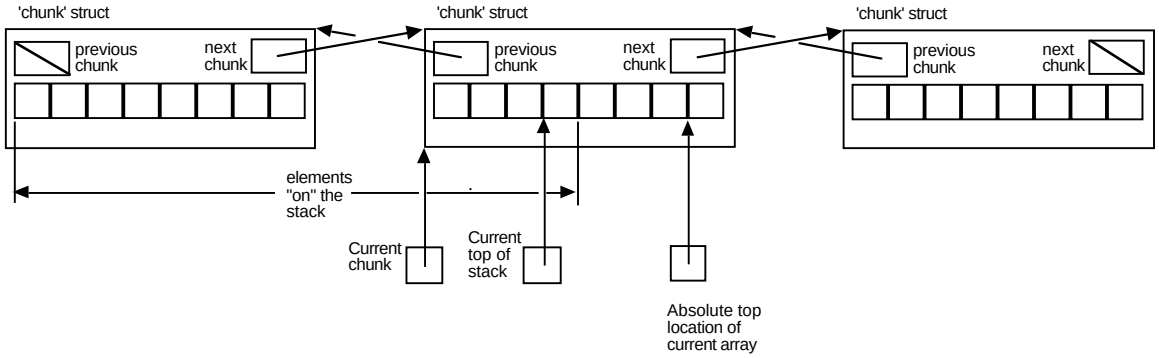


Figure 85: A stack using chunked array allocation

7.9 Classes of Data Containers

A stack is just one form of **data container**, which in turn is just one form of ADT. Different types of container can be created depending on the discipline of access we wish to have. The stack, for example, exhibits a last-in, first-out (LIFO) discipline. Items are extracted in the reverse order of their entry. In other words, extraction from a stack is "youngest out first". A different discipline with a different set of uses is a **queue**. Extraction from a queue is "oldest out first", or first-in, first-out (FIFO). A queue's operations are often called **enqueue** (for inserting) and **dequeue** for extraction. Yet another discipline uses an ordering relation among the data values themselves: "minimum out first". Such a discipline is called a **priority queue**. The figure below illustrates some of these disciplines. A discipline, not shown, which combines both and stack and queue capabilities is a **deque**, for "double-ended queue". The operations might be called **enqueue_top**, **enqueue_bottom**, **dequeue_top**, and **dequeue_bottom**.

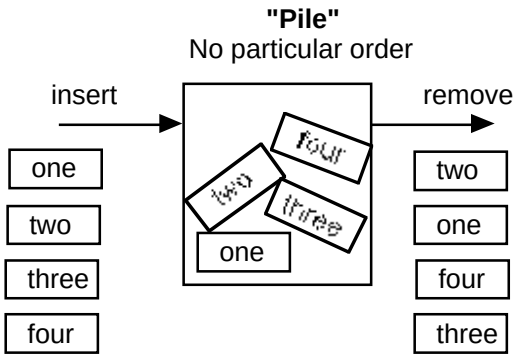


Figure 86: A container with no particular discipline

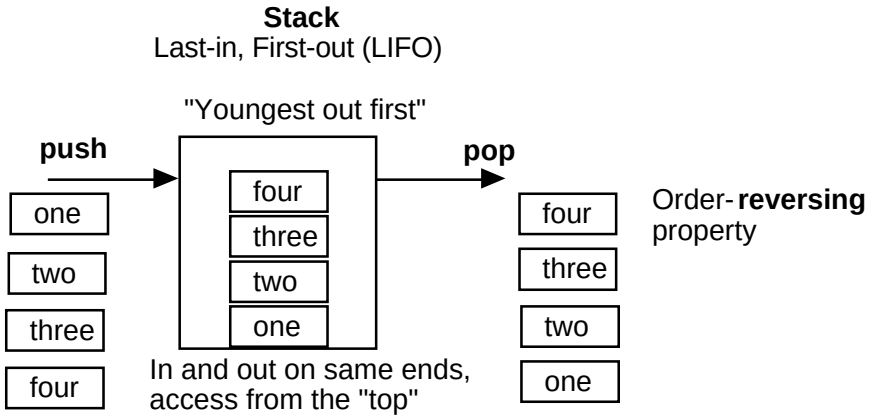


Figure 87: A container with a stack discipline

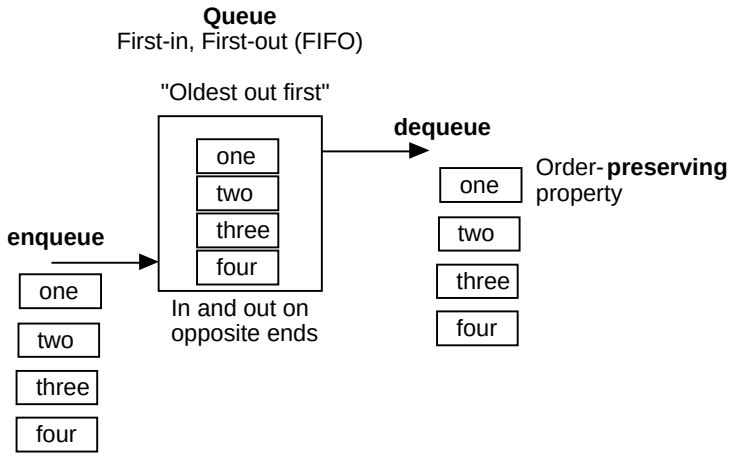


Figure 88: A container with a queue discipline

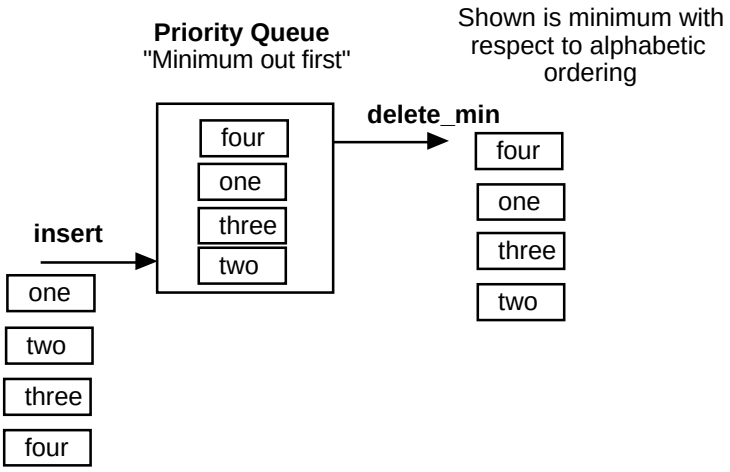


Figure 89: A container with a priority queue discipline

7.10 Implementing a Queue as a Circular Array

Consider implementing a queue data abstraction using an array. The straightforward means of doing this is to use two indices, one indicating the oldest item in the array, the other indicating the youngest. Enqueues are made near the youngest, while dequeues are done near the oldest. The following diagram shows a typical queue state:

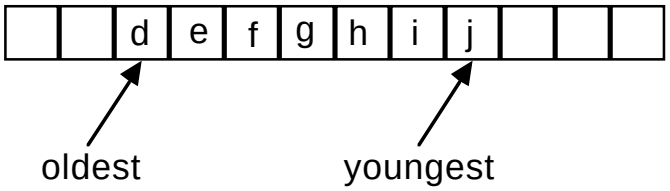


Figure 90: Queue implemented with an array, before enqueuing k

If the next operation is enqueue(k), then the resulting queue will be:

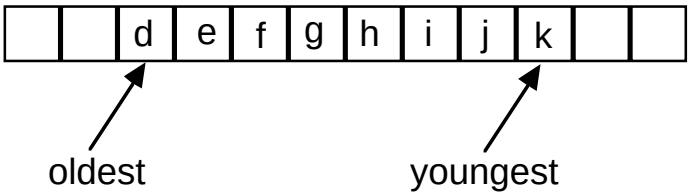


Figure 91: Queue implemented with an array, after enqueuing k

From this state, if the next operation is dequeue(), then d would be dequeued and the resulting state would be:

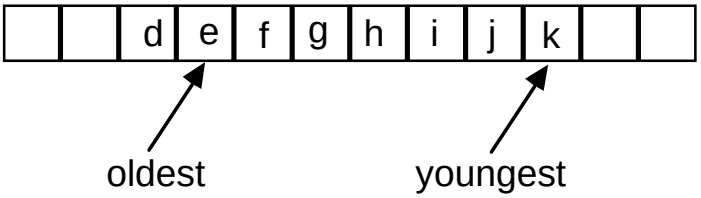


Figure 92: Queue implemented with an array, after dequeuing d

Of course, the value being dequeued need not actually be obliterated. It is the pair of indices that tell us which values in the queue are valid, not the values in the cells themselves. Things get more interesting as additional items are enqueued, until youngest points to the top end of the array. Note that there still may be available space in the queue, namely that below oldest. It is desirable to use that space, by "wrapping around" the pointer to the other end of the queue. Therefore, after having enqueued l, m, n, o, and p, the queue would appear as:

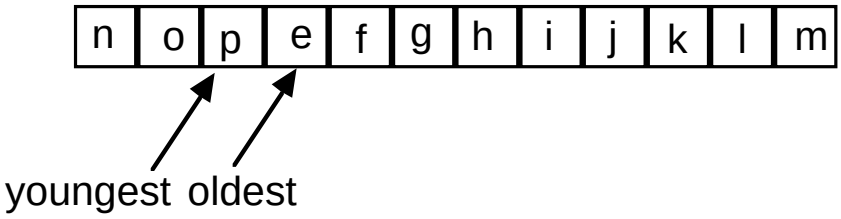


Figure 93: Queue implemented with an array, after wrap-around

When the index values meet, as they have here, we need to allocate more space. The simplest way to do this is to allocate a new array and copy the current valid values into it. From the previous figure, attempting to enqueue q now would cause an overflow condition to result. Assuming we can double the space allocated were the same as that in the original queue, we would then have the following, or one of its many equivalents:

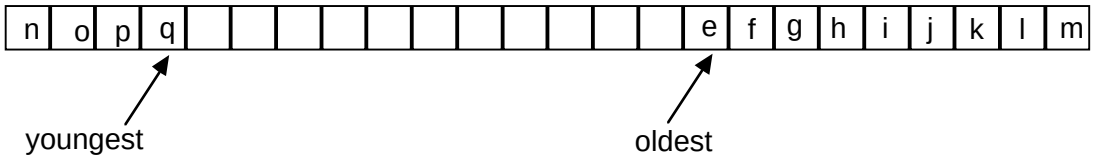


Figure 94: Queue implemented with an array, after space extension

How do we detect when additional allocation is necessary? It is tempting to try to use the relationship between the values of the two indices to do this. However, this may be clumsy to manage (why?). A simpler technique is to just maintain a count of the number in the queue at any given time and compare it with the total space available. Maintaining the number might have other uses as well, such as providing that information to the clients of the queue through an appropriate call.

Exercises

1 ••• Construct a class definition and implementation for a queue of items. Use the stack class example developed here as a model. Use circular arrays as the implementation, so that it is not necessary to extend the storage in the stack unless all space is used up. Be very careful about copying the queue contents when the queue is extended.

2 ••• Construct a class definition and implementation for a *deque* (*double-ended queue*, in which enqueuing and dequeuing can take place at either end). This should observe the same storage economies defined for the queue in the previous exercise.

7.11 Code Normalization and Class Hierarchies

"Normalization" is the term (borrowed from database theory) used to describe an on-going effort, during code development, of concentrating the specification of functionality in as few places as possible,. It has also come to be called “factoring”.

> **Reasons to "normalize" code:**
>
> **Intellectual economy**: We would prefer to gain an understanding of as much functionality as possible through as little code reading as possible.
>
> **Maintainability/evolvability**: Most programs are not written then left alone. Programs that get used tend to evolve as their users desire new features. This is often done by building on existing code. The fewer places within the code one has to visit to make a given conceptual change, the better.

An example of normalization we all hopefully use is through the *procedure* concept. Rather than supporting several segments of code performing similar tasks, we try to generalize the task, package it as a procedure, and replace the would-be separate code segments by procedure calls with appropriate parameters.

Other forms of normalization are:

> Using identifiers to represent key constants.
>
> The *class* concept, as used in object-oriented programming, which encourages procedural normalization by encapsulating procedures for specific abstract data types along with the specifications of those data types.

As we have seen, classes can be built up out of the raw materials of a programming language. However, an important leveraging technique is to build classes out of other classes as well. In other words, an object X can employ other objects Y, Z, ... to achieve X's functionality. The programmer or class designer has a number of means for doing this:

- Variables in a class definition can be objects of other classes. We say that the outer object is **composed** of, or **aggregated** from, the inner objects.

- A class definition can be directly based on the definition of another class (which could be based on yet another class, etc.). This is known as ***inheritance***, since the functionality of one class can be used by the other without a redefinition of this functionality.

As an example of composing an object of one class from an object of another, recall the stack example. The stack was built using an array. One of the functionalities provided for the array was extendability, the ability to make the array larger when more space is needed. But this kind of capability might be needed for many different types of container built from arrays. Furthermore, since the array extension aspect is the trickiest part of the code, it would be helpful to isolate it into functionality associated with the array, and not have to deal with it in classes built *using* the array. Thus we might construct a class `Array` that gives us the array access capability with extension and use this class in building other classes such as stacks, queues, etc. so that we don't have to reimplement this functionality in each class.

```
class Array
{
int increment;                // incremental number to be added
int array[ ];                 // actual array contents

Array(int limit)              // constructor
  {
  increment = limit;          // use increment for limit
  array = new int[limit];     // create actual array
  }

void ensure(int desiredSize)  // make sure size is at least desiredSize
  {
  if( desiredSize > array.length )
    {
    int newArray[] = new int[desiredSize]; // create new array
    for( int i = 0; i < array.length; i++ )
      {
      newArray[i] = array[i]; // copy elements
      }
    array = newArray;         // replace array with new one
    }
  }
}

class Stack             // Stack built using class Array
{
int number;                 // number of items in the stack
int increment;              // incremental number to be added
Array a;                    // stack contents

Stack(int limit)
  {
  a = new Array(limit);     // create array for stack
  increment = limit;        // use increment for limit
  number = 0;               // stack contains no items initially
  }
```

```
void ensure()               // make sure push is possible
  {
  if( number >= a.array.length )
    {
    a.ensure(a.array.length + increment);
    }
  }

void push(int x)
  {
  ensure();
  a.array[number++] = x; // put element at position number, increment
  }

int pop()
  {
  return a.array[--number]; // decrement number and take element
  }
....                         // other methods remain unchanged
}
```

Within class `Stack`, a reference to an object of class `Array` is allocated, here identified by `a`. Notice how the use of the `Array` class to implement the `Stack` class results in a net simplification of the latter. By moving the array extension code to the underlying `Array` class, there is less confusion in the `Stack` class itself. Thus using two classes rather than one results in a separation of concerns, which may make debugging easier.

7.12 Inheritance

A concept linked to that of class, and sometimes thought to be required in object-oriented programming, is that of *inheritance*. This can be viewed as a form of normalization. The motivation for inheritance is that different classes of data abstractions can have functions that are both similar among classes and ones that are different among classes. Inheritance attempts to normalize class definitions by providing a way to merge similar functions across classes.

> Inheritance entails defining a "parent class" that contains common methods and one or more child classes, each potentially with their separate functions, but which also *inherit* functions from the parent class.

With the inheritance mechanism, we do not have to recode common functions in each child class; instead we put them in the parent class.

The following diagram suggests inheritance among classes. At the programmer's option, the sets of methods associated with the child classes either augment or over-ride the methods in the parent class. This diagram is expressed in a standard known as UML (Unified Modeling Language).

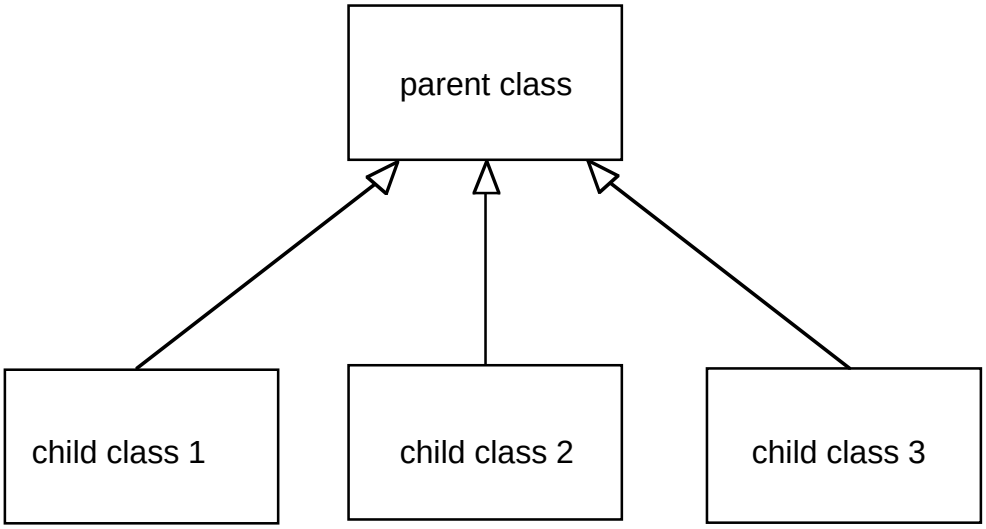


Figure 95: Inheritance among classes

This concept can be extended to any number of children of the same parent. Moreover, it can be extended to a class hierarchy, in which the children have children, etc. The terminology ***base class*** is also used for parent class and ***derived class*** for the children classes. It is also said that the derived class ***extends*** the base class (note that this is a different idea from extending an array).

Possible Advantages of Inheritance:

- Code for a given method in a base class is implemented once, but the method may be used in all derived classes.
- A method declared in a base class can be "customized" for a derived class by *over-riding* it with a different method.
- A method that accepts an object of the base class as an argument will also accept objects of the derived classes.

As an example, let's derive a new class `IndexableStack` from class `Stack`. The idea of the new class is that we can index the elements on the stack. However, indexing takes place from the top element downward, so it is not the same as ordinary array index. In other words, if `s` is an `IndexableStack`, `s.fromTop(0)` represents the top element, `s.fromTop(1)` represents the element next to the top, and so on.

The following Java code demonstrates this derived class. Note that the keyword *extends* indicates that this class is being derived from another.

```
class IndexableStack extends Stack
```

```
{
IndexableStack(int limit)
  {
  super(limit);
  }

int fromTop(int index)
  {
  return a.array[number-1-index];
  }

int size()
  {
  return number;
  }
}
```

Note the use of the keyword `super` in the constructor. This keyword refers to the object in the base class `Stack` that underlies the `IndexableStack`. The use of the keyword with an argument means that the constructor of the base class is called with this argument. In other words, whenever we create an `IndexableStack`, we are creating a `Stack` with the same value of argument `limit`.

Note the use of the identifiers `a` and `number` in the method `fromTop`. These identifiers are not declared in `IndexableStack`. Instead, they represent the identifiers of the same name in the underlying class `Stack`. Every variable in the underlying base class can be used in a similar fashion in the derived class.

Next consider the idea of *over-riding*. A very useful example of over-riding occurs in the class `Applet`, from which user applets are derived. An *applet* was intended to be an application program callable within special contexts, such as web pages. However, applets can also be run in a free-standing fashion. A major advantage of the class `Applet` is that there are pre-implemented methods for handling mouse events (down, up, and drag) and keyboard events. By creating a class derived from class `Applet`, the programmer can over-ride the event-handling methods to be ones of her own choosing, without getting involved in the low-level details of event handling. This makes the creation of interactive applets relatively simple.

The following example illustrates handling of mouse events by over-riding methods `mouseDown`, `mouseDrag`, etc. which are defined in class `Applet`. The reader will note the absence of a main program control thread. Instead actions in this program are driven by mouse events. Each time an event occurs, one of the methods is called and some commands are executed.

Another example of over-riding that exists in this program is in the update and paint methods. The standard applet protocol is that the program does not update the screen directly; instead, the program calls `repaint()`, which will call `update(g)`, where `g` is the applet's graphics. An examination of the code reveals that `update` is never called explicitly. Instead this is done in the underlying Applet code. The reason that `update` calls `paint` rather than doing the painting directly is that the applet also makes implicit

calls to `paint` in the case the screen needs repainting due to being covered then re-exposed, such as due to user actions involving moving the window on the screen.

```
// file:    miniMouse.java
// As mouse events occur, the event and its coordinates
// appear on the screen.

// This applet also prescribes a model for the use of double-buffering
// to avoid flicker: drawing occurs in an image buffer, which is then
// painted onto the screen as needed. This also simplifies drawing,
// since each event creates a blank slate and then draws onto it.

import java.applet.*;           // applet class
import java.awt.*;              // Abstract Window Toolkit


public class miniMouse extends Applet
  {
  Image image;          // Image to be drawn on screen by paint method
  Graphics graphics;    // Graphics part of image, acts as buffer


  // Initialize the applet.

  public void init()
    {
    makeImageBuffer();
    }


   // mouseDown is called when the mouse button is depressed.

  public boolean mouseDown(Event e, int x, int y)
    {
    return show("mouseDown", x, y);
    }


   // mouseDrag is called when the mouse is dragged.

  public boolean mouseDrag(Event e, int x, int y)
    {
    return show("mouseDrag", x, y);
    }


   // mouseUp is called when the mouse button is released.

  public boolean mouseUp(Event v, int x, int y)
    {
    return show("mouseUp", x, y);
    }
```

```
 // mouseMove is called when the mouse moves without being dragged

public boolean mouseMove(Event v, int x, int y)
  {
  return show("mouseMove", x, y);
  }

// show paints the mouse coordinates into the graphics buffer

boolean show(String message, int x, int y)
  {
  clear();
  graphics.setColor(Color.black);
  graphics.drawString(message +
                           " at (" + x + ", " + y + ")", 50, 100);
  repaint();
  return true;
  }


// update is implicitly called when repaint() is called
// g will be bound to the Graphics object in the Applet,
// not the one in the image. paint will draw the image into g.

public void update(Graphics g)
  {
  paint(g);
  }


// paint(Graphics) is called by update(g) and whenever
// the screen needs painting (such as when it is newly exposed)

public void paint(Graphics g)
  {
  g.drawImage(image, 0, 0, null);
  }


// clear clears the image

void clear()
  {
  graphics.clearRect(0, 0, size().width, size().height);
  }

// Make image buffer based on size of the applet.

void makeImageBuffer()
  {
  image = createImage(size().width, size().height);
  graphics = image.getGraphics();
  }
}
```

Drawbacks of Inheritance

While inheritance can be a wonderful time-saving tool, we offer these cautions:

Possible drawbacks of inheritance:

- Once an inheritance hierarchy is built, functionality in base classes cannot be changed unless the impact of this change on derived classes is clearly understood and managed.
- The user of a derived class may have to refer to the base class (and its base class, etc., if any) to understand the full functionality of the class.

There is a fine art in developing the inheritance hierarchy for a large library; each level in the hierarchy should represent carefully-chosen abstractions.

7.12 Inheritance vs. Composition

Inheritance is just one of two major ways to build hierarchies of classes. The second way, which is called composition, is for the new class to make use of one or more objects of other classes. Although these two ways appear similar, they are actually distinct. For example, whereas inheritance adds a new layer of functionality to that of an existing class, composition *uses* functionality of embedded objects but does not necessarily provide similar functionality to the outside world. The following diagram is meant to suggest this distinction.

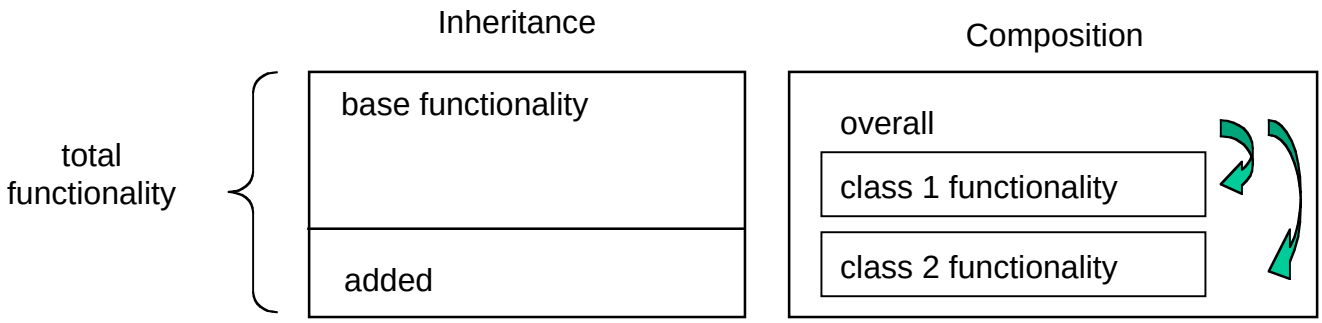


Figure 96: Inheritance vs. Composition

With composition, if the outer classs wishes to provide functionality of inner clases to its clients, it must explicitly provide methods for that purpose. For example, an alternate way to have built the `Stack` class above would be to have `Stack` inherit from `Array`, rather than be composed of an `Array`. In this case, methods such as extend that are available in `Array` would automatically be available in `Stack` as well. Whether or not this is desirable

would depend on the client expectations for `Stack` and other considerations. An advantage is that there is less code for the extension; a disadvantage is that it exposes array-like functionality in the `Stack` definition, upon which the client may come to rely.

Composition should also not be confused with function composition, despite there being a similarity. Yet another construction similar to composition is called aggregation. The technical distinction is that with composition the component objects are not free-standing but are instead a part of the composing object, whereas with aggregation, the components exist independently from the aggregating object. This means that a single object may be aggregated in more than one object, much like structure sharing discussed in Chapter 2.

Exercises

1 ••• Using inheritance from class `Array`, construct a class `BiasedArray` that behaves like an `Array` except that the lower limit is an arbitrary integer (rather than just 0) called the *bias*. In this class, an indexing method, say `elementAt`, must be used in lieu of the usual [...] notation so that the bias is taken into account on access.

2 ••• Code a class `Queue` using class `Array` in two different ways, one using composition and the other using inheritance. Use the *circular array* technique described earlier.

3 ••• Code a class `Deque` using class `Array`.

4 ••• Using aggregation, construct a class `Varray` for virtually concatenating arrays, using the *Principle of Virtual Contiguity* described in *Implementing Information Structures*. One of the constructors for this class should take two arguments of class `Array` that we have already presented and provide a method `elementAt` for indexing. This indexing should translate into indexing for an appropriate one of the arrays being concatenated. Also provide constructors that take one array and one virtual array and a constructor that takes two virtual arrays. Provide as much of the functionality of the class array as seems sensible. The following diagram suggests how varrays work:

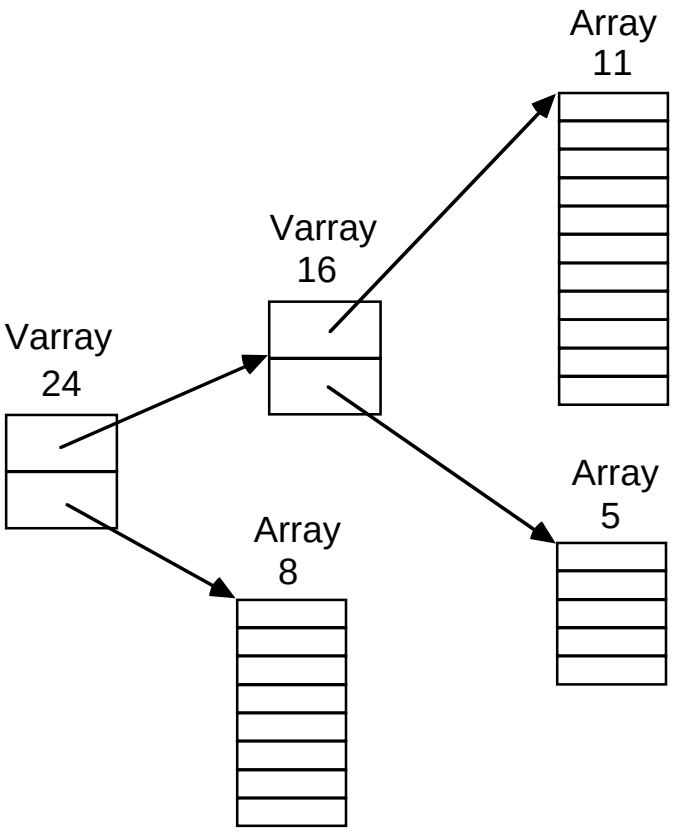


Figure 97: Aggregated objects in a class of virtual arrays

5 •••• Arrays can be constructed of any dimension. Create a class definition that takes the number of dimensions as an *argument* to a constructor. Use a single-dimension array of indices to access arrays so constructed.

7.13 The *is-a* Concept and Sub-Classing

When a class is constructed as a derived class using inheritance, the derived class inherits, by default, the characteristics of the underlying base class. Unless the essential characteristics are redefined, in a sense every derived class object ***is a*** base class object, since it has the capabilities of the base class object but possibly more. For example, an `IndexableStack` ***is a*** `Stack`, according to our definition. This is in the same sense that additional capabilities are possessed by people and things. For example, if the base class is *person* and the derived class is *student*, then a student has the characteristics of a person and possibly more. Another way of saying this is that class *student* is a **sub-class** of class *person*. Every student is a person but not necessarily conversely.

It is common to find class hierarchies in which branching according to characteristics occurs. The programmer should design such hierarchies to best reflect the enterprise underlying the application. For example, if we are developing a computer window system, then there might be a base class window with sub-classes such as:

text_window

graphics_window

window_with_vertical_scrollbar

window_with_horizontal_and_vertical_scrollbars

text_window_with_vertical_scrollbar

graphics_window_with_horizontal_and_vertical_scrollbars

and so on. It is the job of the designer to organize these into a meaningful hierarchy for use by the client and also to do it in such a way that as much code functionality as possible is shared through inheritance.

7.14 Using Inheritance for Interface Abstraction

The type of inheritance discussed could be called **implementation inheritance**, since the objects of the base class are being used as a means of implementing objects of the derived class. Another type of inheritance is called **interface inheritance**. In this form, the *specification* of the interface methods is what is being inherited. As before, each object in the derived class still *is an* object in the base class, so that a method parameter could specify an object of base type and any of the derived types could be passed as a special case.

In Java, there is a special class-like construct used to achieve interface inheritance: The base class is called an `interface` rather than a class. As an example, consider the two container classes `Stack` vs. `Queue`. Both of these have certain characteristics in common: They both have methods for putting data in, removing data, and checking for emptiness. In certain domains, such as search algorithms, a stack or a queue could be used, with attendant effects on the resulting search order.

We could consider both `Stack` and `Queue` to be instances of a common interface, say `Pile`. We'd have to use the same names for addition and removal of data in both classes. So rather than use `push` and `enqueue`, we might simply use `add`, and rather than use `pop` and `dequeue`, we might use `remove`. Our interface declaration might then be:

```
interface Pile
{
void add(int x);

int remove();

boolean isEmpty();
}
```

Note that the interface declaration only declares the types of the methods. The definition of the methods themselves are in the classes that implement the interface. Each such class must define all of the methods in the interface. However, a class may define other methods as well. Each class will define its own constructor, since when we actually create a `Pile`, we must be specific about how it is implemented.

The following shows how class `Stack` might be declared to implement interface `Pile`:

```
class Stack implements Pile  // Stack built using class Array
{
int number;                     // number of items in the stack
int increment;                  // incremental number to be added
Array a;                        // stack contents

Stack(int limit)
  {
  a = new Array(limit);         // create array for stack
  increment = limit;            // use increment for limit
  number = 0;                   // stack contains no items initially
  }

void ensure()               // make sure add is possible
  {
  if( number >= a.array.length )
    {
    a.ensure(a.array.length + increment);
    }
  }

public void add(int x)
  {
  ensure();
  a.array[number++] = x; // put element at position number and increment
  }

public int remove()
  {
  return a.array[--number]; // decrement number and take element
  }

public boolean isEmpty()
  {
  return number == 0;       // see if number is 0
  }
}
```

Note the `public` modifiers before the methods that are declared in the interface. Since those methods are by default public, these modifiers are required in the implementing class. Similarly we might have an implementation of class `Queue`:

```
class Queue implements Pile     // Queue built using class Array
{
int number;                 // number of items in the Queue
int increment;              // incremental number to be added
int oldest;                 // index of first element to be removed
int newest;                 // index of last element added
Array a;                    // Queue contents

Queue(int limit)
  {
  a = new Array(limit);     // create array for Queue
  increment = limit;        // use increment for limit
  number = 0;               // Queue contains no items initially
  oldest = 0;
  newest = -1;
  }
... definition of methods add, remove, empty ...
}
```

Now let's give a sample method that uses a `Pile` as a parameter. We do this in the context of a test program for this class. We are going to test both `Stack` and `Queue`:

```
class TestPile
{
public static void main(String arg[])
  {
  int limit = new Integer(arg[0]).intValue();
  int cycles = new Integer(arg[1]).intValue();

  testPile(new Stack(limit), cycles);
  testPile(new Queue(limit), cycles);
  }

static void testPile(Pile p, int cycles)
  {
  for( int i = 0; i < cycles; i++ )
    {
    p.add(i);
    }
  while( !p.isEmpty() )
    {
    System.out.println(p.remove());
    }
  }
}
```

The important thing to note here is the type of the first parameter `Pile` to `testPile`. Since both `Stack` and `Queue` are special cases of pile, we can use either type as a parameter to `testPile`, as shown in main above.

7.15 Abstract Classes

An idea similar to implementation of an interface is that of an *abstract base class*. In Java terminology, a base class is *abstract* if it is intended to tie together similar derived classes, but there is to be no direct creation of objects of the base class itself. Unlike an interface, objects can actually exist in the abstract class. There might be methods and constructors defined in the abstract class as well. However, similar to an interface, those objects are never created by calling their constructors directly. Instead, their constructors are called in the constructors of classes derived from the abstract class.

Abstract classes can also contain `abstract` method declarations. These methods are similar to the declarations in an interface; they do not specify an implementation; instead this is done in the derived classes.

An interesting example of abstract classes is in a shape-drawing program. There are typically several different types of shapes that can be drawn with the mouse, for example:

Box
Oval
Line

Each of these is an object of a different class. Each has a different way of drawing itself. At the same time, there are certain things we wish to do with shapes that do not need to differentiate between these individual classes. For example, each shape has some reference position that defines a relative offset from a corner of the screen. We would expect to find a `move` method that changes this reference position, and that method will be the same for all shapes.

Our inheritance diagram would appear as in Figure 98, with `Shape` being the abstract class. `Shape` would have an abstract method `draw`, which would be defined specially by each shape class, and a concrete method `move` that changes its reference coordinates.

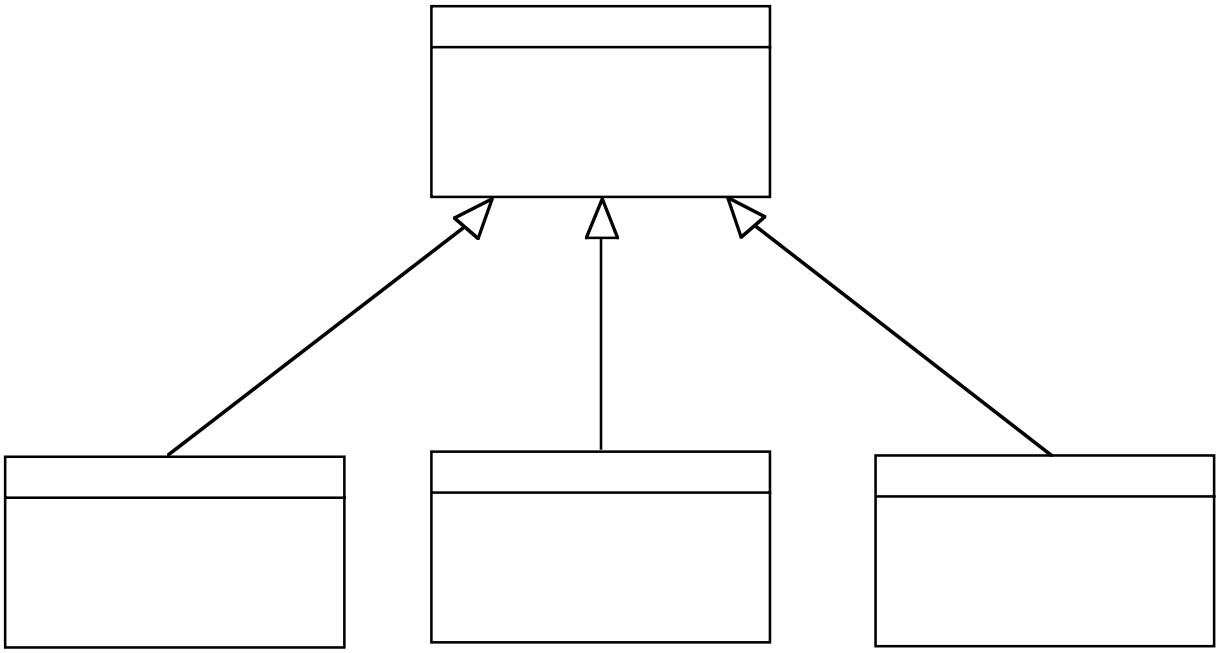


Figure 98: Implementing shapes with inheritance

```
abstract class Shape
  {
  int x, y;                          // coordinates

  Shape(int x, int y)                // constructor
    {
    this.x = x;
    this.y = y;
    }

  void move(int x, int y)            // concrete method
    {
    this.x = x;
    this.y = y;
    }

  abstract void draw(int x, int y);  // defined in derived classes
  }

class Box extends Shape
  {
  Box(int x, int y, ....)            // Box constructor
    {
    super(x, y);                     // call base constructor
    }

  void draw(int x, int y)            // draw method for Box
    {
      ....
    }
  }
```

```
class Oval extends Shape
  {
  Oval(int x, int y, ....)          // Oval constructor
    {
    super(x, y);                    // call base constructor
    }

  void draw(int x, int y)           // draw method for Oval
    {
      ....
    }
  }
....
```

As in the case of interfaces, where both `Stack` and `Queue` could be used for a `Pile`, here both `Box` and `Oval` can be used for a `Shape`. If we have a variable of type `Shape`, we can call its draw method without knowing what kind of shape it is:

```
Box box = new Box(....);
Oval oval = new Oval(....);

Shape shape;

shape = box;
shape.draw(....);

shape = oval;
shape.draw(....);
```

Exactly the same statement may be used to draw either kind of object.

An interesting further step would be to add a class `Group` as a sub-class of `Shape`, with the idea being that a `Group` could hold a list of shapes that can be moved as a unit.

Another example of a class hierarchy with some abstract classes occurs in the Java Abstract Window Toolkit (awt). We show only a portion of this hierarchy, to give the reader a feel for why it is structured as it is. The classes in this hierarchy that we'll mention are:

Component: An abstract class that contains screen graphics and methods to paint the graphics as well as to handle mouse events that occur within.

Specific sub-classes of Component that do not contain other components include:

TextComponent, which has sub-classes
TextField
TextArea.

Label
Scrollbar
Button

List (a type of menu)

Container: An abstract sub-class of Component containing zero or more components. Two specific sub-classes of Container are:

Window, a bordered object that has sub-classes
Frame, which is a Window with certain added features
Dialog

Panel, which has a sub-class **Applet**. A panel has methods for catching mouse events that occur within it.

7.16 The *Object* Class

In Java, there is a single master class from which all classes inherit. This class is called `Object`. If we view the inheritance hierarchy as a tree, then `Object` is the root of the tree.

One approach to creating container classes for different classes of objects is to make the contained type be of class `Object`. Since each class is derived from class `Object`, each object of any class *is* a member of class `Object`. There are two problems with this approach:

1. Not everything to be contained is an object. For example, if we wanted to make a stack of `int`, this type is not an object.

2. Different types of objects can be stored in the same container. This might lead to confusion or errors. The code for accessing objects in the container may get more complicated by the fact that the class of the object will need to be checked dynamically.

Problem 1 can be addressed by *wrapper classes*, to be discussed subsequently. In order to implement checking called for in problem 2, we can make use of the built-in Java operator `instanceof`. An expression involving the latter has the following form:

Object-Reference `instanceof` *Class-Name*

As an example, we could have constructed our class `Stack` using `Object` rather than `int` as the contained type. Then the value returned by the `pop` method would be `Object`. In order to test whether the object popped is of a class `C`, we would have code such as:

```
Stack s = new Stack();
      ....
Object ob = s.pop();

if( ob instanceOf C )
  { .... }
```

7.17 Wrapper Classes

In Java, primitive data items such as `int` and `float` are not objects. However, it is frequently desired to treat them as such. For example, as discussed above, rather than create a different stack class for each different type of datum that we may wish to put in stacks, we could create a single class of type `Object`. The Java language libraries provide classes that serve the purposes of making objects out of primitive objects. But if they didn't, we could still define them ourselves. Frequently used wrapper classes, and the type of data each contains, are:

Wrapper class	Wrapped Type
Integer	int
Long	long
Float	float
Double	double
Char	char
Boolean	boolean

Each wrapper object contains a single object of the wrapped type. Also, these objects are *immutable*, meaning that their values cannot be changed once created.

Methods of the wrapper classes provide ways to extract the wrapped data, and also to construct objects from other types, such as from `String`s. Consult the reference manual for details. The first four of the wrappers mentioned above extend an abstract class called `Number`. By using `Number` as a type, one can extract information using methods such as `floatValue`, without requiring knowledge of whether the actual number is an `Integer`, `Long`, `Float`, or `Double`.

```
Object ob = s.pop();

if( ob instanceOf Number )
  {
  float v = ((Number)ob).floatValue();
  ....
  }
```

7.18 Copying Objects

What does it *mean* to copy an object? Suppose, for example, an object is a list of lists. Does copying this object mean copying the entire list but allowing the elements to be shared among both copies, or does it mean that the elements are copied too?

By **shallow copying**, we mean copying only the references to the elements of the list. A consequence of shallow copying is that the lists cells themselves are shared between the original and the copy. This might lead to unintended side-effects, since a change made in the copy can now change the original. By **deep copying**, we mean copying all of the elements in the list and, if those elements are lists, copying them, and so on. If the list elements are each deep copied recursively, then there is no connection between the copy and the original, other than they have the same shape and atomic values. Obviously we can have types of copying between totally shallow and totally deep copying. For example, we could copy the list elements, but shallow copy them. If those elements are only atoms there would be no sharing. If they are pointers to objects, there still would be some sharing.

Below we illustrate shallow vs. deep copying of an object that references an array. For example, this could be the implementation of a stack as discussed earlier.

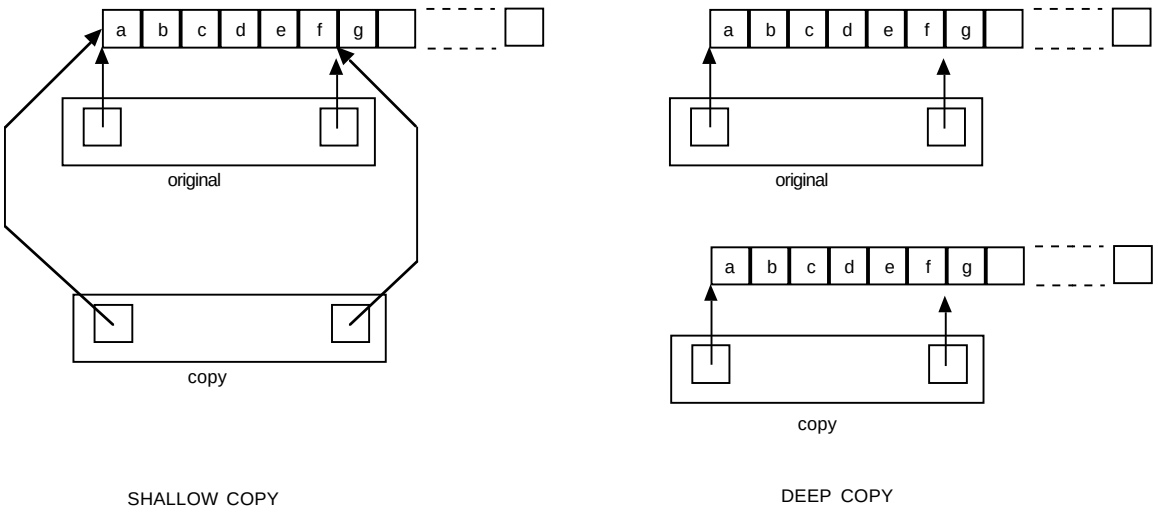


Figure 99: ***Shallow vs. deep copying***

7.19 Equality for Objects

Similar to the copying issue, there is the issue of how objects are compared for equality. We could just compare references to the objects, which would mean that two objects are equal only when they are in exactly the same storage location. This is not a very robust

form of comparison, and is generally meaningful only if an object with a given structure is stored in one unique place, or if we are trying to determine literal identity of objects rather than structural equality. Alternatively, we could compare them more deeply, component-by-component. In this case, there is the issue of how those components are compared, e.g. by reference or more deeply, and so on. It is important to be aware of what equality methods are really doing. The same is true for inequality methods.

7.20 Principle of Interning

For certain cases of read-only objects, such as a set of strings, it is sometimes useful to guarantee that there is *at most one copy* of any object value. Not only does this save space, it allows objects to be compared for equality just by comparing references to them and not delving into their internal structure. This generally improves speed if there are many comparisons to be done. The **principle of interning**, then, is: prior to creating a new (read-only) object, check to see if there is already an object with the same value. If there is, return a reference to the pre-existing object. If not, then create a new object and return a reference to it. The principle of interning is built into languages like Lisp and Prolog: every time a string is read, it is "interned", i.e. the procedure mentioned above is done.

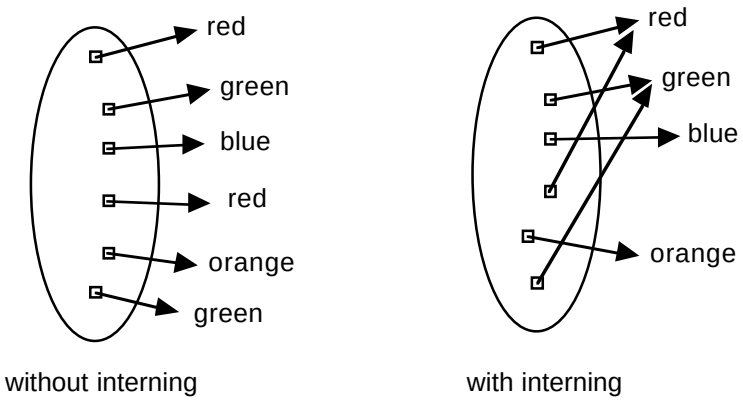


Figure 100: Illustrating use of interning for pointers to read-only strings

A special case of interning can be useful in Java: If we store references to the objects in an array, and refer to the objects by the array index, generally a relatively small index, we can use the **switch** statement to dispatch on the value of an object. Let us call this special case **small-integer interning**.

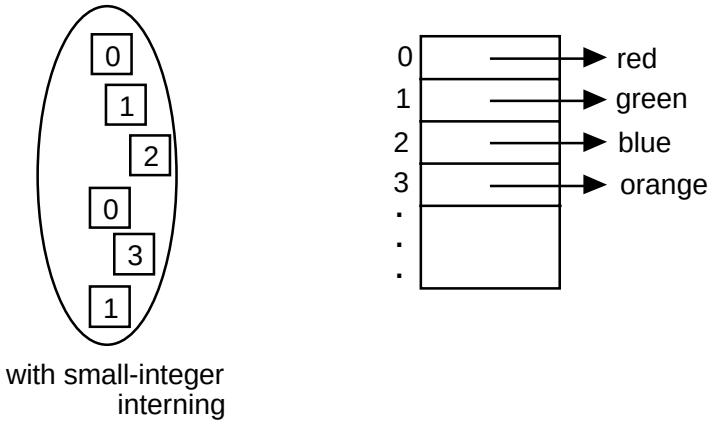


Figure 101: Illustration of small-integer interning

With either ordinary or small-integer interning, a table or list of some kind must be maintained to keep track of the objects that have been allocated. This allows us to search through the set of previously-interned items when a request is made to intern a new item. The use of an array rather than a list for the table, in the case of small-integer interning, allows us to quickly get to the contents of the actual object when necessary.

Exercise

•• 1 Determine whether any standard Java class provides interning. If so, explain how this feature could be used.

7.21 Linked-Lists in Object-Oriented Form

To close this section, we revisit the linked-list implementation discussed in chapter 5. In that implementation, all list manipulation was done with static methods. In our current implementation we will replace many of these with regular methods. For example, if `L` is a list, we will use

```
L.first()
```

to get its first element rather than

```
first(L)
```

One reason this is attractive in Java programming is that to use the static method outside of the list class itself, we would have to qualify the method name with the class name, as in:

```
List.first(L)
```

whereas with the form `L.first()` we would not, since the class is implied from the type of `L` itself.

In presenting our list class, we will use `Object` as the type of a member of list. This is similar in philosophy to some standard Java classes, such as `Vector`. The implication of this choice is that lists may be highly heterogeneous due to the *polymorphic* nature of the Object class. For example, some of the elements of a list may be lists themselves, and some of those lists can have lists as elements, and so on. This gives an easy way to achieve the list functionality of a language such as rex, which was heavily exercised in the early chapters.

Due to the attendant polymorphism of this approach, we call our list class `Polylist`. Another purpose of doing so is that the class `List` is commonly imported into applications employing the Java awt (abstract window toolkit) and we wish to avoid conflict with that name.

We can also introduce input and output methods that are capable of casting Polylists to Strings in the form of S expressions. This is very convenient for building software prototypes where we wish to concentrate on the inner structure rather than the format of data. We can change the input and output syntax to a different form if desired, without disturbing the essence of the application program.

```
public class Polylist
  {
  // nil is the empty-list constant

  public static final Polylist nil = new Polylist();

  private polycell ptr;

  // The constructors are not intended for general use;
  // cons is preferred instead.

  // construct empty Polylist

  Polylist()
    {
    ptr = null;
    }

   // construct non-empty Polylist

  Polylist(Object First, Polylist Rest)
    {
    ptr = new polycell(First, Rest);
    }

   // isEmpty() tells whether the Polylist is empty.

  public boolean isEmpty()
    {
    return ptr == null;
    }
```

```
  // nonEmpty() tells whether the Polylist is non-empty.

  public boolean nonEmpty()
    {
    return ptr != null;
    }

  // first() returns the first element of a non-empty list.

  public Object first()
    {
    return ptr.first();
    }

  // rest() returns the rest of a non-empty Polylist.

  public Polylist rest()
    {
    return ptr.rest();
    }

   // cons returns a new Polylist given a First, with this as a Rest

  public Polylist cons(Object First)
    {
    return new Polylist(First, this);
    }


   // static cons returns a new Polylist given a First and a Rest.

  public static Polylist cons(Object First, Polylist Rest)
    {
    return Rest.cons(First);
    }
  }


public class polycell
  {
  Object First;
  Polylist Rest;

  // first() returns the first element of a NonEmptyList.

  public Object first()
    {
    return First;
    }

  // rest() returns the rest of a NonEmptyList.

  public Polylist rest()
    {
    return Rest;
    }
```

```
// polycell is the constructor for the cell of a Polylist,
// given a First and a Rest.

public polycell(Object First, Polylist Rest)
  {
  this.First = First;
  this.Rest = Rest;
  }
}
```

One possible reason for preferring the static 2-argument form of `cons` is as follows: suppose we construct a list using the 1-argument `cons` method:

```
nil.cons(a).cons(b).cons(c)
```

This doesn't look bad, except that the list constructed has the elements in the reverse order from how they are listed. That is, the first element of this list will be `c`, not `a`.

To give an example of how coding might look using the object-oriented style, we present the familiar append method. Here append produces a new list by following elements of the current list with the elements in the argument list. In effect, the current list is copied, while the argument list is shared.

```
// append(M) returns a Polylist consisting of the elements of this
// followed by those of M.

public Polylist append(Polylist M)
  {
  if( isEmpty() )
    return M;
  else
    return cons(first(), rest().append(M));
  }
```

Exercises

•• 1 Implement a method that creates a range of Integers given the endpoints of the range.

•• 2 Implement a method that returns a list of the elements in an array of objects.

••• 3 Implement a `Stack` class using composition of a `Polylist`.

••• 4 Implement a `Queue` class using linked lists. You probably won't want to use the `Polylist` class directly, since the natural way to implement a queue requires a closed list rather than an open one.

7.22 Enumeration Interfaces

A common technique for iterating over things such as lists is to use the interface `Enumeration` defined in java.util. To qualify as an implementation of an `Enumeration`, a class must provide two methods:

```
public Object nextElement()
public boolean hasMoreElements()
```

The idea is that an `Enumeration` is created from a sequence, such as a list, to contain the elements of the sequence. This is typically done by a method of the underlying sequence class of type

```
public Enumeration elements()
```

that returns the `Enumeration`. The two methods are then used to get one element of a time from the sequence. Note that `nextElement()` returns the next element and has the side-effect of advancing on to the next element after that. If there are no more elements, an exception will be thrown.

Using `Enumeration` interfaces takes some getting used to, but once the idea is understood, they can be quite handy. A typical use of `Enumeration` to sequence through a Polylist would look like:

```
for( Enumeration e = L.elements(); e.hasMoreElements(); )
  {
  Object ob = e.nextElement();
  .... use ob ....
  }
```

This can be contrasted with simply using a Polylist variable, say `T`, to do the sequencing:

```
for( Polylist T = L; T.nonEmpty(); T = T.rest() )
  {
  Object ob = T.first();
  .... use ob ....
  }
```

Note that the `for` statement in the `Enumeration` case has an empty updating step; this is because updating is done as a side effect of the `nextElement` method. A specific example is the following iterative implementation of the `reverse` method, which constructs the reverse of a list. Here `elements()` refers to the elements of *this* list.

```
// reverse(L) returns the reverse of this

public Polylist reverse()
  {
  Polylist rev = nil;
  for( Enumeration e = elements(); e.hasMoreElements(); )
    {
    rev = rev.cons(e.nextElement());
    }
  return rev;
  }
```

Another example is the method `member` that tests whether a list contains the argument:

```
 // member(A) tells whether A is a member of this list

public boolean member(Object A)
  {
  for( Enumeration e = elements(); e.hasMoreElements(); )
    if( A.equals(e.nextElement() )
      return true;
  return false;
  }
```

This form of iteration will not be used for every occasion; for example, recursion is still more natural for methods such as append, which build the result list from the outside-in.

One possible reason to prefer an `Enumeration` is that it is a type, just as a class is a type. Thus a method can be constructed to use an `Enumeration` argument without regard to whether the thing being enumerated is a list, an array, or something else. Thus an `Enumeration` is just an abstraction for a set of items that can be enumerated.

Now we have a look at how the Polylist enumeration is implemented. As with most enumerations, we try not to actually build a new structure to hold the elements, but rather use the elements in place. This entails implementing some kind of *cursor* mechanism to sequence through the list. In the present case, the Polylist class itself serves as the cursor, just by replacing the list with its rest upon advancing the cursor.

The class that implements the enumeration is called PolylistEnum. The method `elements` of class Polylist returns an object of this type, as shown:

```
//  elements() returns a PolylistEnum object, which implements the
//  interface Enumeration.

public PolylistEnum elements()
  {
  return new PolylistEnum(this);
  }
```

The implementation class, `PolylistEnum`, then appears as:

```
public class PolylistEnum implements Enumeration
  {
```

```
Polylist L;               // current list ("cursor")

// construct a PolylistEnum from a Polylist.

public PolylistEnum(Polylist L)
  {
  this.L = L;
  }


// hasMoreElements() indicates whether there are more elements left
// in the enumeration.

public boolean hasMoreElements()
  {
  return L.nonEmpty();
  }


// nextElement returns the next element in the enumeration.

public Object nextElement()
  {
  if( L.isEmpty() )
    throw new NoSuchElementException("No next in Polylist");

  Object result = L.first();
  L = L.rest();
  return result;
  }
}
```

Let's recap how this particular enumeration works:

1. The programmer wants to enumerate the elements of a Polylist for some purpose. She calls the method `elements()` on the list, which returns an `Enumeration` (actually a `PolylistEnum`, but this never needs to be shown in the calling code, since `PolylistEnum` merely implements `Enumeration`.)

2. Method `elements()` calls the constructor of `PolylistEnum`, which initializes `L` of the latter to be the original list.

3. With each call of `nextElement()`, the first of the current list `L` is reserved, then `L` is replaced with its rest. The reserved first element is returned.

4. `nextElement()` can be called repeatedly until `hasMoreElements()`, which tests whether `L` is non-empty, returns false.

Exercises

•• 1 Implement the method `nth` that returns the n*th* element of a list by using an enumeration.

•• 2 Implement a method that returns an array of objects given a list.

•• 3 Locate an implementation of `Enumeration` for the Java class Vector and achieve an understanding of how it works.

••• 4 Implement an `Enumeration` class for an array of `Objects`.

••• 5 Implement an `Enumeration` class that enumerates an array of Objects in reverse order.

7.23 Higher-Order Functions as Objects

The preceding material has shown how we can implement nested lists as in rex. We earlier promised that all of the functional programming techniques that we illustrated could be implemented using Java. The one item unfulfilled in this promise is higher-order functions"higher-order functions" : functions that can take functions as arguments and ones that can return functions as results. We now address this issue.

Java definitely does not allow methods to be passed as arguments. In order to implement the equivalent of higher-order functions, we shall have to use objects as functions, since these *can* be passed as arguments. Here's the trick: the objects we pass or create as functions will have a pre-convened method, say `apply`, that takes an `Object` as an argument and returns an `Object` as a result. We define this class of objects by an interface definition, called `Function1` (for 1-argument function):

```
public interface Function1
  {
  Object apply(Object x);
  }
```

To be used as a function, a class must implement this interface. An example of such a class is one that concatenates the string representation of an object with the String "xxx":

```
Object concatXXX implements Function1
  {
  Object apply(Object arg)
    {
    return "xxx" + arg.toString();
    }

  concatXXX()     // constructor
    {}
  }
```

Note that this particular implementation has a static character, but this will not always be the case, as will be seen shortly. An application of a `concatXXX` method could be shown as:

```
(new concatXXX()) . apply("yyy");    // note: Strings are Objects
```

which would return a String `"xxxyyy"`.

Here's the way `map`, a method which applies a `Function1` to each element of a `Polylist`, would be coded:

```
// map maps an object of class Function1 over a Polylist returning a
// Polylist

Polylist map(Function1 F)
  {
  if( isEmpty() )
    return nil;
  else
    return cons(F.apply(first()), rest().map(F));
  }
```

For example, if the list `L` contained `["foo", "bar", "baz"]` then

```
L.map(new concatXXX)
```

would produce a list containing `["xxxfoo", "xxxbar", "xxxbaz"]`.

More interesting is the case where the object being applied gets some data from "the outside", i.e. through its constructor. Suppose we want a function that returns a function that concatenates a specified prefix, not just "xxx" invariably. Here's how we can modify the class definition `concatXXX` to one, call it `concat`, that does this:

```
Object concat implements Function1
  {
  String prefix;              // prefix to be concatenated

  Object apply(Object arg)
    {
    return prefix + arg.toString();
    }

  concat(String prefix) // constructor
    {
    this.prefix = prefix;
    }
  }
```

Now we can use `map` to create a method that concatenates an argument string to each element of a list:

```
static Polylist concatToAll(String prefix, Polylist L)
  {
  return L.map(new concat(prefix));
  }
```

In rex, the same idea could be shown as:

```
concatToAll(prefix, L) = map((X) => prefix + x, L);
```

We are now just one step away from functions that return functions as results. All such a function needs to do is to call the constructor of a `Function1` object to return a new object that can be applied. For example, the rex definition:

```
f(X) = (Y) => X + Y;
```

represents a function that takes an argument (X) and returns a function. In Java this would be the method:

```
static Object f(Object X)
  {
  return new concat(X.toString());
  }
```

The object-oriented version of higher-order functions might be a little harder to understand than the rex version, which is one reason we wanted to present the rex version first. Underneath the syntax, the implementation using objects is very similar to standard implementations using functions, where the function objects are called *closures* (meaning that they are a closed environment in which otherwise free variables are bound).

Exercises

••• 1 The higher-order function reduce takes a function argument that has two arguments. Define an interface definition `Function2` analogous to `Function1` above, then give the implementation of reduce so as to take a `Function2` as an argument.

•••• 2 Define a method `compose` that takes two `Function1` arguments and returns their composition as a `Function1` argument.

••• 3 Develop a class `FunctionFromArray` that implements a `Function` given an array. The function applied to the integer `i` is to give the ith element of the array.

7.24 Conclusion

This chapter has presented a number of ideas concerning object-oriented programming. Java is an object-oriented language accompanied by a rich library of classes, including an abstract window toolkit for doing graphics, menus, etc. As this book does not attempt to be a tutorial on Java, we recommend other books that can be studied for the finer details. Attempting to program applications in Java is the best way to understand the concepts described here, including inheritance and interfaces.

7.25 Chapter Review

Define the following terms:

abstract base class
abstract data type (ADT)
aggregation
applet
attribute
circular array
class
client
composition
constructor
container class
deep copying
deque
derived class
enumeration (Java-style)
equality
extend (a class)
getter
inheritance
implement (an interface)
interface
interning
is-a
message
method
modularity
normalization (of code)
object
Object class
over-ride
priority-queue
queue
setter
shallow copying
stack
static method
static variable
sub-class
wrapper

7.26 Further Reading

G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, K. Nygaard, *Simula Begin*, Auerbach, Philadelphia, 1973. [Describes the *Simula* language, generally regarded as the original object-oriented programming language. Moderate.]

Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988. [Moderate.]

David Flanagan, *Java in a Nutshell*, Second Edition, O'Reilly, 1997. [A succinct guide to Java for moderately-experienced programmers. Moderate.]

James Gosling, Frank Yellin, the Java Team, *The Java™ application programming interface*, Volumes 1 and 2, Addison-Wesley, 1996. [A thorough guide to the essential Java classes. Moderate.]

James Rumbaugh, Ivar Jacobson, Grady Booch, The unified modeling language reference manual, Reading, Mass : Addison-Wesley, 1998. [One of many references on UML.]

8. Induction, Grammars, and Parsing

8.1 Introduction

This chapter presents the notion of grammar and related concepts, including how to use grammars to represent languages and patterns, after first discussing further the general idea of inductive definitions.

The general mathematical principle of inductive definition will be presented first. We then focus on the use of this principle embodied within the notion of *grammar*, an important concept for defining programming languages, data definition languages, and other sets of sequences. We devote some time to showing the correspondence between grammars and programs for parsing statements within a language.

8.2 Using Rules to Define Sets

In *Low-Level Functional Programming*, we used rules to define partial functions. A partial function can be considered to be just a *set* of source-target pairs, so in a way, we have a preview of the topic of this section, using rules to define sets. There are many reasons why this topic is of interest. In computation, the data elements of interest are generally members of some set, quite often an infinite one. While we do not usually *construct* this entire set explicitly in computation, it is important that we have a way of *defining* it. Without this, we would have no way to argue that a program intended to operate on members of such a set does so correctly. The general technique for defining such sets is embodied in the following principle.

Principle of Inductive Definition

A set S may be defined inductively by presenting

(i) A **basis**, *i.e.* a set of items asserted to be in S.

(ii) A set of **induction rules**, each of which produces, from a set of items known to be in S, one or more items asserted to be in S.

(iii) The **extremal clause**, which articulates that the only members of set S are those obtainable from the basis and applications of the induction rules.

For brevity, we usually avoid stating the extremal clause explicitly. However, it is always assumed to be operative.

The Set ω of Natural Numbers

Here is a very basic inductive definition, of the set of natural numbers, which will be designated ω :

Basis: 0 is in ω .

Induction rule: If *n* is in ω, then so is the number *n*+1.

Unfortunately, this example can be considered lacking, since we haven't given a precise definition of what n+1 means. For example, the definition would be satisfied in the case that n+1 happens to be the same as n. In that case, the set being defined inductively ends up being just {0}, not what we think of as the natural numbers.

We could give more substance to the preceding definition by defining +1 in the induction rule in a more elementary way. Following Halmos, for example, we could define the natural numbers purely in terms of sets. Intuitively, we think of a number n as being represented by a certain set with n items. The problem then is how to construct such a set.

There is only one set with 0 items, the empty set {}, so we will take that to represent the number 0. To build a set of one item, we could take that one item to be the empty set, so the first two numbers are:

0 is equated to {}
1 is equated to {0} *i.e.*, to {{}}

How do we get a set of 2 items? We can't just take two copies of 1, since according to the notion of a set, repetitions don't really count: {{}, {}} would be the same as {{}}, and our 2 would equal 1, not what we want. So instead, take the two elements to be 0 and 1, which we know to be distinct, since 0 is an empty set whereas 1 is a non-empty set:

2 is equated to {0, 1} *i.e.*, to {{}, {{}}}

Continuing in this way,

3 is equated to {0, 1, 2} *i.e.*, to {{}, {{}}, {{}, {{}}}}

How do we form *n*+1 given *n* in general? Since *n* is a set of "n" items, we can get a set of "n+1" items by forming a new set with the elements of *n* and adding in *n* as a new element. But the way to add in a single element is just to form the *union* of the original set with the set of that one element, i.e.

$n + 1$ is equated to $n \cup \{n\}$

Since {n} is distinct from every element of n, this new set has one more element than n has.

The representation above is not the only way to construct the natural numbers. Another way, which gives more succinct representations, but one lacking the virtue that n is represented by a set of elements, is:

0	is equated to	{ }
$n + 1$	is equated to	$\{n\}$

For the same reason as before, 0 is distinct from 1. Similarly, n+1 is distinct from n in general, because at no stage of the construction is {n} ever a member of n. In this definition, numbers are characterized by the number of times an element can be extracted iteratively before arriving at the empty set, e.g.

{ }	0 times
{ { } }	1 time
{ { { } } }	2 times

and so on.

Finite, Infinite, and Countable Sets

Using the first definition of the natural numbers, we are able to give a more accurate definition of what is meant by "finite" and "infinite", terms that get used throughout the book:

> A set is **finite** if there is a one-to-one correspondence between it and one of the sets in ω using the first definition above (i.e. one of the sets of *n* elements for some *n*).
>
> A set that is not finite is called **infinite**.

The set of all natural numbers ω, is the simplest example of an infinite set. Here's how we know that it is infinite: For any element *n* of ω, there is no one-to-one correspondence between *n* and a subset of *n*. (This can be proved by induction). However, for itself, there are many such one-to-one correspondences. The following shows one such correspondence:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots \\ 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & \end{pmatrix}$$

the general rule being that *n* corresponds to $2(n + 1)$. Thus ω and the individual members of ω have substantially different properties.

A set such that there is a one-to-one correspondence between it and ω is called **countably-infinite**. A set is called **countable** if it is either finite or countably infinite. A

set can be shown to be countable if there is a method for enumerating its elements. For example, the display above indicates that the even numbers are countable.

Example: The set of *all* subsets of ω is not countable.

For a justification of this assertion, please see the chapter *Limitations of Computing.*

The Set of All Finite Subsets of ω

Let fs(ω) stand for the set of all finite subsets of ω. Obviously fs(ω) is infinite. We can see this because there is a subset of it in one-to-one correspondence with ω, namely the set {{0}, {1}, {2}, {3}, ...}. To show fs(ω) is countable, we can give a method for enumerating its members. Here is one possible method: Let fs(n) represent the subsets of the set {0, 1, 2, ..., n-1}. An initial attempt at enumeration consists of a concatenation:

fs(0), fs(1), fs(2), fs(3),....

i.e.

fs(0) = {},
fs(1) = {}, {0},
fs(2) = {}, {0}, {1}, {0, 1},
fs(3) = {}, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2}, {0, 1, 2},
...

There are some repetitions in this list, but when we drop the repeated elements, we will have the enumeration we seek:

{}, {0}, {1}, {0, 1}, {2}, {0, 2}, {1, 2}, {0, 1, 2}, ...

The Set Σ^* of Strings over a Given Finite Alphabet Σ

This is another example of a countabley-infinite set defined by induction. It is used very frequently in later sections.

Basis: The empty string λ is in Σ*.

Induction rule: If x is in Σ* and σ is in Σ, then the string σx is in Σ*.

(where σx means symbol σ *followed by* symbols in x.)

As a special case, suppose that Σ = {0, 1}. Then elements of Σ* could be introduced in the following order:

λ, by the basis
0, as λ0, since 0 ∈ Σ and λ is in Σ*
1, as λ1, since 1 ∈ Σ and λ is in Σ*
00, since 0 ∈ Σ and 0 is in Σ*
10, since 1 ∈ Σ and 0 is in Σ*
01, since 0 ∈ Σ and 1 is in Σ*
11, since 1 ∈ Σ and 1 is in Σ*
000, since 0 ∈ Σ and 00 is in Σ*
100, since 1 ∈ Σ and 00 is in Σ*
...

For reemphasis, note that the length of every string in Σ* is finite, while Σ* is a set with an infinite number of elements.

The Sets L_n of Strings of Length n over a Given Alphabet

Often we will want to define a set inductively using an indexed series of previously-defined sets. For example, L_0, L_1, L_2, ... could be a family of sets indexed by the natural numbers and the set we are defining inductively is their union.

Basis: L_0 is $\{\lambda\}$, the set consisting of just the empty string.

Induction rule: L_{n+1} is the set $\{\sigma x \mid \sigma \text{ is in } \Sigma \text{ and } x \text{ is in } L_n\}$

Another way to define Σ* is then $\Sigma^* = L_0 \cup L_1 \cup L_2 \cup \ldots$.

The set of Hypercubes

A hypercube is a particular type of undirected graph structure that has recurrent uses in computer science. The hypercube of dimension n is designated as H_n

Basis: H_0 consists of just one point.

Induction rule: H_{n+1} consists of two copies of H_n, with lines connecting the corresponding points in each copy.

The figure below shows the how the first four hypercubes emerge from this definition.

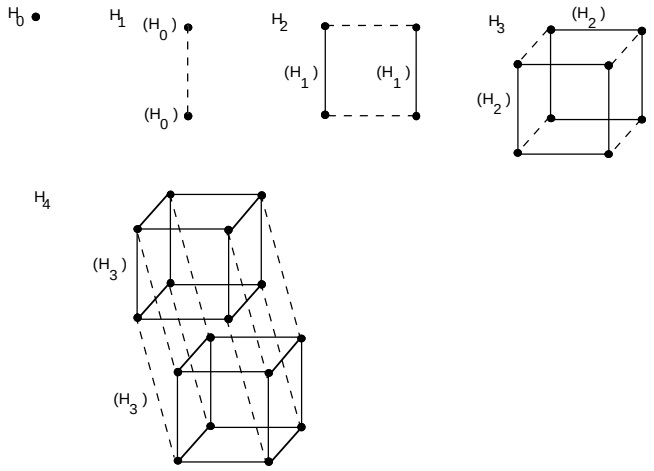


Figure 102: Hypercubes H_0 through H_4.

The Partial Recursive Functions (Advanced)

The partial recursive functions (PRF's) are those functions on the natural numbers that are defined inductively by the following definition. The importance of this set of functions is that it is hypothesized to be exactly those functions that are computable. This was discussed in *States and Transitions* as the Church/Turing Hypothesis.

Basis:

1. Every *constant* function (function having a fixed result value) is a PRF.

2. Every *projection*, a function having the form $\pi_i(x_1, ..., x_n) = x_i$ for fixed i and n, is a PRF.)

3. The *successor* function σ, defined by $\sigma(x) = x + 1$, is a PRF.

Induction rules:

3. Any *composition* of PRF's is a PRF, i.e. if *f* is an n-ary PRF, and $g_1, ..., g_n$ are n m-ary PRF's, then the m-ary function *h* defined by

$$h(x_1, ..., x_m) = f(g_1(x_1, ..., x_m), g_2(x_1, ..., x_m), ..., g_n(x_1, ..., x_m))$$

   is also a PRF.

4. If f is a n-ary PRF and g is an (n+2)-ary PRF, then the (n+1)-ary function *h* defined by *primitive recursion*, i.e. one defined by following the pattern:

$h(0, x_1, ..., x_m) = f(x_1, ..., x_m)$

$h(y+1, x_1, ..., x_m) = g(y, h(y, x_1, ..., x_m), x_1, ..., x_m)$

is a PRF. Note that primitive recursion is just a way to provide definite iteration, as in the case of for-loops.

6. If f is a (n+1)-ary PRF, then the n-ary function g defined by the μ *operator*

$$g(x_1, ..., x_m) = \mu y[f(y, x_1, ..., x_m) = 0]$$

is a PRF. The right-hand side above is read "the least y such that $f(y, x_1, ..., x_m) = 0$". The meaning is as follows: $f(0, x_1, ..., x_m)$ is computed. If the result is 0, then the value of $\mu y[f(y, x_1, ..., x_m) = 0]$ is 0. Otherwise, $f(1, x_1, ..., x_m)$ is computed. If the result is 0, then the value of $\mu y[f(y, x_1, ..., x_m) = 0]$ is 1. If not, then $f(2, x_1, ..., x_m)$ is computed, etc. The value of $\mu y[f(y, x_1, ..., x_m) = 0]$ diverges if there is no value of y with the indicated property, or if any of the computations of $f(y, x_1, ..., x_m)$ diverge.

Notice that primitive recursion corresponds to *definite iteration* in programming languages, such as in a *for loop*, whereas the μ operator corresponds to *indefinite iteration* (as in a *while loop*).

Exercises

1 •• Using the first definition of natural numbers based on sets, give the set equivalents of numbers 4, 5, and 6.

2 •• Give a rex program for a function *nset* that displays the set equivalent of its natural number argument, using lists for sets. For example,

nset(0) ==> [ ]

nset(1) ==> [[ ]]

nset(2) ==> [[[ ]], [ ]]

nset(3) ==> [[[[ ]], [ ]], [[ ]], [ ]]

etc.

3 ••• Let *fswor*(n) (*finite sets without repetition*) refer to the nth item in the list of all finite subsets of natural numbers presented above. Give a rex program that computes *fswor*, assuming that the sets are to be represented as lists.

4 ••• Construct rex rules that will generate the infinite list of finite sets of natural numbers without duplicates. The list might appear as

[[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2], ...]

5 •• Let ω – n mean the natural numbers beginning with n. Show that there is a one-to-one correspondence between ω and ω – n for each n.

6 ••• Show that no natural number (as a set) has a one-to-one correspondence with a subset of itself. (Hint: Use induction.)

7 ••• Construct a rex program that, with a finite set Σ as an argument, will generate the infinite list of all strings in **Σ*.**

8 •• Draw the hypercube H_5.

9 •• Prove by induction that the hypercube H_n has 2^n points.

10 ••• How many edges are in a hypercube H_n? Prove your answer by induction.

11 ••• A *sub-cube* of a hypercube is a hypercube of smaller or equal dimension that corresponds to selecting a set of points of the hypercube and the lines connecting them. For example, in the hypercube H_3, there are 6 sub-cubes of dimension 2, which correspond to the 6 faces as we usually view a 3-dimensional cube. However, in H_4, there are 18 sub-cubes of dimension 3, 12 that are easy to see and another 6 that are a little more subtle. Devise recursive rules for computing the number of sub-cubes of given dimension of a hypercube of given dimension.

12 ••• Refer to item 5 in the definition of partial recursive functions. Assuming that *f* and *g* are available as callable functions, develop both a flowchart and rex code for computing *h*.

13 ••• Refer to item 6 in the definition of partial recursive functions. Assuming that *f* is available as callable functions, develop both a flowchart and rex code for computing *h*.

14 ••• Show that each of the general recursive functions on natural numbers defined in *Low-Level Functional Programming* is also a partial recursive function.

8.3 Languages

An important use of inductive definition is to provide a definition of a **formal language**. Programming languages are examples of formal languages, but we shall see other uses as well. Let us consider a set of symbols Σ. We have already defined Σ* to be the set of all strings of symbols in Σ.

> By a **language** over Σ, we just mean a subset of Σ*, in other words, any set of finite sequences with elements in Σ.

Here are a few examples of simple languages over various alphabets:

- For Σ = {1}, {1}* is the set of all strings of 1's. We have already encountered this set as one way to represent the set of natural numbers.
- For Σ = {1}, {λ, 11, 1111, 111111, ... } is the language of all even-length strings of 1's.
- For Σ = { '(', ')'}, {λ, (), ()(), (()), ()(()), (())(), (())(()), (()()), ((())), ...} is the language of all well-balanced parentheses strings.
- For any Σ, Σ* itself is a language.
- All finite sets of strings over a given set of symbols (which includes the empty set) are all languages.

When languages are reasonably complex, we have to resort to inductive definitions to define them explicitly.

The language of all well-balanced parenthesis strings

This language L is defined inductively as follows:

Basis: λ (the empty string) is in L.

Induction rules:

a. If *x* is in L, then so is the string (*x*).

b. If *x* and *y* are in L, then so is the string *xy*.

For example, we derive the fact that string (()(())) is in L:

1. λ is in L from the basis

2. () is in L, from 1 and rule a.

3. (()) is in L, from 2 and rule a.

4. (()(())) is in L, from 1, 2, and rule b.

Although we have presented languages as *sets* of strings, there is another way to use language-related concepts: to talk about *patterns.* Think of a set (of strings) as corresponding to a (possibly abstract) "pattern" that matches all strings in the set (and only those). For example, the pattern could be "every 0 in the string is immediately followed by two 1s". This corresponds to the language of all strings in which every 0 is immediately followed by two 1s. (Later on, we shall see how to use *regular expressions* as a means for concisely expressing such patterns.) In summary, there is a one-to-one correspondence between patterns and languages. The following section on grammars gives us another, more formal, tool for expressing languages, and therefore patterns.

8.4 Structure of Grammars

The form of inductive definition of languages used above occurs so often that special notation and nomenclature have been developed to present them. This leads to the concept of a *grammar*.

> A **grammar** is a shorthand way of presenting an inductive definition for a language.

A grammar consists of the following four parts:

- The **terminal alphabet**, over which the language being defined is a language.
- The **auxiliary** or **non-terminal alphabet**, which has no symbols in common with the terminal alphabet. The symbols in the auxiliary alphabet provide a kind of "scaffolding" for construction of strings in the language of interest.
- The **start symbol**, which is always a member of the auxiliary alphabet.
- A finite set of **productions**. Each production is a rule that determines how one string of symbols can be rewritten as another.

It is common, but not universal, to write the productions using an arrow, so that production $x \rightarrow y$ means that *x can be rewritten as y*. Thus we have rewriting similar to rex rules, with the following exceptions:

With grammars, rewriting is purely string-based; there is no immediate identification of variables, no arithmetic, no lists, etc. Thus grammars are a lot more primitive than rex rules in this regard.

With grammars, the choice of rule is non-deterministic; we are not required to apply the *first* rule that matches; instead we can apply *any* rule that matches.

In order to define the language defined by a grammar, we begin with the strings generated by a grammar. If G denotes our grammar, then S(G) will denote this set of strings defined inductively:

Basis: The start symbol is in S(G).

Induction step: If *u* is in S(G) and *u* can be written as the concatenation *vxw* and there is a rule $x \rightarrow y$, then *vyw* is also in S(G). Whenever this relation holds, between strings *vxw* and *vyw*, we can write $vxw \Rightarrow vyw$.

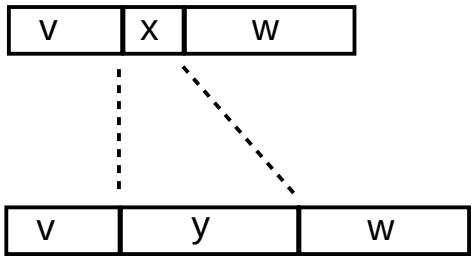


Figure 103: Application of a production x → y, in a string vxw to get string vyw

For example, if $A \rightarrow 0B10$ is a rule, then the following are both true: $1AA \Rightarrow 10B10A$, $1AA \Rightarrow 1A0B10$. Furthermore, $10B10A \Rightarrow 0B100B10$ and also $1A0B10 \Rightarrow 10B100B10$.

> **The language L(G) defined by G** is just that subset of S(G) the strings of that consist of only terminal symbols (*i.e.* no auxiliary symbols in the string).

The definition of $\rightarrow$ and $\Rightarrow$ applies to grammars in which the left-hand side of $\rightarrow$ is an arbitrary string. However, the applications we will consider will mostly involve a left-hand side that is one-character long. Grammars with this property are called *context-free grammars*. When it is necessary to contrast with the fully general definition of grammar, the term *phrase-structure grammar* is sometimes used for the general case.

Grammar for Well-Balanced Parenthesis Strings

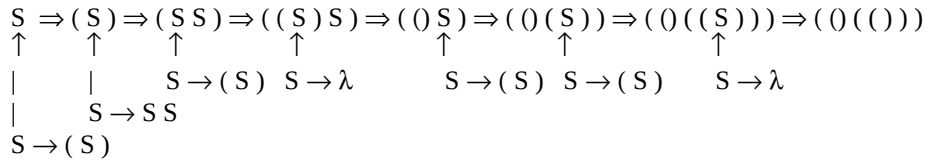
This grammar will exactly mimic the inductive definition for the parenthesis string language given earlier. The terminal alphabet has two symbols { '(', ')' }. The auxiliary alphabet has one symbol S. The start symbol is S. The productions are:

$S \rightarrow \lambda$ corresponding to the basis of the original inductive definition.

$S \rightarrow (S)$ corresponding to rule a. of the original inductive definition.

$S \rightarrow S S$ corresponding to rule b. of the original inductive definition.

Let us construct the derivation of (()(())) using the grammar notation. The top line shows the strings generated, while the bottom line shows the production used at each stepand the arrow points to the symbol being replaced.



Grammar for One-Operator Arithmetic Expressions

This grammar will generate arithmetic expressions of the form a, a+b, a+b+c, etc. The terminal alphabet is { a, b, c, + }. The auxiliary alphabet is {E, V}. The start symbol is E. The productions are:

```
E → V       // Every variable is an expression.
E → E + V   // Every string of the form E + V, where E is an expression
            //      and V is a variable, is also an expression.
V → a
V → b
V → c
```

Abbreviation for Alternatives: The | Symbol

The typical programming language definition includes hundreds of productions. To manage the complexity of such a definition, various abbreviation devices are often used. It is common to group productions by common left-hand sides and list the right-hand sides. One *meta-syntactic* device (i.e. device for dealing with the syntax of grammars, rather than the languages they describe) for doing this is to use the vertical bar | to represent alternatives. The bar binds less tightly than does juxtaposition of grammar symbols. Using this notation, the preceding grammar could be rewritten more compactly as:

E → V | E + V
V → a | b | c

Here the | symbol, like →, is part of the meta-syntax, not a symbol in the language.

We will adopt the practice of *quoting* symbols that are in the object language. With this convention, the grammar above becomes:

E → V | E '+' V

V → 'a' | 'b' | 'c'

Notice that, by repeated substitution for E, the productions for E effectively say the following:

E ⇒ V
E ⇒ E '+' V ⇒ V '+' V
E ⇒ E '+' V '+' V ⇒ V '+' V '+' V
E ⇒ E '+' V '+' V '+' V ⇒ V '+' V '+' V '+' V
...

In other words, from E we can derive V followed by any number of the combination '+' V. We introduce a special notation that replaces the two productions for E:

E → V { '+' V }

read "E produces V followed by *any number of* '+' V". Here the braces {....} represent the *any number of* operator. Later on, in conjunction with regular expressions, we will use (....)* instead of {....} and will call * the *star operator*.

By replacing two productions E → V | E '+' V with a single rule E → V { '+' V } we have contributed to an understanding of how the combination of two productions is actually used.

Other Syntactic Conventions in the Literature

Another related notation convention is to list the left-hand side followed by a colon as the head of a paragraph and the right-hand sides as a series of lines, one right-hand side per line. The preceding grammar in this notation would be:

E:
    V
    E '+' V

V: *one of*

'a' 'b' 'c'

A portion of a grammar for the Java language, using this notation, is as follows:

```
Assignment:
        LeftHandSide AssignmentOperator AssignmentExpression
LeftHandSide:
        Name
        FieldAccess
        ArrayAccess
AssignmentOperator: one of
        = *= /= %= += -= <<= >>= >>>= &= ^= |=
AssignmentExpression:
        ConditionalExpression
        Assignment
ConditionalExpression:
        ConditionalOrExpression
        ConditionalOrExpression ? Expression : ConditionalExpression
```

Grammar for S Expressions

S expressions (S stands for "symbolic") provide one way to represent arbitrarily-nested lists and are an important class of expressions used for data and language representation. S expressions are defined relative to a set of "atoms" A, which will not be further defined in the grammar itself. The start symbol is E. The productions are

$S \rightarrow A$	// Every atom by itself is an S expression.
$S \rightarrow (L)$	// Every parenthesized list is an S expression.
$L \rightarrow \lambda$	// The empty list is a list.
$L \rightarrow S L$	// An S expression followed by a list is a list.

An example of an S expression relative to common words as atoms is:

(This is (an S expression))

Later in this chapter we will see some of the uses of S expressions.

Grammar for Regular Expressions (Advanced)

Regular expressions are used to represent certain kinds of patterns in strings, or equivalently, to represent sets of strings. These are used to specify searches in text editors and similar software tools. Briefly, regular expressions are expressions formed using letters from a given alphabet of letters and the meta-syntactic operators of juxtaposition and | as already introduced. However, instead of using {...} to indicate iteration, the expression that would have been inside the braces is given a superscript asterisk and there is no more general form of recursion. The symbols 'λ' for the empty string and '∅' for the empty *set* are also allowable regular expressions. Each regular expression defines a language. For example,

(0 1)* | (1 0)*

defines a language consisting of the even-length strings with 0's and 1's strictly alternating. In the notation of this chapter, we could have represented the set as being generated by the grammar (with start symbol E)

E → { 0 1 } | { 1 0 }

In the following grammar for regular expressions themselves, the start symbol is R. The productions are:

R → S { '|' S }

R → T { T }

T → U '*'

U → 'λ'

U → '∅'

U → σ (for each symbol σ in A)

U → '(' R ')'

Above the symbol λ is quoted to indicate that we mean that λ is a *symbol* used in regular expressions, to distinguish it from the empty string. We will elaborate further on the use of regular expressions in the chapter *Finite-State Machines*. Meanwhile, here are a few more examples of regular expressions, and the reader may profit from the exercise of verifying that they are generated by the grammar:

(0*1 | 1*0)*

((000)*1)*

A Grammar for Grammar Rules (Advanced)

This example shows that grammars can be self-applied, that the given structure of a grammar rule is itself definable by a grammar. We take the set of auxiliary and terminal symbols to be given (not further defined by this grammar). The start symbol is *rule*. The rules are:

rule → lhs '→' rhs

lhs → auxiliary_symbol

rhs → symbol

rhs → rhs symbol

symbol → 'λ'
symbol → auxiliary_symbol
symbol → terminal_symbol

Note that in this grammar we have not included the meta-syntactic | and { } operators. Doing so is left to the reader. It is a simple matter of combining this and the previous example.

The grammars described above all had the property that the left-hand side of a rule consists of a single auxiliary symbol. As already mentioned, such a grammar is known as a **context-free grammar**. There are languages that are definable by rules with *strings* of grammars symbols on the left-hand side that are not definable by context-free grammars. The multiple-symbol capability for a general grammar allows arbitrary Turing machines to be simulated by grammar rules. This most general type of grammar is known as a **phrase-structure grammar**.

8.5 The Relationship of Grammars to Meaning

So far, our discussion of grammars has focused on the "syntactic" aspects, or "syntax", of a language, i.e. determining the member strings of the language. A second role of grammars bears a relation to the "semantics" of the language, i.e. determining meanings for strings. In particular, the *way* in which a string is derived using grammar rules is used by the compiler of a programming language to determine a meaning of strings, i.e. programs. The grammar itself does not provide that meaning, but the structure of derivations in the grammar allows a meaning to be assigned.

In the arithmetic expression example, we can think of a meaning being associated with each use of an auxiliary symbol:

The meaning of an E symbol is the value of an *expression*.

The meaning of a V symbol is the value of the corresponding *variable* (a, b, or c).

The production E → V suggests that the *value* of a single-variable expression is just the *value* of the variable.

The production E → E '+' V suggests that the *value* of an expression of the form E + V is derived from the sum of the *values* of the constituent E and V.

Derivation Trees

The easiest way to relate grammars to meanings is through the concept of **derivation tree**. A derivation tree is a tree having symbols of the grammar as labels on its nodes, such that:

> If a node with parent labeled P has children labeled C_1, C_2, ..., C_n, then there is a production in the grammar P → C_1 C_2 ....C_n.

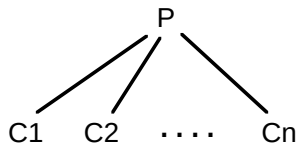


Figure 104: Depicting a production in a derivation tree

A **complete derivation tree** is one such that the root is labeled with the start symbol of the grammar and the leaves are labeled with terminal symbols.

Example

Consider again the grammar for additive arithmetic expressions:

E → V
E → E '+' V
V → a | b | c

Here *E* is the start symbol, and *a*, *b*, and *c* are terminals.

Below we show two derivation trees for this grammar, and a third tree that is not.

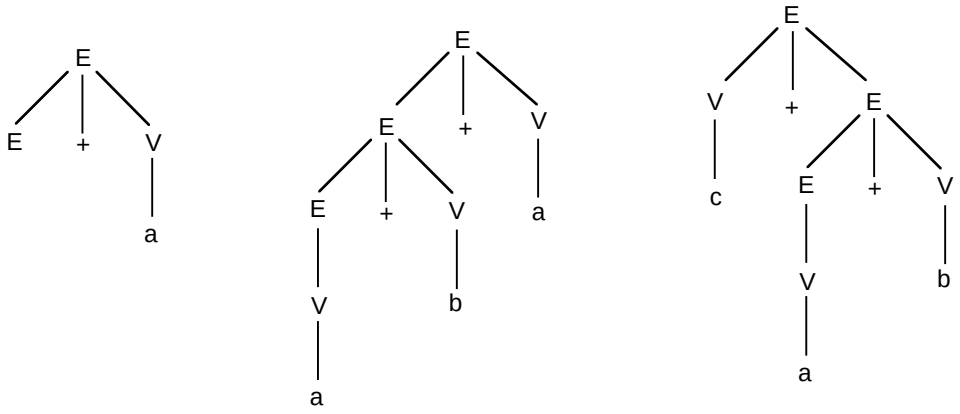


Figure 105: On the left is a derivation tree for the grammar above. In the middle is a complete derivation tree. Despite its similarity, the tree on the right is not a derivation tree for this grammar, since there is no production E → V '+' E.

Notice that the leaves of a complete derivation tree, when read left to right, give a string that is in the language generated by the grammar, for example a+b+a is the leaf sequence for the middle diagram. In this case, we say the derivation tree *derives* the string.

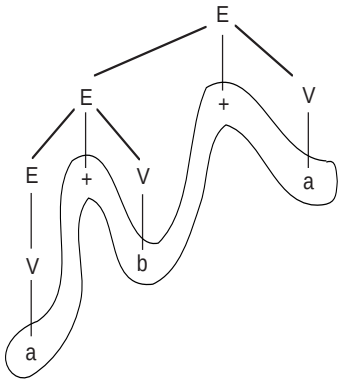


Figure 106: Deriving the string a + b + a in a complete derivation tree

A meaning to the strings in a grammar can be explained by associating a meaning to each of the nodes in the tree. The leaf nodes are given a meaning of some primitive value. For example, *a*, *b*, and *c* might mean numbers associated with those symbols as variables. The meaning of + might be the usual add operator. Referring to the tree above, the meaning of the V nodes is just the meaning of the symbol nodes below them. The meaning of the E nodes is either the sum of the nodes below, if there are three nodes, or the meaning of the single node below. So if *a*, *b*, and *c* had meanings 3, 5, and 17, we would show the meanings of all nodes as follows:

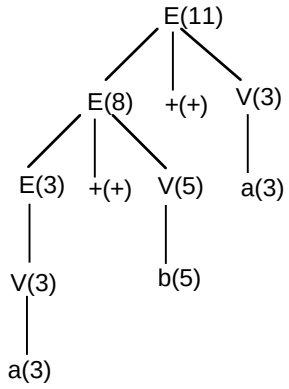


Figure 107: Complete derivation tree annotated with meanings (in parentheses) for each node

Processing a string in a language to determine its meaning, known as **parsing**, is essentially a process of constructing a derivation tree that derives the string. From that tree, a meaning can be determined. In some sense, parsing is like working backward against an inductive definition, to determine whether or not the proposed end result can be derived. Parsing is one of the tasks that programming language compilers have to perform.

Ambiguity

In order to convert a string into a derivation tree and then to a meaning, it is helpful for each string in the language to have a unique derivation tree. Otherwise there might be ambiguity in the meaning.

> A grammar is called **ambiguous** if there is at least one string in the language the grammar generates that is derived by more than one tree.

Example – An ambiguous grammar

The grammar below will generate arithmetic expressions of the form a, a+b, a+b+c, a*b, a+b*c, etc. The terminal alphabet is { a, b, c, + }. The auxiliary alphabet is {E, V}. The start symbol is E. The productions are:

$E \rightarrow V$
$E \rightarrow E$ '+' E
$E \rightarrow E$ '*' E
$V \rightarrow a \mid b \mid c$

To see that the grammar is ambiguous, we present two different trees for the string a+b*c.

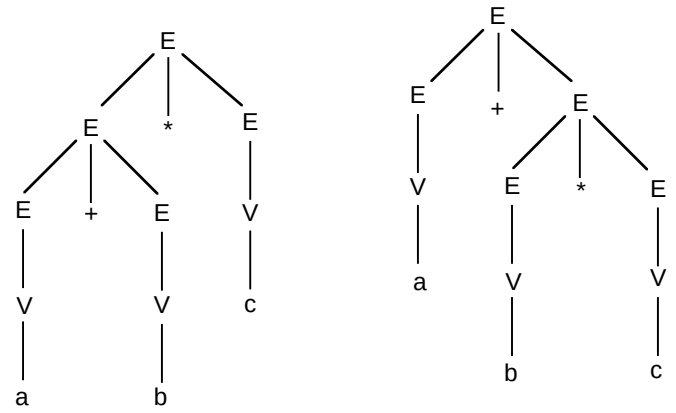


Figure 108: Two distinct derivation trees for a string in an ambiguous grammar

For example, if *a*, *b*, and *c* had meanings 3, 5, and 17, the tree on the left would give a meaning of 136 for the arithmetic expression, whereas the tree on the right would give a meaning of 88. Which meaning is correct? With an ambiguous grammar, we can't really say. But we can say that in common usage, a+b*c corresponds to a+(b*c) (i.e. * *takes precedence* over +), and not to (a+b)*c. The expression usually regarded as correct corresponds to the right-hand derivation tree.

Although the problem of ambiguity is sometimes resolved by understandings about which rule should be given priority in a derivation when there is a choice, it is perhaps most cleanly resolved by finding another grammar that generates the same language, but one that is not ambiguous. This is usually done by finding a different set of productions that does not cause ambiguity. For the present language, a set of productions that will do is:

An unambiguous grammar for simple arithmetic expressions

$$E \rightarrow T$$
$$E \rightarrow E\ \text{'+'}\ T$$
$$T \rightarrow V$$
$$T \rightarrow T\ \text{'*'}\ V$$
$$V \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'}$$

This implements * taking precedence over +.

Here we have added a new auxiliary T (for "term"). The idea is that value of an expression (an E) is either that of a single term or that of an expression plus a term. On the other hand, the value of a term is either that of a variable or a term times a variable. There is now only one leftmost derivation of each string. For example, the derivation of a+b*c is

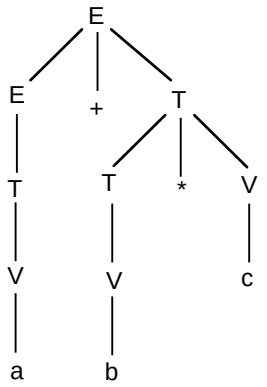


Figure 109: Derivation of a+b*c in an unambiguous grammar

In essence, the new grammar works by enforcing precedence: It prohibits us from expanding an expression (i.e. an E-derived string) using the * operator. We can only expand a term (i.e. a T-derived string) using the * operator. We can expand expressions using the + operator. But once we have applied the production, we can only expand the term, i.e. only use the * operator, not the + operator. Thus the new grammar has a *stratifying* effect on the operators.

The new grammar coincidentally enforces a left-to-right grouping of sub-expressions. That is, a + b + c is effectively grouped as if (a + b) + c, rather than a + (b + c). To see this, note that the derivation tree for this expression is:

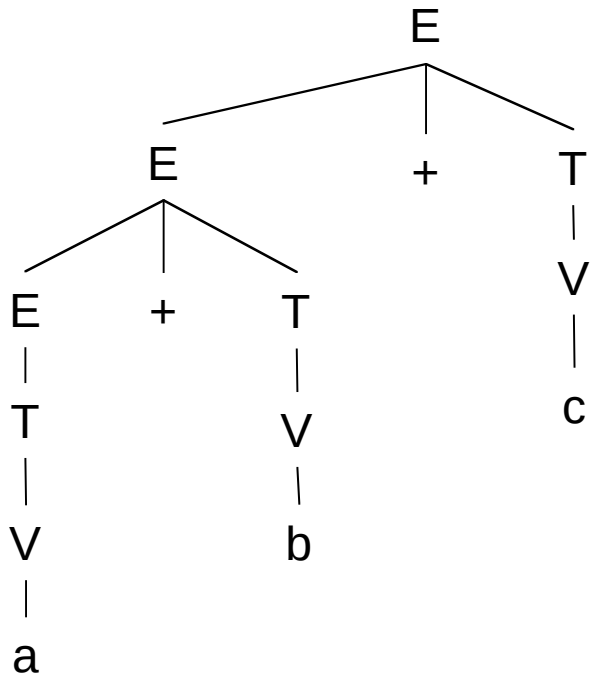


Figure 110: Derivation tree showing grouping

Exercises

1 •• Try to find another derivation tree for a+b*c in the unambiguous grammar. Convince yourself it is not possible.

2 •• Show that the grammar for regular expressions given earlier is ambiguous.

3 •• Give an unambiguous grammar for regular expressions, assuming that * takes precedence over juxtaposition and juxtaposition takes precedence over |.

4 ••• Determine, enough to convince yourself, whether the given grammar for S expressions is ambiguous.

5 •• The unambiguous grammar above is limited in that all the expressions derived within it are *two-level* + and *. That is, we can only get expressions of the form:

V*V*...V + V*V*...V + ... + V*V*...V

consisting of an overall sum of terms, each consisting of a product of variables. If we want more complex structures, we need to introduce parentheses:

(a + b)*c + d

for example. Add the parenthesis symbols to the unambiguous grammar and add a new production for including parentheses that represents this nuance and leaves the grammar unambiguous.

Abstract Syntax Trees

A tree related to the derivation tree is called the **abstract-syntax tree**. This tree is an abstract representation of the *meaning* of the derived expression. In the abstract-syntax tree, the non-leaf nodes of the tree are labeled with *constructors* rather than with auxiliary symbols from the grammar. These constructors give an indication of the meaning of the string derived.

In the particular language being discussed, it is understood that an expression is a summation-like idea, and the items being summed can be products, so an abstract syntax tree would be as follows:

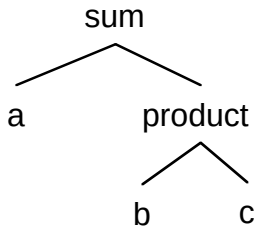


Figure 111: Abstract syntax tree for a+b*c

Note that abstract syntax does not show the specific operators, such as + and *; it only shows the type of entity and the constituent parts of each such entity. Using abstract syntax, we can appreciate the *structure* of a language apart from its "concrete" representation as a set of strings.

An abstract syntax tree could be represented as the result of applying *constructors*, each of which makes a tree out of component parts that are subtrees. For the language discussed in preceding examples, the constructors could be as follows (where *mk* is an abbreviation for *make*):

mk_sum
mk_product
mk_variable

Equivalent to the abstract syntax tree is an expression involving the constructors. For the tree above, this expression would be:

```
mk_sum(mk_variable('a'),
       mk_product(mk_variable('b'),
                  mk_variable('c')))
```

The final meaning of the overall expression can now be derived by simply evaluating the constructor functions appropriately redefined. For example, if we wanted to compute an arithmetic value, we would use the following definitions for the constructors:

```
mk_sum(X, Y) => X + Y;
mk_product(X, Y) => X * Y;
mk_variable(V) => .... code to look up V's value in a symbol table....
```

On the other hand, if our objective were to generate machine language code, we would use functions that produce a code list. If we are just interested in verifying that our parser works, we could use functions that return a description of the abstract syntax of the expression, such as:

```
mk_sum(X, Y) => [“sum”, X, Y];
mk_product(X, Y) => [“product”, X, Y];
mk_variable(V) => [“fetch”, V];
```

One of the nice things about the abstract syntax idea is that we can leave these functions unspecified until we decide on the type of output desired.

Abstract Grammars (Advanced)

Corresponding to abstract syntax, we could invent the notion of an "**abstract grammar**". Again, the purpose of this would be to emphasize structure over particular character-based representations. For the arithmetic expression grammar, the abstract grammar would be:

$$E \rightarrow V$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$

Abstract grammars are frequently left ambiguous, as the above grammar is, with an understanding that the ambiguity can be removed with appropriate manipulation to restore precedence. The purpose of such abstract grammars is to convey the general structure of a programming language as succinctly as possible and issues such as precedence just get in the way.

Example

Assume that mailing lists are represented the following way: There is a list of all mailing lists. Each list begins with the name of the list, followed by the people in the list. (Assume that lists can't appear on lists.) So, for example, if the lists were "students", "faculty", and "staff", the overall list of lists might be:

```
[["students", "Joe", "Heather", "Phil"],
 ["faculty", "Patti", "Sam", "John", "Cathy"],
 ["staff", "Phil", "Cathy", "Sheldon"]]
```

Suppose we want to specify a grammar for the correct construction of such an overall list of lists (with an arbitrary number of lists and list members). There are two types of answer: An ordinary grammar would require that one specify each symbol, including brackets, commas, and quote marks. An *abstract grammar* (corresponding to *abstract syntax*) gives just the list *structure* without the specific punctuation marks.

An abstract grammar for this problem, with S as the start symbol, is:

S → { M } // directory of zero or more mailing lists

M → N { N } // list name followed by people names

N → A { A } // a list or person name

A → 'a' | 'b' | .... | 'Z'

An ordinary grammar, with S as the start symbol, is:

S → '[' L ']'

L →> λ | M { ',' M } // directory of zero or more mailing lists

M → '[' N { ',' N } ']' // list name followed by people names

N → '"' A { A } '"' // one or more letters in quotes symbols

A → 'a' | 'b' | .... | 'Z'

Additional Meta-Syntactic Forms

We mentioned in the previous section how the symbols | and * can be used to fold several rules together into one. There are several different symbols that are commonly used for informal and formal communication about patterns.

The Brackets-as-Optional Convention:

In many descriptions of patterns, brackets around text means that the text is *optional*. If we include this convention in grammar rules, for example, then

N → ['+'] D

reads: "An N produces an optional + followed by a D." This can be taken as an abbreviation for two separate productions, one that includes the + and one that does not.

N → '+' D
N → D

For example, if we are describing numerals with an optional + or - sign, we could use brackets in combination with |:

Numeral → ['+' | '-'] Digits

This convention is used to describe optional command-line arguments for UNIX™ programs. For example, typing 'man awk' produces for the synopsis something like:

`awk [ -f` *program-file*`] [`*program*`] [`*parameters*`] [`*filename*`]`

meaning that there are four optional command-line arguments: a program-file (preceded by `-f`), a literal program, some parameters, and a filename. (Presumably there won't be both a program-file and a program, but the specification does not preclude it.)

The Braces-as-Alternatives Convention:

In other descriptions of patterns, people use braces to represent a number of different alternatives rather than *any number of* as we have been doing. For example, if Bob, Josh, and Mike all have email addresses at cs.hmc.edu, then one might give their addresses collectively as

```
{bob, josh, mike}@cs.hmc.edu
```

rather than spelling them out:

```
bob@cs.hmc.edu, josh@cs.hmc.edu, mike@cs.hmc.edu.
```

Obviously this convention is redundant in grammars if both | and [...] conventions are operative. One also sometimes sees the same effect using braces or brackets, with the alternatives stacked vertically:

```
⎧bob  ⎫
⎨josh ⎬ @cs.hmc.edu
⎩mike ⎭
```

The Ellipsis Convention:

This is another way to represent iteration:

(D ...)

means 0 or more D's.

8.6 Syntax Diagrams

The graphical notation of "syntax diagrams" is sometimes used in place of grammars to give a better global understanding of the strings a grammar defines. In a syntax diagram, each left-hand side auxiliary symbol has a graph associated with it representing a combination of productions. The graph consists of arrows leading from an incoming side to an outgoing side. In between, the arrows direct through terminal and auxiliary symbols. The idea is that all strings generated by a particular symbol in the grammar can be obtained by following the arrows from the incoming side to the outgoing side. In turn, the auxiliary symbols that are traversed in this process are replaced by traversals of the graphs with those symbols as left-hand sides. A recursive traversal of the graph corresponding to the start symbol corresponds to a string in the language.

A syntax diagram equivalent to the previous grammars is shown below. Syntax diagrams are sometimes preferred because they are intuitively more readable than the corresponding grammar, and because they enable one to get a better overview of the connectivity of concepts in the grammar. Part of the reason for this is that recursive rules in the grammar can often be replaced with cyclic structures in the diagram.

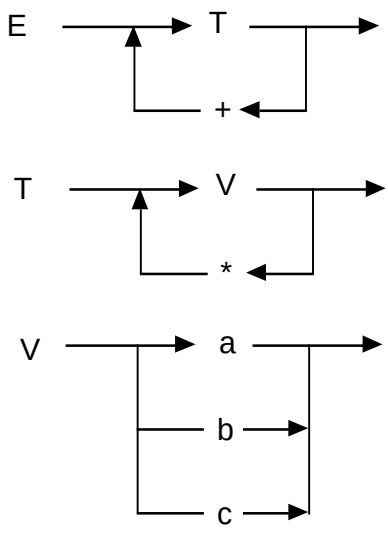


Figure 112: Syntax diagram for a simple language

Derivation by syntax diagram is similar to derivation in a grammar. Beginning with the graph for the start symbol, we trace out a path from ingoing to outgoing. When there is a "fork" in the diagram, we have a *choice* of going either direction. For example, in the diagram above, E is the start symbol. We can derive from E the auxiliary T by taking the upper choice, or derive T + ... by taking the lower choice. Thus we can derive any number of T's, separated by +'s. Similarly, any T derives any number of V's separated by *'s. Finally a V can derive only one of a, b, or c. So to generate a+b*c, we would have the following steps:

E
T+T looping back once through the E graph
V+T going straight through the T graph
V+V*V looping back once through the T graph
a+V*V
a+b*V using the V graph three times, each with a different choice
a+b*c

A syntax diagram for S expressions over an unspecified set of atoms would be:

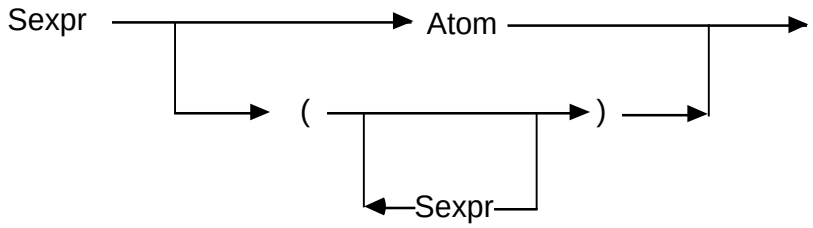


Figure 113: S expression syntax diagram

Actually, there is another more general notion of S expressions, one that uses a period symbol '.' in a manner analogous to the vertical bar in rex list expressions. For example, (A B (C D) . E) corresponds to [A, B, [C, D] | E]. This diagram for this generalized form is:

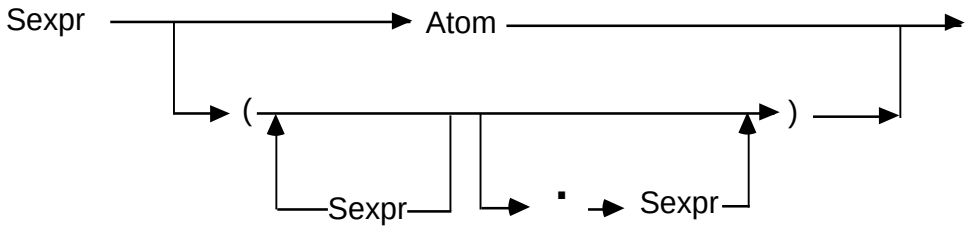


Figure 114: Generalized S expression syntax diagram

Exercises

1 •• Give a syntax diagram for the set of well-balanced parenthesis strings.

2 •• Give a syntax diagram, then a grammar, for the set of signed integers, i.e. the set of strings consisting of the any number of the digits {0, 1, ..., 9} preceded by an optional + or -.

3 •• Give a grammar corresponding to the generalized S expression syntax given in the diagram above.

4 ••• Give a syntax diagram, then a grammar, for the set of floating-point numerals, i.e. the set of strings consisting of an optional + or -, followed by an optional whole number part, followed by an optional decimal point, an optional fraction, and optionally the symbol 'e' followed by an optional signed exponent. The additional constraint is that, with all the options, we should not end up with numerals having *neither* a whole number part nor a fraction.

5 ••• Give a syntax diagram for terms representing lists as they appear in rex. Include brackets, commas, and the vertical bar as terminal symbols, but use A to represent atomic list elements.

6 ••• Locate a complete copy of the Java grammar and construct syntax diagrams for it.

8.7 Grouping and Precedence

We showed earlier how the structure of productions has an effect on the precedence determination of operators. To recapitulate, we can insure that an operator * has higher precedence than + by structuring the productions so that the + is "closer" to the start symbol than *. In our running example, using the * operator, E is the start symbol:

$$E \rightarrow T \{ \text{'+'} T \}$$
$$T \rightarrow V \{ \text{'*'} V \}$$
$$V \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'}$$

> Generally, the lower precedence operators will be "closer" to the start symbol.

For example, if we wanted to add another operator, say ^ meaning "raise to a power", and wanted this symbol to have higher precedence than *, we would add ^ "farther away" from the start symbol than '*' is, introducing a new auxiliary symbol P that replaces V in the existing grammar, and adding productions for P:

$$E \rightarrow T \{ \text{'+'} T \}$$
$$T \rightarrow P \{ \text{'*'} P \}$$
$$P \rightarrow V \{ \text{'^'} V \}$$
$$V \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'}$$

Another issue that needs to be addressed is that of **grouping**: how multiple operators at the same level of precedence are grouped. This is often called "associativity", but this term is slightly misleading, because the discussion applies to operators that are not necessarily associative. For example, an expression

a + b + c

could be interpreted with left grouping:

(a + b) + c

or with right grouping:

a + (b + c)

You might think this doesn't matter, as everyone "knows" + is associative. However, it does matter for the following sorts of reasons:

- In floating point arithmetic, + and * are not associative.

- It is common to group related operators such as + and - together as having the same precedence. Then in a mixed statement such as

  a - b + c

  grouping matters very strongly: (a - b) + c is not the same as a - (b + c). When these operators are arithmetic, left-grouping is the norm.

Let's see how grouping can be brought out in the structure of productions. For an expression that is essentially a sequence of terms separated by '+' signs, there are two possible production sets:

	E → T	*vs.*	E → T
	E → E '+' T		E → T '+' E
or			or
	E → T { '+' T }		E → { T '+' } T
	left grouping		*right grouping*

To see that the production set on the left gives left-grouping, while the one on the right gives right-grouping, compare the forms of the respective derivation trees:

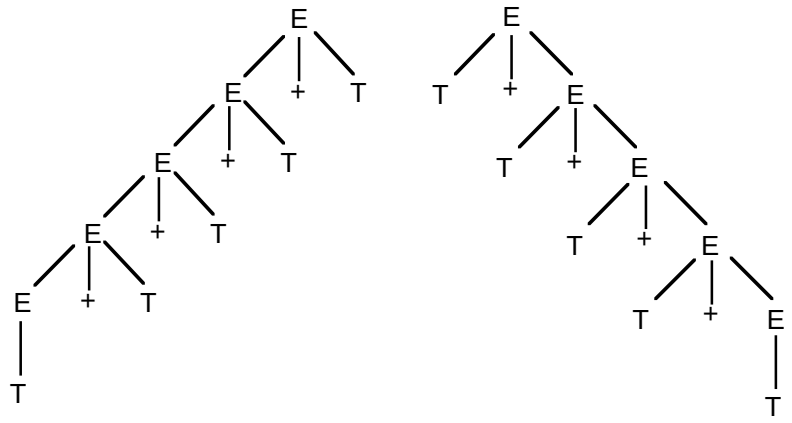


Figure 115: Left and right grouping shown by derivation trees

The *interpretations* of these two trees are:

(((T + T) + T) + T) + T) vs. (T + (T + (T + (T + T))))

respectively. [Do not confuse the parentheses in the above groupings with the meta-syntactic parentheses in the grammar. They are, in some sense, opposite!]

8.8 Programs for Parsing

One function that computer programs have to perform is the interpretation of expressions in a language. One of the first aspects of such interpretation is parsing the expression, to determine a meaning. As suggested earlier, parsing effectively constructs the derivation tree for the expression being parsed. But parsing also involves the rejection of ill-formed input, that is, a string that is not in the language defined by the grammar. In many ways,

this error detection, and the attendant recovery (rather than just aborting when the first error is detected in the input) is the more complex part of the problem.

We will describe one way to structure parsing programs using the grammar as a guideline. This principle is known as "**recursive-descent**" because it begins by trying to match the start symbol, and in turn calls on functions to match other symbols recursively. Ideally, the matching is done by scanning the input string left-to-right without backing up. We will give examples, first using rex rules that correspond closely to the grammar rules, then using Java methods. Before we can do this, however, we must address a remaining issue with the structure of our productions. The production

$$E \rightarrow E\ '+'\ T$$

has an undesirable property with respect to left-to-right scanning. This property is known as "**left recursion**". In the recursive descent method, if we set out to parse an E, we will be immediately called upon to parse an E before any symbols in the input have been matched. This is undesirable, as it amounts to an infinite loop. The production

$$E \rightarrow T\ '+'\ E$$

does not share this problem. With it, we will not try for another E until we have scanned a T and a '+'. However, as we observed, this production changes the meaning to right-grouping, something we don't want to do.

To solve this problem, we already observed that the net effect of the two left-grouping productions can be cast as

$$E \rightarrow T\ \{\ '+'\ T\ \}$$

To represent this effect in a recursive language such as rex, *without* left recursion, we change the productions so that the first T is produced first, then as many of the combination ('+' T) are produced as are needed. We add a new symbol C that stands for "continuation". The new productions are:

$$E \rightarrow T\ C$$
$$C \rightarrow\ '+'\ T\ C$$
$$C \rightarrow \lambda$$

We can see that this is correct, since the last two productions are clearly equivalent to

$$C \rightarrow\ \{\ '+'\ T\ \}$$

and when we substitute for C in $E \rightarrow T\ C$ we get

$$E \rightarrow T\ \{\ '+'\ T\ \}$$

as desired. Informally, under recursive descent, these productions say: "To parse an E, first parse a T followed by a C. To parse a C, if the next symbol is '+', parse another T, then another C. Otherwise return." Evidently, these productions do not have the left-recursion problem. The rules only "recurse" as long as there is another '+' to parse.

An interesting observation in connection with the above rules is that they correspond naturally to a certain while loop:

```
to parse E:
        parse T;
        while( next char is '+' )
                parse T;
```

We will see this connection in action as we look at a parser written in Java.

A Language Parser in Java

We are going to give a Java parser for an arithmetic expression grammar with two operators, '+' and '*', with '*' taking precedence over '+'. This parser simply checks the input for being in the language. Then we will give one that also computes the corresponding arithmetic value.

```
A -> M { '+' M }            sum

M -> V { '*' V }            product

V -> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

To each auxiliary symbol there corresponds a Java method responsible for determining whether the remainder of the input can be generated by that symbol.

```
class addMult extends LineBuffer                // addMult parser class
  {
  // Parse method for A -> M { '+' M }

  Object A()
    {
    Object result;
    Object M1 = M();                            // get the first addend
    if( isFailure(M1) ) return failure;

    result = M1;

    while( peek() == '+' )                      // while more '+'
      {
      nextChar();                               // absorb the '+'
      Object M2 = M();                          // get the next addend
      if( isFailure(M2) ) return failure;
      result = mkTree("+", result, M2);         // create tree
      }
    return result;
    }
```

```
  // Parse method for M -> V { '*' V }

  Object M()
    {
    Object result;
    Object V1 = V();                            // get the first variable
    if( isFailure(V1) ) return failure;

    result = V1;

    while( peek() == '*' )                      // while more '*'
      {
      nextChar();                               // absorb the '*'
      Object V2 = V();                          // get the next variable
      if( isFailure(V2) ) return failure;
      result = mkTree("*", result, V2);         // create tree
      }
    return result;
    }


// Parse method for V -> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

  Object V()
    {
    skipWhitespace();
    if( isVar(peek()) )                         // if there is a variable
      {
      return (new StringBuffer(1).append(nextChar())).toString();
      }
    return failure;
    }

  Object parse()        // TOP LEVEL: parse with check for residual input
    {
    Object result = A();                        // parse an A
    skipWhitespace();                           // ignore trailing whitespace
    if( position < lastPosition )               // see if any residual junk
      {
      System.out.print("*** Residual characters after input: ");
      while( !eof )                             // print residual characters
        {
        char c = nextChar();
        System.out.print(c);
        }
      System.out.println();
      }
    return result;                              // return result of parse
    }


  // isVar indicates whether its argument is a variable

  boolean isVar(char c)
    {
    switch( c )
      {
      case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
      case 'h': case 'i': case 'j': case 'k': case 'l': case 'm': case 'n':
      case 'o': case 'p': case 'q': case 'r': case 's': case 't': case 'u':
      case 'v': case 'w': case 'x': case 'y': case 'z':
        return true;
      default:
        return false;
      }
```

```
    }

  // SUPPORT CODE

  static String promptString = "input: ";               // prompt string

  static ParseFailure failure = new ParseFailure();     // failure

  addMult(String input)                                 // parser constructor
    {
    super(input);                                       // construct LineBuffer
    }

  boolean isFailure(Object ob)                          // test for failure
    {
    return ob instanceof ParseFailure;

  static boolean prompt()                               // prompter
    {
    System.out.print(promptString);
    System.out.flush();
    return true;
    }
  }  // class addMult


// ParseFailure object is used to indicate a parse failure.

class ParseFailure
  {
  }


// LineBuffer provides a way of getting characters from a string
// Note that eof is a variable indicating end-of-line. It should
// not be confused with the eof method of LineBufferInputStream.

class LineBuffer
  {
  String input;                 // string being parsed
  boolean eof;                  // end-of-file condition exists

  int position;                 // position of current character

  int lastPosition;             // last position in input string

  LineBuffer(String input)
    {
    this.input = input;         // initialize variables dealing with string
    position = -1;              // initialize "pointer"
    lastPosition = input.length()-1;
    eof = lastPosition == -1;
    }

  char nextChar()                     // get next character in input
    {
    if( position >= lastPosition )    // if no more characters
      {
      eof = true;                     // set eof
      return ' ';                     // and return a space
      }
    return input.charAt(++position);
    }

  void skipWhitespace()               // skip whitespace
```

```
    {
    while( !eof && peek() == ' ' )
      {
      nextChar();
      }
    }
  }  // LineBuffer


// LineBufferInputStream is an input stream capable of reading one line
// at a time and returning the line as a string, by calling method getLine().
// It also provides the method eof() for testing for end-of-file.
// It extends PushbackInputStream from package java.io

class LineBufferInputStream extends PushbackInputStream
  {
  /**
    * LineBufferInputStream constructs from an InputStream
    */

  LineBufferInputStream(InputStream in)
    {
    super(in);               // call constructor of PushbackInputStream
    }

  /**
    *  getLine() gets the next line of input
    */

  String getLine()
    {
    StringBuffer b = new StringBuffer();         // buffer for line
    try
      {
      int c;
      while( !eof() && ((c = read()) != '\n') )  // read to end-of-line
        {
        b.append((char)c);                        // input.charAt(++position);
        }
      return b.toString();                       // get string from buffer
      }
    catch( java.io.IOException e )
      {
      handleException("getLine", e);
      return "";
      }
    }

  /**
    *  eof() tells whether end-of-file has been reached.
    */

  public boolean eof()
    {
    try
      {
      int c = read();
      if( c == -1 )
        {
        return true;
        }
      else
        {
        unread(c);
        return false;
```

```
      }
    }
  catch( IOException e )
    {
    handleException("eof", e);
    }
  return false;
  }

/**
  *  handleException is called when there is an IOException in any method.
  */

public void handleException(String method, IOException e)
  {
  System.out.println("IOException in LineBufferInputStream: " + e +
                     " calling method " + method);
  }
}
```

Extending the Parser to a Calculator

The following program extends the parse to calculate the value of an input arithmetic expression. This is accomplished by converting each numeral to a number and having the parse methods return a number value, rather than simply an indication of success or failure. We also try to make the example a little more interesting by adding the nuance of parentheses for grouping. This results in mutually-recursive productions, which translate into mutually-recursive parse methods. The grammar used here is:

```
A -> M { '+' M }            sum

M -> U { '*' U }            product

U -> '(' A ')' | N          parenthesized expression or numeral

N -> D {D}                  numeral

D -> 0|1|2|3|4|5|6|7|8|9    digit
```

We do not repeat the definition of class `LineBuffer`, which was given in the previous example.

```
class SimpleCalc extends LineBuffer              // SimpleCalc parser class
  {
  static public void main(String arg[])          // USER INTERFACE
    {
    LineBufferInputStream in = new LineBufferInputStream(System.in);

    while( prompt() && !in.eof() )               // while more input
      {
      String input = in.getLine();               // get line of input
      SimpleCalc parser = new SimpleCalc(input); // create parser
      Object result = parser.parse();            // use parser
      if( result instanceof ParseFailure )       // show result
        System.out.println("*** syntax error ***");
      else
        System.out.println("result: " + result);
      System.out.println();
      }
    System.out.println();
    }


  Object A()                            // PARSE FUNCTION for A -> M { '+' M }
    {
    Object result = M();                // get first addend
    if( isFailure(result) ) return failure;

    while( peek() == '+' )                      // while more addends
      {
      nextChar();
      Object M2 = M();                          // get next addend
      if( isFailure(M2) ) return failure;
      try
        {
        result = Poly.Arith.add(result, M2);    // accumulate result
        }
      catch( argumentTypeException e )
        {
        System.err.println("internal error: argumentTypeException caught");
        }
      }
    return result;
    }


  Object M()                            // PARSE FUNCTION for M -> U { '*' U }
    {
    Object result = U();                // get first factor
    if( isFailure(result) ) return failure;

    while( peek() == '*' )              // while more factors
      {
      nextChar();
      Object U2 = U();                  // get next factor
      if( isFailure(U2) ) return failure;
      try
        {
        result = Poly.Arith.multiply(result, U2);       // accumulate result
        }
      catch( argumentTypeException e )
        {
        System.err.println("internal error: argumentTypeException caught");
        }
      }
    return result;
    }
```

```
Object U()                    // PARSE FUNCTION for U -> '(' A ')' | N
  {
  if( peek() == '(' )         // Do we have a parenthesized expression?
    {
    nextChar();
    Object A1 = A();          // Get what's inside parens
    if( peek() == ')' )
      {
      nextChar();
      return A1;
      }
    return failure;
    }

  return N();                 // Try for numeral
  }


Object parse()        // TOP LEVEL: parse with check for residual input
  {
  Object result = A();
  if( position < lastPosition )
    {
    return failure;
    }
  return result;
  }


static String promptString = "input: ";               // prompt string

static ParseFailure failure = new ParseFailure();     // failure

SimpleCalc(String input)              // constructor for parser
  {
  super(input);                       // construct LineBuffer
  }


boolean isFailure(Object ob)          // test for failure
  {
  return ob instanceof ParseFailure;
  }


static boolean prompt()
  {
  System.out.print(promptString);
  System.out.flush();
  return true;
  }
}  // class SimpleCalc
```

8.9 More on Expression Syntax vs. Semantics

The syntax of expressions in programming languages varies widely from language to language. Less variable is the "semantics" of expressions, that is, the meaning of expressions. In this note, we discuss some of the different syntaxes both the programmer

and the user might encounter. It is important for the student to get used to thinking in terms of semantics apart from specific syntax. In other words, we should try to "see through" the expressions to the conceptual meaning. Acquiring this knack will make the programming world less of a blur.

For our initial comparisons, we shall confine ourselves to arithmetic expressions. We will later discuss more general forms. In arithmetic cases, the **meaning** of an expression is defined as follows:

> Given any assignment of values to each of the variables in an expression, a value is determined by "evaluating" the expression.

In the chapter on Information Structures, we described the idea of bindings. Here we are going to become more precise and make additional use of this idea.

Definition: A **binding** (also called **assignment**) for an expression is a function from each of its variable symbols to a value. [Please do not confuse assignment here with the idea of an *assignment statement*.]

Example Consider the expression A + B. There are two variables, A and B. A binding is any function that gives a value for each of these. For example,

$\{A \rightarrow 3, B \rightarrow 7\}$ is a binding giving a value of 3 to A and 7 to B.
$\{A \rightarrow 9, B \rightarrow 13\}$ is another binding,

and so on.

Definition: The **meaning** of an expression is a function from bindings to values.

Example Considering again the expression A+B, the meaning of A+B is the function that maps

$\{A \rightarrow 3, B \rightarrow 7\} \rightarrow 10$
$\{A \rightarrow 9, B \rightarrow 13\} \rightarrow 22$
etc.

The exact meaning of a given expression is defined recursively, in terms of meta rules. To avoid plunging too deeply into the theory at this time, we will rely on the intuitive meanings of such expressions.

In the chapter on Predicate Logic we will have occasion to extend this idea to other kinds of expressions.

Exercises

1 •• Extend the expression evaluator coded in Java to include the subtraction and division operators, - and /. Assume that these are on the same precedence level as + and *, respectively, and are also left-grouping.

2 •• Give a grammar for floating-point numerals.

3 •• Extend the expression evaluator coded in C++ to include floating-point numerals as operands.

8.10 Syntax-Directed Compilation (Advanced)

A common use of grammars is to control the parsing of a language within a compiler. Software tools that input grammars directly for this purpose are called *syntax-directed compilers* or *compiler generators*. We briefly illustrate a common compiler generator, yacc (*Yet Another Compiler-Compiler*). In yacc, the grammar rules are accompanied by program fragments that indicate what to do when a rule is successfully applied.

A second way to handle the issue of ambiguity is to leave the grammar in ambiguous form, but apply additional control over the applicability of productions. One form of control is to specify a precedence among operators; for example, if there is a choice between using the + operator vs. the * operator, always use the * first. In yacc, such precedence can be directed to the compiler, as can the grouping of operators, i.e. should a + b + c be resolved as (a + b) + c (left grouping) or a + (b + c) (right grouping).

Yacc Grammar for Simple Arithmetic Expressions

Below is the core of a yacc specification of a compiler for the simple arithmetic language. The output of yacc is a C program that reads arithmetic expressions containing numerals, +, and * and computes the result. The yacc specification consists of several productions, and each production is accompanied by a code section that states the action to be performed. These actions involve the symbols $$, $1, $2, etc. $$ stands for the *value* of the expression on the left-hand side of the production, while $1, $2, etc. stand for the value of the symbols in the first, second, etc. positions on the right-hand side. For example, in the production

```
expr:
         expr '+' expr
             { $$ = $1 + $3; }
```

the code fragment in braces says that the value of the expression will be the sum of the two expressions on the right-hand side. ($2 refers to the symbol '+' in this case.) Although the value in this particular example is a numeric one, in general it need not be. For example, an abstract syntax tree constructed using constructors could be used as a

value. In turn, that abstract syntax tree could be used to produce a value or to generate code that does.

```
%left '+'             /* tokens for operators, with grouping          */
%left '*'             /* listed lowest precedence first               */

%start seq            /* start symbol for the grammar                 */

%%                    /* first %% demarks beginning of grammar rules  */

                      /* seq is LHS of first grammar rule.            */

seq :                               /* The "null" event"              */
      expr eol                      /* Parse expression.              */
       { printf("%d\n", $1);}       /* Print value of expr            */
    ;

expr   :
         expr '+' expr
           { $$ = $1 + $3; }        /* forward the sum                */
       | expr '*' expr
           { $$ = $1 * $3; }        /*   etc.                         */
       | number
           { $$ = $1; }
     ;

number :                            /* accumulate numeric value       */
         digit
           { $$ = $1; }
       | number digit
           { $$ = 10 * $1 + $2; }
       ;

digit  :                            /* get digit value                */
          '0' { $$ = 0; }  | '5' { $$ = 5; }
        | '1' { $$ = 1; }  | '6' { $$ = 6; }
        | '2' { $$ = 2; }  | '7' { $$ = 7; }
        | '3' { $$ = 3; }  | '8' { $$ = 8; }
        | '4' { $$ = 4; }  | '9' { $$ = 9; }
         ;

eol     : '\n';                     /* end of line                    */
%%
```

yacc source for a simple arithmetic expression calculator

8.11 Varieties of Syntax

So far, we have mostly considered expressions in "infix form", i.e. ones that place binary (i.e. two-argument) operators between expressions for the operands. As we have discussed, parentheses, implied precedence, and grouping are used to resolve ambiguity. Examples are

A + B
A + B * C
(A + B) * C
A - B - C

There are several other types of syntax that occur either in isolation or in combination with the types we have studied already. These are mentioned briefly in the following sections.

Prefix syntax

Prefix syntax puts the operator before the arguments. It is used in many languages for user-defined functions, e.g.

```
f(A, B, C).
```

The equivalent of the preceding list of expressions in prefix syntax would be:

```
+(A, B)
+(A, *(B, C))
*(+(A, B), C)
-(-(A, B), C)
```

Parenthesis-free Prefix Syntax

This is like prefix syntax, except that no parentheses or commas are used. In order for this form of syntax to work without ambiguity, the **arity** (number of arguments) of each operator must be fixed. For example, we could not use - as both unary minus and binary minus. The running set of expressions would appear as

```
+ A B
+ A * B C
* + A B C
- - A B C
```

Parenthesis-free prefix syntax is used in the Logo language for user-defined procedures.

Expressions in the Prolog language using arithmetic operators can be written in either infix or postfix.

Postfix and Parenthesis-free Postfix

This is like prefix, except the operator comes after the arguments. It is most usually seen in the parenthesis-free form, where it is also called **RPN (reverse Polish notation)**.

(A, B)+	A B +
(A, (B, C)*)+	A B C * +
((A, B)+, C)*	A B + C *

((A, B)-, C)- A B - C -

The parenthesis-free version of this syntax also requires the property that each operator must be of fixed arity.

S Expression Syntax

A grammar for S expressions was presented earlier. The notion of S expressions provide a syntax that, despite its initially foreign appearance, has several advantages:

> The arity of operators need not be fixed. For example, we could use a + operator that has any number of arguments, understanding the meaning to be compute the sum of these arguments. [For that matter, prefix and postfix **with parentheses required** also have this property.]
>
> The number of symbols is minimal and a parsing program is therefore relatively easy to write.

In S-expression syntax, expressions are of the form

> (*operator argument*-1 *argument*-2 .... *argument*-N)

where the parentheses are required. S-expressions are therefore a variant on prefix syntax. Sometimes S-expressions are called "Cambridge Polish" in honor of Cambridge, Mass., the site of MIT, the place where the language Lisp was invented. Lisp uses S-expression syntax exclusively.

> **In S expressions, additional parentheses cannot be added without changing the meaning of the expression.** Additional parentheses are never necessary to remove ambiguity because there is never any ambiguity: every operator has its own pair of parentheses.

Example: Arithmetic Expressions in S Expression Syntax

```
( + A B)
( + A ( * B C ) )
( * (+ A B) C)
( - ( - A B ) C )
```

S expressions are not only for arithmetic expressions; they can be used for representing many forms of structured data, such as heterogeneous lists. For example, the heterogeneous list used in an earlier rex example

```
[ my_dir,
  mail,
  [ archive, old_mail, older_mail],
  [ projects,
    [ sorting, quick_sort, heap_sort],
    [ searching, depth_first, breadth_first]
  ],
  [games, chess, checkers, tic-tac-toe]
]
```

could be represented as the following S expression:

```
( my_dir
  mail
  ( archive old_mail older_mail)
  ( projects
    ( sorting quick_sort heap_sort)
    ( searching depth_first breadth_first)
  )
  (games chess checkers tic-tac-toe)
)
```

S expressions are, in many ways, ideal for representing languages directly in their abstract syntax. The syntax is extremely uniform. Moreover, given that we have a reader for S expressions available, we don't need any other parser. We can analyze a program by taking the S expression apart by recursive functions. This is the approach taken in the language Lisp. Every construct in the language can play the role of an abstract syntax constructor. This constructor is identified as the first argument in a list. The other arguments of the constructor are the subsequent items in the list.

We saw above how arithmetic expressions are represented with S expressions. Assignment in Lisp is represented with the *setq* constructor:

`(setq` *Var Expression* `)`

sets the value of variable *Var* to that of expression *Expression*. A two-armed conditional is represented by a list of four elements:

`(if` *Condition T-branch F-branch*`)`

and function definitions are represented by the constructor *defun*. For example, the following is a factorial program in Lisp:

```
(defun factorial (N)
 (if ( < N 1 )
   1
   (* N (factorial (- N 1) ) ) ) )
```

In the next major section, we give examples of extendible interpreters that exploit the S expression syntax.

Expression-Tree Syntax

Expression trees are trees in which the leaves are labeled with variables or constants and the interior nodes are labeled with operators. The sub-trees of an operator node represent the argument expressions. The advantage of trees is that it allows us to visualize the flow of values that would correspond to the evaluation of an expression. These correspond closely to the abstract syntax trees discussed earlier.

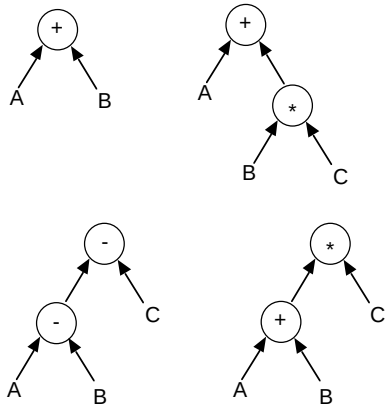


Figure 116: Expression-tree syntax for the equivalents of infix A+B, A+(B*C), (A+B)*C, and (A-B)-C (reading clockwise from upper-left)

DAG (Directed Acyclic Graph) Syntax

DAG syntax is an extension of expression-tree syntax in which the more general concept of a DAG (Directed, Acyclic, Graph) is used. The added benefit of DAGs is that sharing of common sub-expressions can be shown. For example, in

(A + B) * C + D / (A + B)

we might want to indicate that the sub-expression A + B is one and the same. This would be shown by having two arcs leaving the node corresponding to A + B. Note that is related to the issue of structure sharing which has appeared several times before.

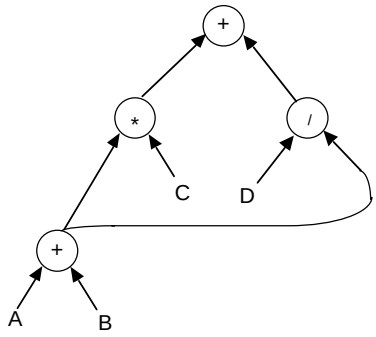


Figure 117: DAG for an arithmetic expression

DAG syntax is a subset of what is used in dataflow languages. Typically the latter allow loops as well, assuming that an appropriate semantics can be given to a loop, and are therefore not confined to DAGs (they can be cyclic). A typical example of languages that make use of loops are those for representing signal-flow graphs, as occur in digital signal processing.

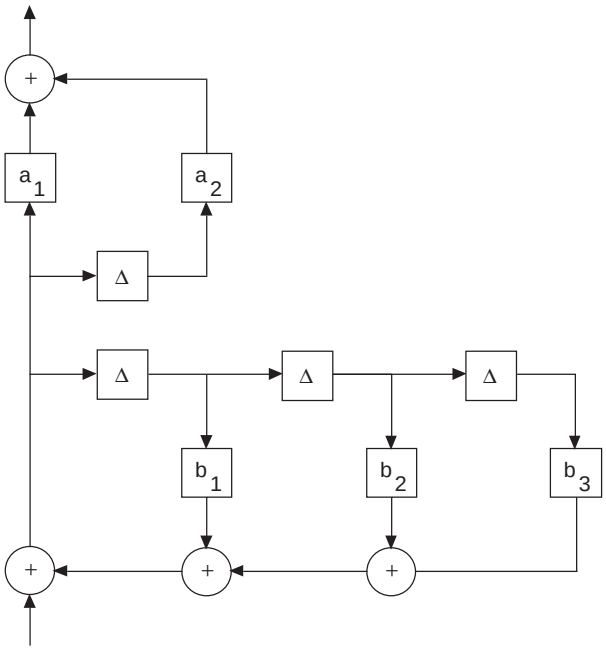


Figure 118: Graphical syntax for a digital filter

In the jargon of digital filters, a filter with loops is called "recursive".

The Spreadsheet Model

The spreadsheet model is a relatively user-friendly computing model that implicitly employs DAGs to represent computations. In a spreadsheet, the computational framework is represented in a two-dimensional array of cells. Each cell contains either a primitive data value, such as an integer or string, or an expression. An expression is typically an arithmetic or functional expression entailing operators and the names of other cells as variables. The spreadsheet system computes the values in cells containing expressions, and displays the value in the cell itself. Whenever the user changes one of the primitive data values in its cell, the relevant cells containing expressions are changed by the system to reflect the new value. The underlying mechanism is that of a DAG, where some nodes are identified with cells.

For simplicity, consider a 2 x 3 spreadsheet, as shown below. The cells in the spreadsheet are designated A1, A2, A3, B1, B2, B3.

	1	2	3
A	B1 * B2 + B3 / B1	value of A	value of B
B	A2 + A3	value of C	value of D

Figure 119: A simple spreadsheet representable as a DAG

Cell A1 of this spreadsheet represents the value of the previous DAG expression, if we associate the values A, B, C, and D with A2, A3, B2, and B3 respectively. The expression in cell B1 represents the common sub-expression shown in the DAG.

Typical states of the spreadsheet, with values entered by the user for A, B,C, and D, would show as follows:

	1	2	3
A	32.75	3	5
B	8	4	6

	1	2	3
A	50.7	4	6
B	10	5	7

Figure 120: Two different states of the simple spreadsheet

Exercises

1 ••• Give unambiguous grammars for representing arithmetic expressions with operators + and *:

(a) In postfix form

(b) In prefix form

(c) In S-expression form

2 •• Build an evaluator for expressions in postfix form. Although it could be done with recursion, such an evaluator is typically described using a "stack" data structure that retains operand values. Values are removed in the order of their recency of insertion:

> When an operand is scanned, it is pushed on the stack.
>
> When an operator is scanned, the appropriate number of arguments is removed from the stack, the operator is applied, and the result pushed on the stack.

Such is the principle on which RPN calculators work. It is also the basis of the FORTH language, and the PostScript™ language that is built on FORTH syntax.

3 ••• Build an evaluator for expressions in postfix form that uses no explicit stack. (An implicit stack is used in the recursive calls in this evaluator). [Hint: This can be done by defining recursive functions having arguments corresponding to the top so many values on the stack, e.g. there would be three functions, assuming at most 2-ary operators). When such a function returns, it corresponds to having used those two arguments.]

4 •• Having done the preceding two problems, discuss tradeoffs of the two versions with respect to program clarity, tail recursion, space efficiency, etc.

5 ••• Build an evaluator for expressions in prefix form.

6 •••• Devise an expression-based language for specifying graphics. Develop a translator that produces PostScript™. Display the results on a laser printer or workstation screen.

7 •••• Develop a spreadsheet calculator, including a means of parsing the expressions contained within formula cells.

8 •••• Explore the ergonomics of copying formulas in cells, as are done in commercial spreadsheet implementations. After devising a good explanation (or a better scheme), implement your ideas.

9 ••• Implement a system simplifying arithmetic expressions involving the operators +, -, *, /, and *exp* (exponent) with their usual meanings. For convenience, use S expression input syntax, wherein each expression is either:

A numeric constant

An atom, representing a variable

A list of the form (Op E1 E2), where E1 and E2 are themselves expressions and Op is one of the four operators.

The kinds of operations involved in simplification would include:

removing superfluous constants such as 1's and 0's:

(+ E1 0) → E1

(* E1 1) → E1

carrying out special cases of the operators symbolically:

(- E1 E1) → 0

(*exp* E 0) → 1

eliminating / in favor of *:

(/ (/ A B) C) → (/ A (* B C))

etc.

For example, if your system is given the input expression

```
(/ (/ A (* B (- C C)) D)
```

it would produce

```
(/ A (* B D))
```

These kinds of simplifications are included automatically in computer algebra systems such as Mathematica™ and Maple, which have a programming language component. However, the user is usually unable to customize, in an easy way, the transformations to suit special purposes.

10 •••• Implement a system in the spirit of the previous exercise, except include trigonometric identities.

11 •••• Implement a system in the spirit of the previous exercise, except include integrals and derivatives.

8.12 Chapter Review

Define the following terms:

abstract syntax tree
ambiguity
assignment
auxiliary symbol
basis
binding
countable
countably-infinite
DAG syntax
derivation tree
expression tree
finite
grammar
hypercube
induction rule
inductive definition
infinite
language
left recursion
natural numbers
parsing
precedence
prefix vs. postfix syntax
production
recursive descent
semantics
S expression
spreadsheet syntax
start symbol
syntax
syntax diagram
terminal alphabet

8.13. Further Reading

•• Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1985. [Introduces the PostScript™ language, which uses RPN syntax.]

••• A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986. [Demonstrates how grammars are used in the construction of compilers for programming languages.]

• Alan L. Davis and Robert M. Keller, *Data Flow Program Graphs*, Computer, February, 1982, pages 26-41. [Describes techniques and interpretations of data flow programs.]

• Bob Frankston and Dan Bricklin, *VisiCalc*, VisiCalc Corporation, 1979. [The first spreadsheet.]

•• Paul R. Halmos, *Naive Set Theory*, Van Nostrand, Princeton, New Jersey, 1960. [Develops set theory from the ground up.]

• Brian Harvey, *Computer science Logo style*, MIT Press, 1985. [One of several possible references on the Logo language.]

•• S.C. Johnson, *Yacc – Yet Another Compiler-Compiler*, Computer Science Tech. Rept. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975. [Original reference on yacc.]

•• John R. Levine, Tony Mason, and Doug Brown, *lex & yacc*, O'Reilly & Associates, Inc., Sebastpol, CA, 1990.

••• J. McCarthy, *et al., Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass., 1965. [The original reference on Lisp list processing. Moderate.]

••• J. McCarthy and J.A. Painter, *Correctness of a compiler for arithmetic expressions*, in J.T. Schwartz (ed.), Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science, pp. 33-41, American Mathematical Society, New York, 1967. [The first mention of "abstract syntax".]

•• C.H. Moore, F*ORTH: A New Way to Program Minicomputers*, Astronomy and Astrophysics, Suppl. No. 15, 1974. [Introduces the FORTH language.]

9. Proposition Logic

9.1 Introduction

This chapter describes proposition logic and some of the roles this form of logic plays in computer hardware and software. From early on, computers have been known as “logic machines”. Indeed, “logic” plays a central role in the design, programming, and use of computers. Generally, “logic” suggests a system for reasoning. But in computer science, reasoning is only one use of logic. We also use logic in a fairly mechanistic way in the basic construction of computers. This form of logic is called “proposition logic”, “switching logic”, or sometimes (not quite correctly) “Boolean algebra”. There are also several other, much more powerful, types of logic that are used in other aspects of computer science. “Predicate logic”, also called “predicate-calculus” or “first-order logic” is used in programming and in databases. Predicate logic, and “temporal logic”, which is built upon predicate logic, are used for reasoning about programs and dynamic systems. Varieties of “modal logic” are used in building artificial intelligence reasoning systems.

In this course, we will be concerned mostly with proposition logic, and to some extent, and mostly informally, predicate logic. Proposition logic is used in the design and construction of computer hardware. It is also related to a simple kind of deductive reasoning. In proposition logic, we deal with relationships among variables over a **two-valued domain**, for these reasons:

- It is simplest to build high-speed calculating devices out of elements that have only two (as opposed to several) stable states, since such devices are the simplest to control. We need only to be able to do two things:
    - sense the current state of the device
    - set the state of the device to either of the two values
- All finite sets of values (such as the control states of a Turing machine or other computer) can be *encoded* as combinations (“tuples”) of two-valued variables.

In this book we shall mostly use 0 and 1 as our two values, although any set of two values, such as true and false, yes and no, *a* and *b*, red and black, would do. When we need to think in terms of truth, we will usually use 1 for true and 0 for false.

9.2 Encodings of Finite Sets

As mentioned above, every finite set can be encoded into tuples of 0 and 1. These symbols are usually called "bits". Technically the word "bit" stands for "binary digit", but it is common to use this term even when we don't have a binary or base-2 numeral system in mind. More precisely, an **encoding** of a set S is a one-to-one function of the form

$$S \to \{0, 1\}^N$$

for some N. Here $\{0, 1\}^N$ means the set of all N-tuples of 0 and 1. A given tuple in this context is called a "codeword". Remember that "one-to-one" means that no two elements of S map to the same value. This is necessary so that a codeword can be decoded unambiguously. Since $\{0, 1\}^N$ has exactly 2^N elements, in order for the one-to-one property to hold, N would therefore have to be such that

$$2^N \geq |S|$$

where we recall that |S| means the **size**, or number of elements, of S. Put another way, N is an integer such that

$$N \geq \log_2 |S|.$$

This inequality still gives us plenty of leeway in choosing encodings. There are many considerations that come into play when considering an encoding, and this motivates the use of different encodings in different situations. A fair amount of programming ends up being conversion of one encoding to another. Some of the considerations involved in the choice are:

- *Conciseness*: a code that uses as few symbols as possible.
- *Ease in decoding*: a code that is simple for humans, or circuitry, to decode.
- *Difficulty in decoding*: a code that is hard to decode, for example, one to be used in encrypting data for security purposes.
- *Error detection*: a code designed in such a way that if one of the bits changes inadvertently, this fact can be detected.
- *Error correction*: like error detection, except that we can determine the original value, as well as determining whether a change occurred.
- Other special properties, such as the "one change" property found in Gray codes, as discussed below.

We already encountered the binary numerals in our earlier discussion of encoding the infinite set of natural numbers. Obviously, the same idea can be used to encode a finite

set. It is common to use a fixed number of bits and include all of the leading zeroes when encoding a finite set.

Binary Code Example

Encode the set {0, 1, 2, 3, 4, 5, 6, 7} in binary: We give the encoding by a table showing the correspondence between elements of the set and $\{0, 1\}^3$, the set of all 3-tuples of bits:

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Figure 121: The binary encoding of {0, ..., 7}

It is easy to see that this is a code, i.e. that the correspondence is one-to-one.

In general, N bits is adequate to encode the set of numbers $\{0, ..., 2^N-1\}$. There are many other ways to encode this same set with the same number of bits. Not all have any particular pattern to them. One that does is the following:

Gray Code Example

A Gray Code is also called “reflected binary”. Encoding the same set as above, this code starts like binary:

0	000
1	001

However, once we get to 2, we change the second bit from the right only:

1	001	
2	011	(instead of 010 as in straight binary)

In general, we wish to change only one bit in moving from the encoding from a number K to K+1. The trick is to do this without returning to 000 prematurely. By using the pattern of “reflecting” a certain number of bits on the right, we can achieve coverage of all bit patterns while maintaining the one-change property.

```
0       000
1       001 ↑
          ---     last bit reflected above and below
2       011 ↓ read up above the line, copy bit downward
3       010
          -----   last two bits reflected
4       110
5       111
6       101
7       100
0       000
```

Figure 122: The Gray encoding of {0, ..., 7}

One-Hot Code Example

This code is far from using the fewest bits, but is very easy to decode. To encode a set of N elements, it uses N bits, only one of which is 1 in any codeword. Thus, to encode {0, ..., 5}, a one-hot code is:

```
0       000001
1       000010
2       000100
3       001000
4       010000
5       100000
```

Figure 123: A one-hot encoding of {0, ..., 5}

Examples of one-hot codes include a push-button telephone (one out of 12) and a standard traffic light (one out of 3). An electronic piano keyboard would not be one-hot, because it allows chords to be struck.

Subset Code Example

This code is useful when the set to be encoded is, in fact, the set of all subsets of a given set. If the latter has N elements, then exactly 2^N elements are in the set of subsets. This suggests an N-bit code, where one bit is used to represent the presence of each distinct element. For example, if the set is {a, b, c}, then the set of all subsets is {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}. The code would appear as:

{}	000
{a}	100
{b}	010
{c}	001
{a, b}	110
{a, c}	101
{b, c}	011
{a, b, c}	111

Figure 124: An encoding of all subsets of {a, b, c}

Note that we could use this code and the binary code to establish a relationship between subsets and the numbers $\{0, ..., 2^N-1\}$:

number	**subset**	**binary**
0	{}	000
1	{c}	001
2	{b}	010
3	{b, c}	011
4	{a}	100
5	{a, c}	101
6	{a, b}	110
7	{a, b, c}	111

Figure 125: Correspondence between binary encoding subset encoding

This consideration is of importance in the construction of computers and also in programming, as will be seen later.

The Pascal language has a built-in subset code feature in the form of a *set* type, which can be derived from any enumeration type.

Binary-Coded Decimal Example

This code, abbreviated BCD, is sometimes used to make it easy to decode into decimal representations. The number to be encoded is first represented in decimal (radix 10). Each digit is separately coded in 4-bit binary. The resulting 4-bit codes are concatenated to get the codeword. For example, the number 497 would encode as 0100 1001 0111.

The Cartesian Encoding Principle

The BCD encoding illustrates a general principle: We can achieve economy in the description of an encoding when we can decompose the set to be encoded as a Cartesian Product of smaller sets. In this case, we can separately encode each set, then take the

overall code to be a tuple of the individual codes. Suppose that the set S to be encoded contains M elements, where M is fairly large. Without further decomposition, it would take a table of M entries to show the encoding. Suppose that S can be represented as the Cartesian product T x U, where T contains N and U contains P elements. Then M = NP. Knowing that we are using a product encoding, we need only give two tables, one of size N and the other of size P, respectively, to present the encoding. For example, if N and P are roughly the same, the table we have to give is on the order of 2 times the square root of M, rather than M. For large M this can be a substantial saving.

Error-Correcting Code Principle (Advanced)

Richard W. Hamming invented a technique that provides one basis for a family of error-correcting codes. Any finite set of data can be encoded by adding sufficiently many additional bits to handle the error correction. Moreover, the number of error correction bits added grows as the logarithm of the number of data bits.

The underlying principle can be viewed as the multiple application of *parity bits*.

> With the parity bit scheme, a single bit is attached to the transmitted data bits so that the sum modulo 2 (i.e. the exclusive-or) of all bits is always 0. In this way, the corruption of any single bit, *including the parity bit itself*, can be detected. If there are N other bits, then the parity bit is computed as $b_1 \oplus b_2 \oplus .... \oplus b_N$, where $\oplus$ indicates modulo-2 addition (defined by $0 \oplus 1 = 1 \oplus 0 = 1$, and $0 \oplus 0 = 1 \oplus 1 = 0$). If the sum of the bits is required to be 0, this is called "even parity". If it is required to be 1, it is called "odd parity".

The Hamming Code extends the parity error- detection principle to provide single-bit error *correction* as follows: Designate the ultimate codewords as $\{c_0, c_1, c_2, ..., c_K\}$. (We haven't said precisely what they are yet.) Suppose that N is the number of bits used in the encoding. Number the bits of a generic codeword as $b_1, b_2, b_3, ...$ The code is to be designed such that the sum of various sets of bits of each codeword is always 0. In particular, for each appropriate i, the sum of all bits having 1 as the i^{th} bit of their binary expansion will be 0 in a proper codeword. In symbolic terms, supposing that there are 7 bits in each codeword, the code requires the following even parities:

$$b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 0$$
$$b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0$$
$$b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0$$
$$....$$

There is a simple way to guarantee that these properties hold for the code words: Reserve bits b_1, b_2, b_4, b_8, etc. as parity bits, leaving the others $b_3, b_5, b_6, b_7, ...$ for the actual data. Observe that each equation above entails only one parity bit. Hence each equation can be *solved* for that bit, thus determining the parity bits in terms of the data bits:

$b_1 = b_3 \oplus b_5 \oplus b_7$
$b_2 = b_3 \oplus b_6 \oplus b_7$
$b_4 = b_5 \oplus b_6 \oplus b_7$
....

To construct a Hamming code for M data bits, we would begin allocating the bits between parity (the powers of 2) and data, until M data bits were covered. In particular, the highest order bit can be a data bit. For example, if M = 4 data bits were desired, we would allocate b_1 as parity, b_2 as parity, b_3 as data, b_4 as parity, b_5 as data, b_6 as data, and b_7 as data. The index numbered 7 is the least index that gives us 4 data bits. We construct the code by filling in the data bits according to the ordinary binary encoding, then determining the parity bits by the equations above. The result for M = 4 is shown in the table below. Note that it takes two bits of parity to provide error-correcting support for the first bit of data, but just one more bit of parity to provide support for three more bits of data.

		data	**data**	**data**	**parity**	**data**	**parity**	**parity**
decimal	**binary**	**b_7**	**b_6**	**b_5**	b_4	**b_3**	b_2	b_1
0	0000	**0**	**0**	**0**	0	**0**	0	0
1	0001	**0**	**0**	**0**	0	**1**	1	1
2	0010	**0**	**0**	**1**	1	**0**	0	1
3	0011	**0**	**0**	**1**	1	**1**	1	0
4	0100	**0**	**1**	**0**	1	**0**	1	0
5	0101	**0**	**1**	**0**	1	**1**	0	1
6	0110	**0**	**1**	**1**	0	**0**	1	1
7	0111	**0**	**1**	**1**	0	**1**	0	0
8	1000	**1**	**0**	**0**	1	**0**	1	1
9	1001	**1**	**0**	**0**	1	**1**	0	0
10	1010	**1**	**0**	**1**	0	**0**	1	0
11	1011	**1**	**0**	**1**	0	**1**	0	1
12	1100	**1**	**1**	**0**	0	**0**	0	1
13	1101	**1**	**1**	**0**	0	**1**	1	0
14	1110	**1**	**1**	**1**	1	**0**	0	0
15	1111	**1**	**1**	**1**	1	**1**	1	1

Figure 126: The Hamming code for 4 bits of data, requiring a total of 7 bits. The bold-face bits represent data bits. The plain-face bits are determined from the data bits by the parity equations.

For example, consider row 14. The data bits are 1110 = $b_3\ b_5\ b_6\ b_7$. According to our equations,

$b_1 = b_3 \oplus b_5 \oplus b_7 = 0 \oplus 1 \oplus 1 = 0$
$b_2 = b_3 \oplus b_6 \oplus b_7 = 0 \oplus 1 \oplus 1 = 0$
$b_4 = b_5 \oplus b_6 \oplus b_7 = 1 \oplus 1 \oplus 1 = 1$

Thus we have assigned in row 14 $b_4b_2b_1 = 1\ 0\ 0$.

Error-correction rule: A word in the Hamming code is error-free (i.e. is a code-word) iff each parity equation holds. Thus, given the word received, compute the sums

$$b_1 \oplus b_3 \oplus b_5 \oplus b_7 = s_0$$
$$b_2 \oplus b_3 \oplus b_6 \oplus b_7 = s_1$$
$$b_4 \oplus b_5 \oplus b_6 \oplus b_7 = s_2$$
$$....$$

If any of these is non-zero, then there is an error. A clever part of Hamming's design is that the sums $s_2s_1s_0$, when interpreted as a binary numeral, **indexes the bit that is incorrect**. For example, consider the codeword for 12:

1 **1** **0** 0 **0** 0 1

Suppose that bit 2 gets changed, resulting in:

1 **1** **0** 0 **0** 1 1

Then $s_2s_1s_0$ will be 0 1 0, indicating b_2 is incorrect. On the other hand, if bit 6 were to change, resulting in

1 **0** **0** 0 **0** 0 1

$s_2s_1s_0$ will be 1 1 0, indicating b_6 is incorrect.

The total number of encodings possible with 7 bits is $2^7 = 128$. For each 7-bit code, we need 7 other codes that translate to the same data word, i.e. 8 codes per group. Fortunately $8 * 16 \leq 128$.

We can visualize what is going with Hamming codes by using a hypercube, a recurrent theme in computer science. To do so, we will use a smaller example. Below is shown a 3-dimensional hypercube. The connections on the hypercube are between points that differ by only one bit-change, Hamming distance 1, as we say. To make an error-correcting code, we need to make sure that no code differs by one-bit change from more than one other code. The diagram shows an error-correcting code for one data bit. The dark nodes 000 and 111 are representative code words for 0 and 1 respectively. The nodes with arrows leaving encode the same data as the nodes to which they point. In order for this to be error correcting, each node must be pointed to by all the nodes Hamming distance1 away, and no node can point to more than one representative.

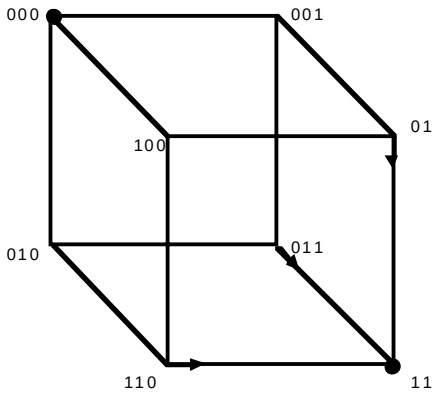


Figure 127: Three-dimensional hypercube for an error-correcting code

The above code is "efficient" in the sense that no node goes unused. If we move to a four-dimensional hypercube, we could try to encode more than one data bit, for example two data bits. A feasibility test for this encoding is that there must be five code words in each group: the main representative and its four immediate neighbors (to account for all 1-bit changes). Also, there are four different combinations of two data bits, so we need at least 5 * 4 == 20 distinct code words. But there are only 16 code words in a four-dimensional hypercube, so with it we cannot do better than one data bit, the same as with a three-dimensional hypercube. We need a 5-dimensional hypercube to achieve error-correction with 2 data bits.

9.3 Encodings in Computer Synthesis

Many problems in synthesis of digital systems involve implementing functions on finite sets. The relationship of this to proposition logic, discussed in the following section, is the following:

> **Combinational Switching Principle**
>
> Once an encoding into bits has been selected for the finite sets of interest, the implementation of functions on those sets reduces to the implementation of functions on bits.

Addition Modulo 3 Example

Suppose we wish to represent the function "addition modulo 3" in terms of functions on bits. Below is the definition table for this function.

		b		
	a + b mod 3	**0**	**1**	**2**
	0	0	1	2
a	**1**	1	2	0
	2	2	0	1

We select an encoding for the set {0, 1, 2}. Let us choose the binary encoding as an example:

element	encoded element
0	00
1	01
2	10

We then transcribe the original table by replacing each element with its encoding, to get an image of the table using bit encodings, calling the encoding of *a* uv, and the encoding of *b* wx:

		wx		
		0 0	**0 1**	**1 0**
	0 0	0 0	0 1	1 0
uv	**0 1**	0 1	1 0	0 0
	1 0	1 0	0 0	0 1

We then separate this table into two functions, one for each resulting bit value, where we use [u, v] for the bit encodings of *a* and [w, x] for the bit encodings of *b*.

		wx		
		0 0	**0 1**	**1 0**
	0 0	0 0	0 1	1 0
uv	**0 1**	0 1	1 0	0 0
	1 0	1 0	0 0	0 1
		^	^	^

Carats point to "first result bits", used to construct following table.

		wx		
	f_1	0 0	0 1	1 0
	0 0	0	0	1
uv	0 1	0	1	0
	1 0	1	0	0

Table for the first result bit of encoded modulo 3 addition.

		wx		
	f_2	00	01	10
	00	0	1	0
uv	01	1	0	0
	10	0	0	1

Table for the second result bit of encoded modulo 3 addition.

Each table is then a function on 4 bits, two from the side stub and two from the upper stub, i.e. we have modeled the original function of the form $\{0, 1, 2\}^2 \rightarrow \{0, 1, 2\}$ as two functions of the form $\{0, 1\}^4 \rightarrow \{0, 1\}$. We can compute a + b mod 3 in this encoding by converting a and b to binary, using the two tables to find the first and second bits of the result, then convert the result back to the domain {0, 1, 2}.

Let's try to model this process in rex. The encoded domain will be represented as lists of two elements, each 0 or 1. We will give a specification for the following functions:

encode: $\{0, 1, 2\} \rightarrow \{0, 1\}^2$ is the encoding of the original domain in binary

add: $\{0, 1, 2\}^2 \rightarrow \{0, 1, 2\}$ is the original mod 3 addition

f_1: $\{0, 1\}^4 \rightarrow \{0, 1\}$ is the function for the first bit of the result

f_2: $\{0, 1\}^4 \rightarrow \{0, 1\}$ is the function for the second bit the result

We expect the following relation to hold, for every value of *a* and *b* in {0, 1, 2}:

encode(add(a, b)) == [f_1(append(encode(a), encode(b))), f_2(append(encode(a), encode(b)))];

We can write a program that checks this. The rex rules for the above relations are:

```
add(0, 0) => 0;    add(0, 1) => 1;    add(0, 2) => 2;
add(1, 0) => 1;    add(1, 1) => 2;    add(1, 2) => 0;
add(2, 0) => 2;    add(2, 1) => 0;    add(2, 2) => 1;

encode(0) => [0, 0];        encode(1) => [0, 1];        encode(2) => [1, 0];

f1([0, 0, 0, 0]) => 0;      f1([0, 0, 0, 1]) => 0;      f1([0, 0, 1, 0]) => 1;
f1([0, 1, 0, 0]) => 0;      f1([0, 1, 0, 1]) => 1;      f1([0, 1, 1, 0]) => 0;
f1([1, 0, 0, 0]) => 1;      f1([1, 0, 0, 1]) => 0;      f1([1, 0, 1, 0]) => 0;

f2([0, 0, 0, 0]) => 0;      f2([0, 0, 0, 1]) => 1;      f2([0, 0, 1, 0]) => 0;
f2([0, 1, 0, 0]) => 1;      f2([0, 1, 0, 1]) => 0;      f2([0, 1, 1, 0]) => 0;
f2([1, 0, 0, 0]) => 0;      f2([1, 0, 0, 1]) => 0;      f2([1, 0, 1, 0]) => 1;
```

The test program could be:

```
test(A , B ) =>
    encode(add(A , B )) == [f1(append(encode(A ), encode(B ))), f2(append(encode(A ), encode(B )))];
```

```
test_all(_) =>
        (test(0, 0), test(0, 1), test(0, 2),
         test(1, 0), test(1, 1), test(1, 2),
         test(2, 0), test(2, 1), test(2, 2)),
        "test succeeded";

test_all() => "test failed";
```

The first rule for *test_all* has one large guard. If any test fails, then that rule is inapplicable and we use the second rule.

Although the above method of implementing modulo 3 addition is not one we would use in our everyday programming, it is used routinely in design of digital circuits. The area of proposition logic is heavily used in constructing functions on bits, such as f_1 and f_2 above, out of more primitive elements. We turn our attention to this area next.

Exercises

1 •• Suppose that we chose a different encoding for {0, 1, 2}: $0 \rightarrow 00$, $1 \rightarrow 10$, $2 \rightarrow 11$. Construct the corresponding bit functions f_1 and f_2 for modulo-3 addition.

2 •• Choose an encoding and derive the corresponding bit functions for the *less_than* function on the set {0, 1, 2, 3}.

3 •• If A is a finite set, use |A| to denote the *size* of A, i.e. its number of elements (also called the *cardinality* of A). Express $|A \times B|$ in terms of |A| and |B|.

4 •• Express $|A_1 \times A_2 \times .... \times A_N|$ in terms of the N quantities $|A_1|$, $|A_2|$, ... $|A_N|$.

5 ••• Let A^B denote the *set of all functions* with B as domain and A as co-domain. Supposing A and B are finite, express $|A^B|$ in terms of |A| and |B|.

6 • What is the fewest number of bits required to encode the English alphabet, assuming that we use only lower-case letters? What if we used both lower and upper case? What if we also included the digits 0 through 9?

7 ••• For large values of N, how does the number of bits required to encode an N element set in binary-coded decimal compare to the number of bits required for binary?

8 •• Show that a Gray code can be constructed for any set of size 2^N. [Hint: Use induction.]

9 ••• Devise a rex program for the function *gray* that, given N > 0, will output a Gray code on N bits. For example, gray(3) ==> [[0,0,0], [0,0,1], [0,1,1], [0,1,0], [1,1,0], [1,1,1], [1,0,1], [1,0,0]].

10 ••• Devise a rex program that will "count" in Gray code, in the sense that given a codeword in the form of a list, it will produce the next codeword in sequence. For example, gray_count([1,1,1]) ==> [1,0,1], *etc*. [Hint: Review the Chinese ring puzzle in *Compute by the Rules*.]

9.4 Propositions

By a "proposition" we mean a statement or condition that can be one of 0 ("false") or 1 ("true"). In computer science, there are two primary ways in which we deal with propositions:

- Expressions that contain proposition variables and logical operators
- Functions, the arguments of which range over proposition values.

These two ways are closely related, but sometimes it is more natural to work with expressions while other times it is simpler to work with functions.

Examples

Let us give some variables representing propositions:

a: Harvey Mudd College is in Claremont.

b: Disneyland is in Claremont.

c: It is raining in Claremont.

Each of these statements can be assigned a truth value, 0 or 1. It turns out that it only makes sense to assign *a* the value 1 and *b* the value 0, but this is irrelevant since proposition logic is concerned mostly with relationships between *hypothetical* truth values. These relationships are expressed by propositional operations or functions. In expressions, we usually deal with 1-ary or 2-ary operators, whereas we can deal with functions of arbitrary arity.

The propositional operators are sometimes called "connectives". A typical example of a connective is $\wedge$, read "and". For example, with the above interpretation,

$b \wedge c$

would stand for:

"Disneyland is in Claremont and it is raining in Claremont."

More than wanting to know whether this overall statement is true or not, we want to know how its truth depends on the truth of the constituent proposition variables *b* and *c*. This can be succinctly described by giving the value of the statement for all possible values of b and c in the form of a function table. We have already used such tables before. When we are dealing with functions on propositions or bits, the table is called a "truth table". Such a table can appear many ways. Two common ways are (i) with a stub enumerating all assignments to b and c in a 1-dimensional array, or (ii) with separate stubs for b and c, in a 2-dimensional array.

b	**c**	**b∧c**
0	0	0
0	1	0
1	0	0
1	1	1

Figure 128: Representation (i) of ∧

b∧c	**c**		
		0	**1**
b	**0**	0	0
	1	0	1

Figure 129: Representation (ii) of ∧

Any 2-ary function on the truth values can be described by such a table, and any such table describes a function. Since there are 4 different possible assignments of 0 and 1 to b and c, and each of these can be assigned a value, either 0 or 1, there are $2^4 = 16$ different 2-ary truth functions. Rather than present a separate table with stubs for each, we can use form (i) above with a single stub and show all 16 functions.

args		**Function Number**															
b	c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		Value of Function given Arguments															

Figure 130: The sixteen proposition logic functions of 2 arguments.

Notice that in providing this **numbering** of the functions, we are using one of our encoding principles mentioned earlier. That is, an encoding of the set of functions is implied in our presentation. Moreover, we also see that there is a correspondence

between the set of 2-ary functions and the set of subsets of a four element set. This will be useful later on when we deal with the need to simplify the presentation of functions, which will correspond to minimizing the hardware in a digital circuit. Finally note that the number we give for each function corresponds to the decoded binary represented by the corresponding column, when read most-significant bit at the top.

The following is a description of functions "in order of importance". For some of the functions, we give mnemonic rules for help in remembering when the function gives value 1.

Function	also written	Explanation	Mnemonics
f_0 (0000)	0	constant 0 function	
f_1 (0001)	$\wedge$, $\cdot$, &, &&, $\cap$, ',' (comma in Prolog), and implied by juxtaposition when no operator is shown: i.e. if p, q, and r are understood as proposition symbols, p $\vee$ qr means p $\vee$ (q $\wedge$ r)	"and" function	p $\wedge$ q == 1 when both p == 1 and q == 1 p $\wedge$ q == 0 when either p == 0 or q == 0
f_2 (0010)		negation of “implies”	
f_3 (0011)	π_1	projection of first argument	
f_4 (0100)		negation of “if”	
f_5 (0101)	π_2	projection of second argument	
f_6 (0110)	$\oplus$, $\not\equiv$	"exclusive-or" ("xor")	p $\oplus$ q == 1 when p has the opposite value of q
f_7 (0111)	$\vee$, +, |, ||, and $\cup$	"or" function	p $\vee$ q == 1 when either p == 1 or q == 1 p $\vee$ q == 0 when both p == 0 and q == 0
f_8 (1000)	$\downarrow$	"nor" (not-or) Also called the “dagger” or joint-denial function.	nor(p, q) == 1 when p == 0 or q == 0
f_9 (1001)	$\equiv$, $\leftrightarrow$, $\Leftrightarrow$, and ==	"iff" ("if and only if")	result is 1 exactly when both arguments are equal
f_{10} (1010)	$\neg$	negation of second argument	
f_{11} (1011)	$\leftarrow$, $\Leftarrow$, $\subset$, and :-	“if” function	

	(the last in Prolog		
f_{12} (1100)	¬	negation of first argument	
f_{13} (1101)	→, ⇒, ⊃	"implies" function	p → q == 1 when p == 0 or q == 1 p → q == 0 when p == 1 and q == 0
f_{14} (1110)	|	"nand" (not-and) Clasically called the "Sheffer stroke" function or alternative-denial.	nand(p, q) == 1 when p == 0 and q == 0
f_{15} (1111)	1	constant 1 function	

Aside from the definition of the function in the truth table, there are some associations we should make with the commonly-used functions. Remember that the value of a function can only be 1 or 0. So it suffices to state exactly the cases in which the value is 1, the other cases being implied to be 0. We make these statements using "iff" to mean "if, and only if":

(b ∧ c) == 1 iff b == 1 **and** c == 1
(b ∨ c) == 1 iff b == 1 **or** c == 1
(b → c) == 1 iff b == 0 **or** c == 1
(b ⊕ c) == 1 iff b is not equal to c
(¬b) == 1 iff b == 0

Stand-Alone Convention

Because 1 is equated to true, we sometimes omit the == 1 in a logical equation. In other words, we would read

b ∧ c

standing alone as

b and c

i.e. *b* is true and *c* is true. Likewise, since == can be regarded as an operator on bits, it behaves as "iff":

b iff c

is the same as

b == c

in the stand-alone convention, or

(b == c) == 1.

Finally, using the stand-alone convention

¬b

the negation of b, would be the same as ($\neg$b) == 1, meaning that *b* is false (0).

Tautologies

> A **tautology** is a propositional expression that evaluates to 1 for every assignment of values to its proposition variables.

When we use the stand-alone convention for propositional expressions without any further qualifications on the meaning of variables, we are asserting that the expression is a tautology. The following are examples of tautologies:

$$1$$
$$\neg 0$$
$$p \vee \neg p$$
$$p \rightarrow p$$

However, there will be many cases where it is not so obvious that something is a tautology. Here are some examples:

$$(p \rightarrow q) \vee (q \rightarrow p)$$
$$p \rightarrow (q \rightarrow p)$$

Both of the above tautologies might look unintuitive at first. To prove that they are tautologies, one can try evaluating each assignment of 0 and 1 to the variables, i.e. construct the truth table for the expression, and verify that the result is 1 in each case.

Example Show that $(p \rightarrow q) \vee (q \rightarrow p)$ is a tautology.

For p = 0, q = 0: $(0 \rightarrow 0) \vee (0 \rightarrow 0) == 1 \vee 1 == 1$
For p = 0, q = 1: $(0 \rightarrow 1) \vee (1 \rightarrow 0) == 1 \vee 0 == 1$
For p = 1, q = 0: $(1 \rightarrow 0) \vee (0 \rightarrow 1) == 0 \vee 1 == 1$
For p = 1, q = 1: $(1 \rightarrow 1) \vee (1 \rightarrow 1) == 1 \vee 1 == 1$

Part of the reason that this formula might not appear to be a tautology concerns the way that we English speakers use words like “implies” in conversation. We often use “implies” to suggest a *causal* relationship between two propositions, such as:

> “Not doing homework” implies “low grade in the course”.

In logic, however, we use what is called the *material* sense of implication. Two propositions might be quite unrelated causally, and still an implication holds:

> “Disneyland is in Claremont” implies “It is raining in Claremont”

While there is obviously no relation between the location of Disneyland and whether it is raining in Claremont, the fact that the lefthand proposition has the value 0 (false) renders the above a true statement, since $0 \rightarrow p$ regardless of the truth value of p.

The other source of misdirection in the tautology $(p \rightarrow q) \vee (q \rightarrow p)$ is that we are not saying that one can choose any p and q whatsoever and it will either be the case that always $p \rightarrow q$ or always $q \rightarrow p$. Rather, we are saying that no matter what values we assign p and q, $(p \rightarrow q) \vee (q \rightarrow p)$ will always evaluate to 1. Thus

("It is sunny in Claremont" implies "It is raining in Claremont")
or ("It is raining in Claremont" implies "It is sunny in Claremont")

is true as a whole, even though the individual disjuncts are not always true.

Substitution Principle

The substitution principle is the following:

> **Substitution Principle**
>
> In a tautology, if we replace all occurrences of a given propositional variable with an arbitrary propositional expression, the result remains a tautology.

The reason this is correct is that, in a tautology, it matters not whether the original variable before substitution is true or false; the overall expression is still invariably true.

Example

In the tautology $p \vee \neg p$, replace p with $a \rightarrow b$. The result, $(a \rightarrow b) \vee \neg(a \rightarrow b)$ is also a tautology.

Logic Simplification Rules

These rules follow directly from the definitions of the logic functions $\wedge$, $\vee$, etc. In part they summarize previous discussion, but it is thought convenient **t**o have them in one p*l*ace.

For any propositions p, q, and r:

$\neg (\neg p) == p$	double negative is positive
$(p \wedge 0) == (0 \wedge p) == 0$	0 absorbs $\wedge$
$(p \wedge 1) == (1 \wedge p) == p$	$\wedge$ ignores 1
$(p \vee 1) == (1 \vee p) == 1$	1 absorbs $\vee$
$(p \vee 0) == (0 \vee p) == p$	$\vee$ ignores 0
$(p \vee \neg p) == 1$	the excluded middle
$(p \wedge \neg p) == 0$	the excluded miracle
$(p \rightarrow q) == (\neg p \vee q)$	$\rightarrow$ as an abbreviation
$(0 \rightarrow p) == 1$	false implies anything
$(0 \rightarrow p) == 1$	anything implies true
$(p \rightarrow 1) == 1$	$(1 \rightarrow p)$ forces p
$(p \rightarrow 0) == \neg p$	$(p \rightarrow 0)$ negates p
$\neg(p \wedge q) == (\neg p) \vee (\neg q)$	DeMorgan's laws
$\neg (p \vee q) == (\neg p) \wedge (\neg q)$	DeMorgan's laws
$p \wedge (q \vee r) == (p \wedge q) \vee (p \wedge r)$	$\wedge$ distributes over $\vee$
$p \vee (q \wedge r) == (p \vee q) \wedge (p \vee r)$	$\vee$ distributes over $\wedge$
$p \vee (\neg p \wedge q) == (p \vee q)$	complementary absorption rules
$\neg p \vee (p \wedge q) == (\neg p \vee q)$	complementary absorption rules
$p \wedge (p \vee q) == (p \wedge q)$	complementary absorption rules
$\neg p \wedge (p \vee q) == (\neg p \wedge q)$	complementary absorption rules

The first few of these rules can be used to justify the following convention, used in some programming languages, such as Java:

Short-Circuit Convention

Evaluation of Java logical expressions involving

&& for "and" ($\wedge$)
|| for "or" ($\vee$)

takes place left-to-right only far enough to determine the value of the overall expression.

For example, in Java evaluating

```
f() && g() && h()
```

we would evaluate f(), then g(), then h() in turn only so long as we get non-0 results. As soon as one gives 0, the entire result is 0 and evaluation stops. This is of most interest when the arguments to && and || are **expressions with side-effects**, since some side-effects will not occur if the evaluation of the logical expression is "short circuited". This is in contrast to Pascal, which always evaluates all of the expressions. Experienced programmers tend to prefer the short-circuit convention, so that redundant computation can be avoided.

Exercises

1 • Express the functions f_2 and f_4 from the table of sixteen functions of two variables using $\{\wedge, \vee, \neg\}$.

2 •• Does the exclusive-or function $\oplus$ have the property of commutativity? Of associativity?

3 •• Which of the following distributive properties are held by the exclusive-or function?

$p \wedge (q \oplus r) == (p \wedge q) \oplus (p \wedge r)$	$\wedge$ distributes over $\oplus$
$p \vee (q \oplus r) == (p \vee q) \oplus (p \vee r)$	$\vee$ distributes over $\oplus$
$p \oplus (q \wedge r) == (p \oplus q) \wedge (p \oplus r)$	$\oplus$ distributes over $\wedge$
$p \oplus (q \vee r) == (p \oplus q) \vee (p \oplus r)$	$\oplus$ distributes over $\vee$

9.5 Logic for Circuits

A general problem in computer design is that we need to implement functions on the bit domain out of a library of given functions. Such a library might include primitive circuits for implementing $\wedge$, $\vee$, $\neg$, etc. It would likely include some **multi-argument** variants of these. For example, we have both the **associative** and **commutative** properties:

$a \wedge b == b \wedge a$	commutative property of $\wedge$
$a \vee b == b \vee a$	commutative property of $\vee$
$a \wedge (b \wedge c) == (a \wedge b) \wedge c$	associative property of $\wedge$
$a \vee (b \vee c) == (a \vee b) \vee c$	associative property of $\vee$

When both the associative and commutative properties hold for a binary operator, we can derive a function that operates on a bag of values (i.e. repetitions are allowed and order is not important). For example,

$\wedge (a, b, c, d) == a \wedge (b \wedge (c \wedge d))$

We do not need to observe either ordering or grouping with such operators; so we could use the equivalent expressions

$a \wedge b \wedge c \wedge d$

$\wedge (d, c, b, a)$

among many other possibilities.

Now consider the following:

> **Universal Combinational Logic SynthesisQuestion**
>
> Given a function of the form $\{0, 1\}^N \rightarrow \{0, 1\}$ for some N, is it possible to express the function using functions from a given set of functions, such as $\{\wedge, \vee, \neg\}$, and if so, how?

As it turns out, for the particular set $\{\wedge, \vee, \neg\}$, we can express *any* function for any number N of variables whatsoever. We might say therefore that

$\{\wedge, \vee, \neg\}$ is universal

However, this set is not uniquely universal. There are other sets that would also work, including some with fewer elements.

Modulo 3 Adder Synthesis Example

Consider the functions f_1 and f_2 in our modulo-3 adder example, wherein we derived the following tables:

		wx		
	f_1	00	01	10
	00	0	0	1
uv	01	0	1	0
	10	1	0	0

Table for the first result bit of encoded modulo 3 addition.

		wx		
	f_2	00	01	10
	00	0	1	0
uv	01	1	0	0
	10	0	0	1

Table for the second result bit of encoded modulo 3 addition.

How can we express the functions f_1 and f_2 using only elements from $\{\wedge, \vee, \neg\}$? The reader can verify that the following are true:

$$f_1(u, v, w, x) == (u \wedge \neg v \wedge \neg w \wedge \neg x) \vee (\neg u \wedge v \wedge \neg w \wedge x) \vee (\neg u \wedge \neg v \wedge w \wedge \neg x)$$

$$f_2(u, v, w, x) == (\neg u \wedge \neg v \wedge \neg w \wedge x) \vee (\neg u \wedge v \wedge \neg w \wedge \neg x) \vee (u \wedge \neg v \wedge w \wedge \neg x)$$

How did we arrive at these expressions? We examined the tables for those combinations of argument values uvwx that rendered each function to have the result 1. For each such combination, we constructed an expression using only $\wedge$ and $\neg$ that would be 1 for this combination only. (Such expressions are called **minterms**. There are three of them for each of the two functions above.) We then combined those expressions using $\vee$.

We often use other symbols to make the propositional expressions more compact. Specifically,

It is common to use either a postfix prime (') or an over-bar in place of $\neg$.

It is common to use $\cdot$ in place of $\wedge$, or to omit $\wedge$ entirely and simply juxtapose the literals (where by a "literal" we mean a variable or the negation of a variable). A term constructed using $\wedge$ as the outer operator is called a **product** or a **conjunction**.

We sometimes use + instead of $\vee$. An expression constructed using $\vee$ as the outer operator is called a **sum** or a **disjunction**.

Making some of these substitutions then, we could alternatively express f_1 and f_2 as

$f_1(u, v, w, x) == u\ v'\ w'\ x' + u'\ v\ w'\ x + u'\ v'\ w\ x'$

$f_2(u, v, w, x) == u'\ v'\ w'\ x + u'\ v\ w'\ x' + u\ v'\ w\ x'$

The following diagram indicates how we derive this expression for the first function.

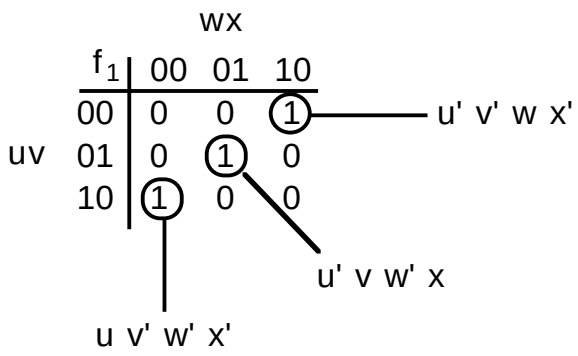


Figure 131: Showing the derivation of minterms for a function

Minterm Expansion Principle

We can apply the technique described in the preceding section to *any* function represented by a truth table. We call this the

> **Minterm Expansion Principle**
>
> To express a function as a sum of minterms, where a minterm is a product of literals that includes each of the arguments of the function:
>
> 1. Identify those combinations of variable values where the function has value 1.
>
> 2. Construct a product of literals corresponding to each combination. If a variable has value 1 in the combination, then the variable appears without negation in the product. If a variable has value 0 in the combination, then the variable appears with negation in the product.
>
> 3. Form the sum of the products constructed in step 2. This is the minterm expansion representation of the function.

The justification of this principle is straightforward. The function has value 1 for certain combinations and 0 for all others. For each combination where it has value 1, the corresponding minterm also has value 1. Since the minterm expansion is exactly the sum of those minterms, the function will have value 1 iff its minterm expansion has value 1.

The minterm expansion principle also shows us that the set $\{\wedge, \vee, \neg\}$ is universal, since the minterm expansion is made up of only these operators and variables. It tells us one way to implement a bit-function from primitive logic elements. Such an implementation is just a different representation of the minterm expansion, specifically a form of the

DAG representation for the syntax of the expression. For example, for the expression for f_1 above, the logic implementation would be shown as

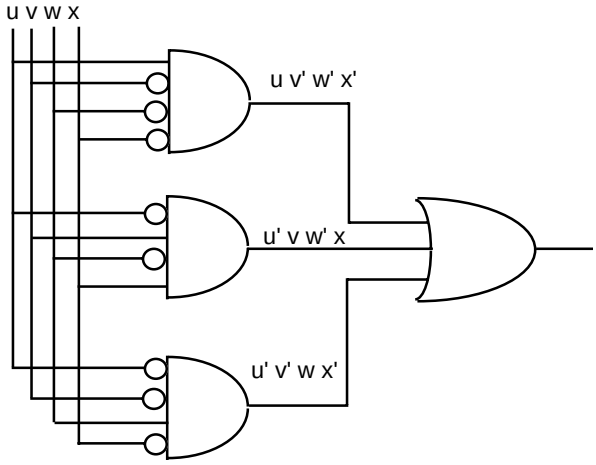


Figure 132: Implementation corresponding to minterm expansion of f_1(u, v, w, x) == u v' w' x' + u' v w' x + u' v' w x'

Here the small circles represent negation, the node with a curved left side is disjunction, and the nodes with straight left sides are conjunctions.

Later on, we will examine some ways to simplify such implementations, for example to reduce the amount of hardware that would be required. Meanwhile, it will be useful to have in our repertoire one other form of expansion that will help our understanding and analysis. This expansion will lead to an implementation sometimes different from the minterm expansion. However, the main uses of this principle will transcend those of minterm expansion.

Programmable Logic Arrays

A **programmable logic array** (**PLA**) is a unit that can be used to implement a variety of different functions, or several functions simultaneously. It is programmable in the sense that the functionality can be specified after the manufacture of the unit itself. This is done by blowing fuse links that are internal to the unit itself. This is allows a single integrated-circuit package to be used to implement fairly complex functions without requiring custom design.

PLAs are oriented toward a two-level gate combination, with the output being an OR-gate fed by several AND-gates of the overall inputs to the unit. Plain or inverted versions of each input signal are available. The structure of a PLA is depicted below. More than two levels can be obtained by connecting outputs of the PLA to inputs.

In the PLA, each AND and OR gate consists of many possible inputs. However, all of these possibilities are represented abstractly by a single wire. By putting a dot on that wire, we indicate a connection of the crossing line as an input. Thus functions that can be represented by two level sum-of-products (SOP) expressions can be coded in the PLA by reading directly from the expression itself.

Example Program a PLA to be a 3-bit binary incrementer modulo 8 (function that adds 1, modulo 8). The truth table for the incrementer is

input			output		
x_2	x_1	x_0	y_2	y_1	y_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

We wire the AND-gates to activate one column corresponding to each row of the truth table. We then wire the OR-gates to activate on any of the rows for which the corresponding output is 1. The result is shown below.

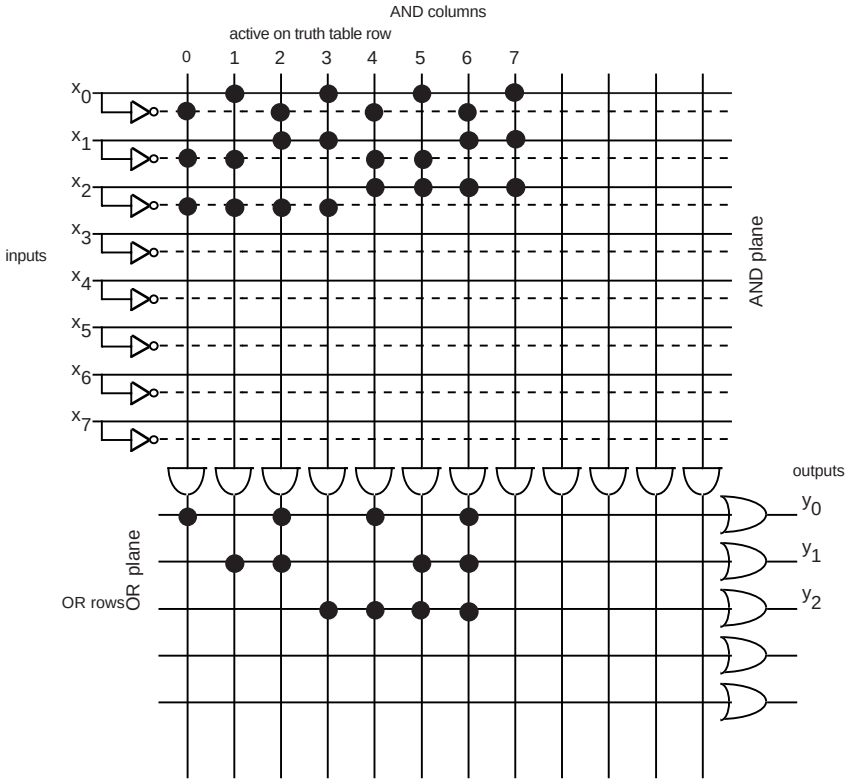


Figure 133: A PLA programmed to add 1 (modulo 8) to a 3-bit binary numeral

A less-cluttered, although not often seen, notation would be to eliminate the depiction of negation wires and indicate negation by open "bubbles" on the same wire. For the above logic functions, this scheme is shown below.

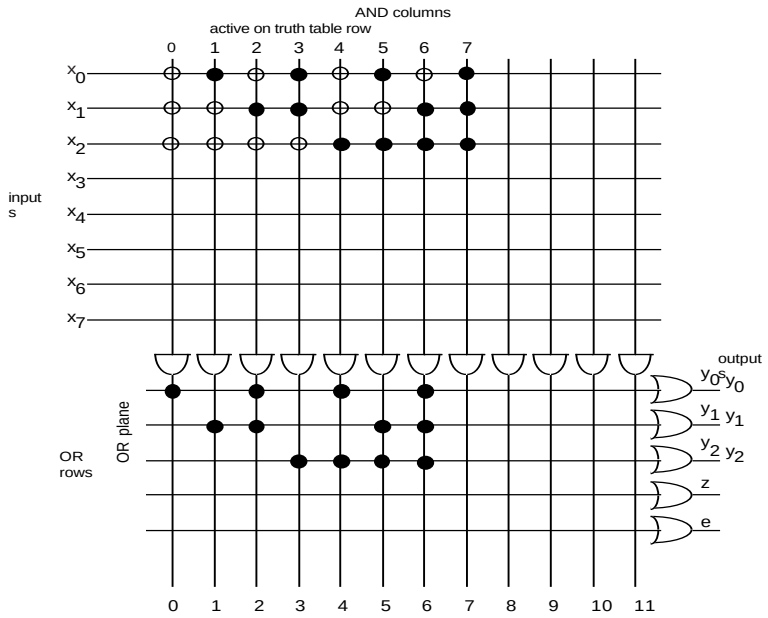


Figure 134: An alternate PLA notation for negation

Boole/Shannon Expansion Principle

Another way of establishing the universality of $\{\wedge, \vee, \neg\}$, as well as having other applications, is this important principle:

> **Boole/Shannon Expansion Principle**
> Let E be any proposition logic expression and p some proposition symbol in E. Let E_1 stand for E with all occurrences of p replaced with 1, and let E_0 similarly stand for E with all occurrences of p replaced with 0. Then we have the equivalence
>
> $$\mathbf{E == (p \wedge E_1) \vee (\neg p \wedge E_0)}$$

Proof: Variable p can only have two values, 0 or 1. We show that the equation holds with each choice of value. If p == 1, the lefthand side is equal to E_1 by definition of the latter. The righthand side simplifies to the same thing, since $(\neg 1 \wedge E_0)$ simplifies to 0 and $(p \wedge E_1)$ simplifies to E_1. On the other hand, if p == 0, the lefthand side is equal to E_0. The righthand side again simplifies E_0 in a manner similar to the previous case.

There are several uses of this principle:

Regrouping an expression by chosen variables (useful in logic circuit synthesis).

Simplifying an expression by divide-and-conquer.

Testing whether an expression is a *tautology* (whether it is equivalent to 1).

The Boole/Shannon Principle can be used to expand and analyze expressions recursively. Let us try it on the same expression for f_1 as discussed earlier. The righthand side for f_1 is

u v' w' x' + u' v w' x + u' v' w x'

If we take this to be E in the Boole/Shannon principle, we can choose any of the four variables as p. Let us just take the first variable in alphabetic order, u. The principle says that E is equivalent to

$u\ E_1 + u'\ E_0$

where E_1 is 1 v' w' x' + 1' v w' x + 1' v' w x', which immediately simplifies to v' w' x', since 1' is 0, which absorbs the other literals. Similarly, E_0 is 0 v' w' x' + 0' v w' x + 0' v' w x', which simplifies to v w' x + v' w x'. So we now have our original expression being recast as

u (v' w' x') + u' (v w' x + v' w x').

The implementation corresponding to the Boole/Shannon expansion could be shown as the following, where E_1 and E_0 can be further expanded.

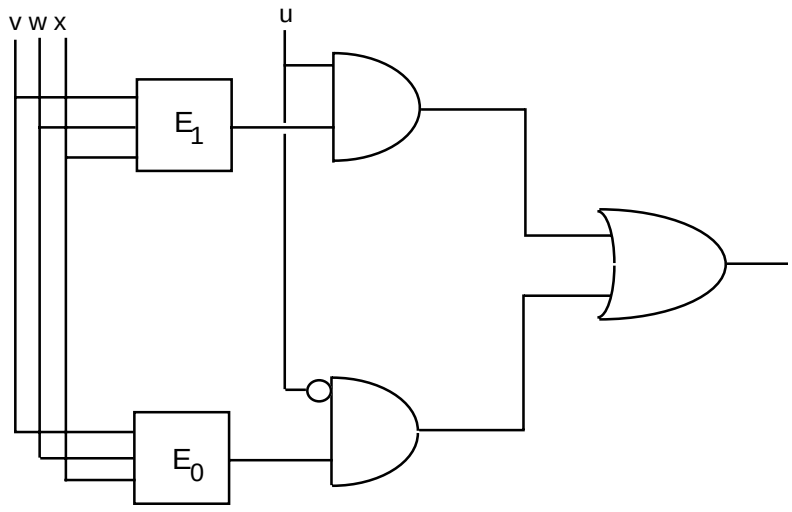


Figure 135: The Boole/Shannon principle applied to logic implementation

Incidentally, the structure below, which occurs in the Boole/Shannon principle, is known as a *multiplexor* or *selector*. It has a multitude of uses, as will be seen later. The reason for the name "selector" is that it can select between one of two logical inputs based upon the setting of the selection control line to 0 or 1. (Later we will call this an "address" line.)

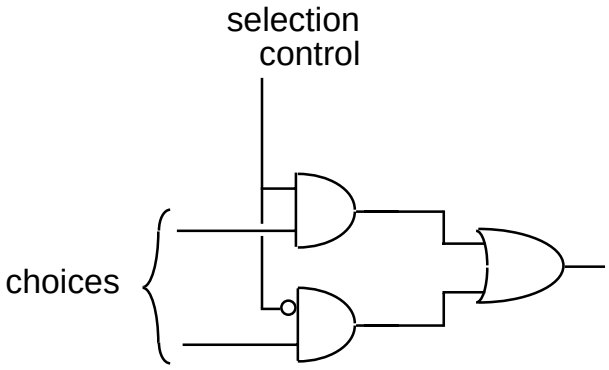


Figure 136: Multiplexor or selector structure

The multiplexor structure can be thought of as the hardware analog to the *if* statement in programming languages.

Tautology Checking by Boole/Shannon Expansion Example 1

Let's investigate whether $(p \rightarrow q) \vee (q \rightarrow p)$ is a tautology using the Boole/Shannon principle. Choose the variable p for expansion. Then

E_1 is $(1 \rightarrow q) \vee (q \rightarrow 1)$

E_0 is $(0 \rightarrow q) \vee (q \rightarrow 0)$

Since we know $(q \rightarrow 1) == 1$, E_1 simplifies to 1. We also know $(0 \rightarrow q) == 1$, so E_0 simplifies to 1. Thus E is equivalent to

$p \cdot 1 \vee p' \cdot 1$

which is a known tautology. Therefore the original is a tautology.

Observations In creating a Boole/Shannon expansion, the original expression is a tautology iff E_1 and E_0 both are tautologies. Since E_1 and E_0 have one fewer variable than the original expression (i.e. neither contains p, for which we have substituted) we have recursive procedure for determining whether an expression is a tautology: Recursively expand E to E_1 and E_0, E_1 to E_{11} and E_{10}, E_{11} to E_{110} and E_{110}, etc. until no variables are left. [We don't actually have to use the numberings in a recursive procedure, since

only two expressions result in any given stage.] If any of the limiting expressions is 0, the original is not a tautology. If all expressions are 1, the original is a tautology.

The following diagram suggests the use of repeated expansions to determine whether an expression is a tautology:

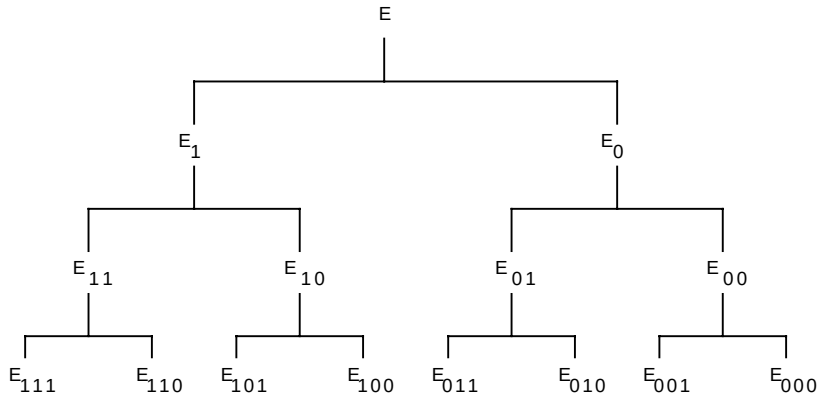


Figure 137: Tree showing the form of recursive use of Boole/Shannon expansion

Tautology Checking by Boole/Shannon Expansion Example 2

Let's determine whether or not $((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$ is a tautology.

E is $((a \rightarrow b) \wedge (b \rightarrow c)) \rightarrow (a \rightarrow c)$.

Looking at E, we see that if we make c = 1, then the whole expression will simplify to 1. Thus c is a good choice for the first expansion variable.

Expanding *E* on *c*:

E_1 is $((a \rightarrow b) \wedge (b \rightarrow 1)) \rightarrow (a \rightarrow 1)$. Since $(a \rightarrow 1) == 1$ independent of *a*, this simplifies to $((a \rightarrow b) \wedge (b \rightarrow 1)) \rightarrow 1$, which further simplifies to 1 for the same reason. Thus we do not have to go on expanding E_1.

E_0 is $((a \rightarrow b) \wedge (b \rightarrow 0)) \rightarrow (a \rightarrow 0)$. Since for any p, $(p \rightarrow 0)$ is $\neg p$, E_0 simplifies to $((a \rightarrow b) \wedge \neg b) \rightarrow \neg a$.

Expanding the simplified E_0 on *a*:

E_{01} is $((1 \rightarrow b) \wedge \neg b) \rightarrow \neg 1$. This simplifies to $(b \wedge \neg b) \rightarrow 0$, which simplifies to $0 \rightarrow 0$, which simplifies to 1.

E_{00} is $((0 \rightarrow b) \wedge \neg b) \rightarrow \neg 0$, which simplifies to $(1 \wedge \neg b) \rightarrow 1$, which simplifies to 1.

Thus, by taking some care in choosing variables, we have shown the original E to be a tautology by expanding only as far as E_1, E_{01}, and E_{00}, rather than to the full set E_{111}, E_{110}, ... E_{000}.

Logic Circuit Simplification by Boole/Shannon Expansion Example

Occasionally when we expand on a particular variable using the Boole/Shannon expansion, the result can be simplified from what it would be with the full-blown multiplexor structure. Here once again is the equation for the Boole/Shannon expansion:

$$\mathbf{E == (p \wedge E_1) \vee (\neg p \wedge E_0)}$$

In the special case that E_1 simplifies to 0, the term $p \wedge E_1$ also simplifies to 0, so that E simplifies to $\neg p \wedge E_0$. Since several such simplifications are possible, let's make a table:

Case	**E simplifies to**
E_1 simplifies to 0	$\neg p \wedge E_0$
E_0 simplifies to 0	$p \wedge E_1$
E_1 simplifies to 1	$p \vee E_0$
E_0 simplifies to 1	$\neg p \vee E_1$
E_0 and E_1 simplify to 0	0
E_0 and E_1 simplify to 1	1
E_0 and E_1 simplify to the same thing	E_0
E_0 and E_1 simplify to opposites	$p \oplus E_0$

Table of some simplifications based on the Boole/Shannon expansion

For example, if E_0 simplifies to 0 then our original logic implementation based on the Boole/Shannon expansion could be replaced with the following much simpler one:

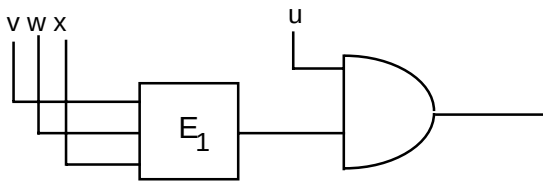


Figure 138: Simplified logic circuit as a result of Boole/Shannon expansion

Counterexamples to Propositions

When a logical expression is not a tautology, there must be some assignment of truth values to the variables under which the expression evaluates to 0 rather than to 1. Such an assignment is sometimes called a "counterexample". It is, of course, possible for multiple counterexamples to exist for a given expression.

The Boole/Shannon expansion tree can be used to produce counterexamples in the case that the starting expression is not a tautology. As discussed, a non-tautology must result in a node that simplifies to 0 somewhere in the tree. The path from the root to a given node corresponds to an assignment of truth values to some of the variables in the expression. Going to the left in the diagram corresponds to assigning the value 1 and to the right, the value 0. It is easy to see that the expression at a given node corresponds to a simplification of the expression under the set of choices made at each branch. Thus, if a node simplifying to 0 is encountered, the choices represented by the path from that node to the root for a counterexample.

Exercises

1 •• Show that the set $\{\vee, \neg\}$ is universal. [Hint: Show that $\wedge$can be expressed using $\{\vee, \neg\}$. Conclude that anything that could be expressed using only $\{\wedge, \vee, \neg\}$ could also be expressed using $\{\vee, \neg\}$. Show that $\{\wedge, \neg\}$ is also universal.

2 •• Show that {*nand*} is universal. Show that {*nor*} is universal.

3 •• Show that the set of functions $\{\rightarrow, \neg\}$ is universal.

4 ••• Let **1** designate the constant 1 function. Is $\{\mathbf{1}, \oplus\}$ universal? Justify your answer.

5 •••• Show that the set of functions $\{\oplus, \neg\}$ is not universal. [Hint: Find a property shared by all functions that can be constructed from this set. Observe that some functions don't have this property.]

6 ••• Is $\{\wedge, \oplus\}$ universal? Justify your answer.

7 ••••• Is it possible to devise a computer program to determine whether a set of functions, say each in the form of a truth table, is universal?

8 •• Show the implementation corresponding to the next phase of expansion using the Boole/Shannon principle to expand both E_1 and E_0 above.

9 •• Using the Boole/Shannon expansion principle, show each of the rules listed earlier in *Simplification Rules Worth Remembering.*

10 ••• Show that the "dual" form of the expansion principle, wherein $\wedge$ and $\vee$ are interchanged and 0 and 1 are interchanged.

11 ••• Verify that each of the simplifications stated in the *Table of some simplifications based on the Boole/Shannon expansion* is actually correct.

12 •• Think up some other useful simplification rules, such as ones involving $\oplus$ and $\equiv$.

13 •• Determine which of the following are tautologies using Boole/Shannon expansion:

$(p \wedge (p \rightarrow q)) \rightarrow q$
$\neg p \rightarrow p$
$\neg (\neg p \rightarrow p)$
$(\neg p \rightarrow p) \rightarrow p$
$\neg p \rightarrow (p \rightarrow q)$
$((p \rightarrow q) \rightarrow p) \rightarrow p$
$(p \rightarrow q) \vee (\neg p \rightarrow q)$
$(p \rightarrow q) \vee (p \rightarrow \neg q)$
$(p \rightarrow q) == (\neg q \rightarrow \neg p)$
$(p \vee q) \rightarrow (p \wedge q)$
$(p \wedge q) \rightarrow (p \vee q)$
$(p \rightarrow q) \wedge (q \rightarrow r) == (p \rightarrow r)$
$(p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \rightarrow s) \rightarrow (p \rightarrow s)$

14 •• For those expressions in the last exercise above that turned out not to be tautologies, produce at least one counterexample.

15 ••• For the Logical Expression Simplifier exercise in the previous section, modify your program so that it gives a counter example for each non-tautology.

Karnaugh Maps

Karnaugh maps are a representation of truth tables for switching (proposition logic) functions that has uses in analysis and simplification. The idea can be traced to Venn diagrams used in visualizing sets. We assume the reader has prior exposure to the latter idea. To relate Venn diagrams to switching functions, consider a diagram with one region inside a universe. This region corresponds to a propositional variable, say x. Any 1-variable switching function corresponds to a shading of the area inside or outside the region. There are four distinct functions, which can be represented by the logical expressions x, x', 0, and 1. The corresponding shadings are shown below.

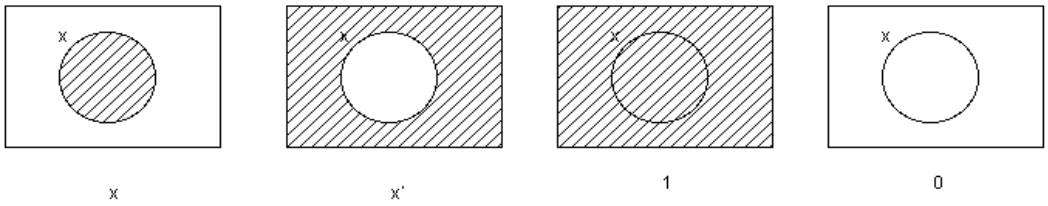


Figure 139: One-variable switching functions and their Venn diagrams.

Now consider two-variable functions. There are 16 of these and, for brevity, we do not show them all.

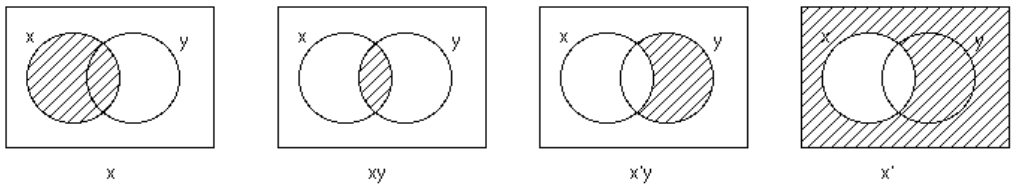


Figure 140: Some two-variable switching functions and their Venn diagrams.

The most important principle about Venn diagrams is that the sum of (the expressions representing) two or more functions can be depicted by forming the union of the shadings of the individual diagrams. This frequently leads to a view of the sum that is simpler than either summand.

Example

Show that x + x'y = x + y.

If we were asked to shade x + y, the result would be as shown below. On the other hand, the shadings for x and x'y are each shown in the previous figure. Note that combining the shadings of those figures results in the same shading as with x + y.

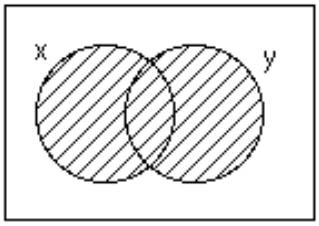


Figure 141: Venn diagram for x + y

Quite often, we would like to simplify a logical expression, but we don't know the answer in advance. To use Venn diagrams for this purpose, we would "plot" each of the

summands on the diagram, then "read off" a simplified expression. But it is not always obvious what the simplified result should be.

Example

Simplify xy'z' + x' + y.

The figure shows shadings for each of the summands, followed by the union of those shadings. The question is, what is the best way to represent the union? We can get a clue from the unshaded region, which is xy'z. Since this regions is unshaded, the shaded regions is the complement of this term, (xy'z)', which by DeMorgan's law is x' + y + z'.

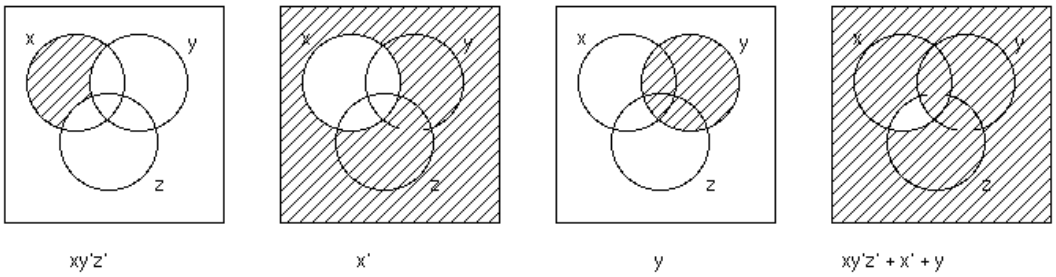


Figure 142: Venn diagrams for various expressions

Karnaugh maps are a stylized form of Venn diagram. They are most useful for simplifying functions of four variables or fewer. They can be used for five or six variables with more difficulty, and beyond six, they are not too helpful. However, a mechanizable method known as "iterated consensus" captures the essence of the technique in a form that can be programmed on a computer.

From Venn Diagrams to Karnaugh Maps

To see the relationship between a Karnaugh Map and a Venn diagram, let us assume three variable functions. The transformation from a Venn diagram to a Karnaugh map is shown below. Note that we are careful to preserve adjacencies between the primitive regions on the diagram (which correspond to minterm functions). The importance of this will emerge in the way that Karnaugh maps are used.

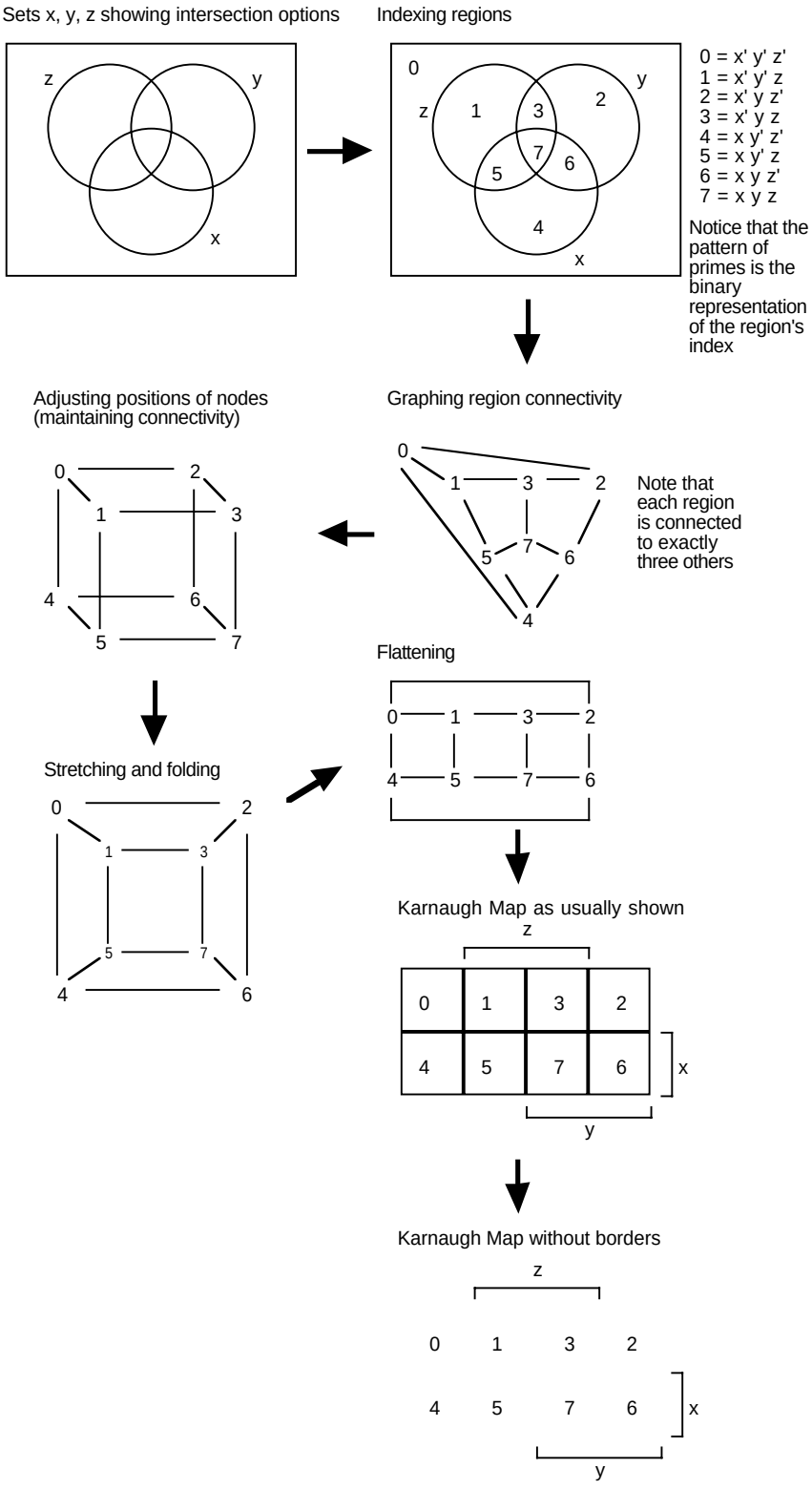


Figure 143: Transforming a Venn diagram into a Karnaugh map.

Hypercubes

Recall that a sum-of-product (SOP) form is a logical sum of products, where each product consists of either variables or their negations. For an n-variable function, each possible product term corresponds exactly to some sub-cube in the **n-dimensional hypercube H_n**, defined in an earlier chapter:

> H_0 is a single point.
>
> H_{n+1} is two copies of H_n, where each point of one copy is connected to the corresponding point in the other copy.

A **sub-cube** is a set of points in Hn that itself forms a hypercube. Above, the following sets of points are examples of sub-cubes: 0246, 1357, 0145, 2367, 04, 02, 0, and 01234567.

Conversely, each sub-cube corresponds to such a product term. Examples are shown in the various attachments. Therefore, any SOP form corresponds to a *set* of sub-cubes, and conversely. The truth table corresponding to such a function can be equated with the union of the points in those sets. These points are the rows of the table for which the function result is 1. Note that any given point might be present in more than one sub-cube. The important thing is that all points are "covered" (i.e. each point is included in at least one). Moreover, no sub-cube can contain points for which the function result is 0.

The means we have for making an SOP simpler are:

> Reduce the number of terms.
>
> Reduce the size of terms.

These two objectives translate into:

> Use **fewer** sub-cubes to cover the function (so there are fewer terms)
>
> Use **larger** sub-cubes (so the terms have fewer literals).

For example, given the choice between two sub-cubes of size 4 and one of size 8, we would always choose the latter.

One of the contributions of the Karnaugh map is to enable spotting the *maximal* sub-cubes, the ones that are not contained in other sub-cubes for the same function. These sub-cubes are usually called the *prime implicants* of the function. The word "prime" in this case carries the same meaning as "maximal". The word "implicant" means that each term in the SOP for a function *implies* the function itself, i.e. whenever the assignment to variables is such that the term is 1, the function itself must be 1. This is because the

function can be represented as the *sum* of such terms.

Karnaugh Map Example

Consider function f(x, y, z) = x'y' + x'yz + xy'z + xy

In this SOP, as in all, each term is an implicant. However, only x'y' and xy are prime. The other two terms correspond to sub-cubes of a larger implicant z, as can be seen from the following map.

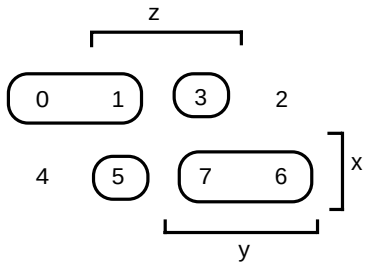


Figure 144: Karnaugh map for x'y' + x'yz + xy'z + xy

By examining the map, we can see that the sub-cube 1-3-5-7 corresponds to an implicant, in this case z. (It is a sub-cube by definition of "sub-cube", and it is an implicant because the function's value for all of its points are 1.) We can thus add this sub-cube to our SOP without changing the function's meaning:

f(x, y, z) = x'y' + x'yz + xy'z + xy + z.

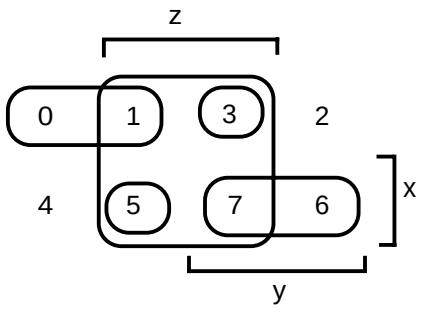


Figure 145: Adding a sub-cube to the map without changing the function

Then we notice that two other terms, xy'z corresponding to sub-cube 5, and x'yz corresponding to sub-cube 3, are both redundant. They are both *subsumed* by the new term z. (C is said to *subsume* D whenever D implies C.) Restricted to sub-cubes, C subsumes D when the points of C include all those of D.

As a consequence, we can eliminate the subsumed terms without changing the meaning of the function:

$$f(x, y, z) = x'y' + xy + z.$$

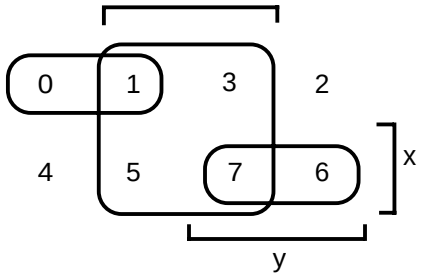


Figure 146: Eliminating subsumed terms from the map

Obviously we have achieved our goal of simplifying the function, by both reducing the number of terms, as well as the complexity of those terms. If we were implementing via NAND gates, we would have to use one NAND gate of 3 inputs and two of 2 inputs for this SOP, vs. one of 4 inputs, two of 3 inputs, and two of 2 inputs, for the original SOP, quite obviously a saving.

Notice that we cannot achieve further simplification by constructing still larger implicants from this point. Each implicant shown is prime.

What we have suggested so far is:

> The simplest SOP is constructed only of prime implicants.

We next observe that including *all* prime implicants is not necessarily the simplest. Since prime implicants can overlap, there might be redundancy if we include all. The following example shows this:

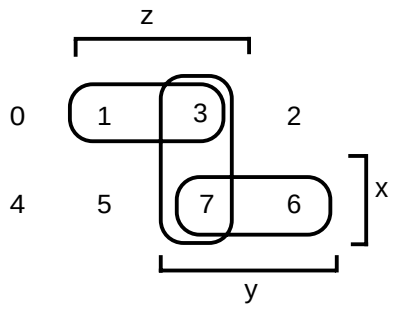


Figure 147: A Karnaugh map with redundancy among prime implicants

In the above example, the prime implicants are x'z, yz, and xy, corresponding to 13, 37, and 67 respectively. However, we do not need the second prime implicant to cover all points in the function. On the other hand, the first and third prime implicants will be required in any SOP for the function that consists only of prime implicants. Such prime implicants are called *essential*.

It is possible for a non-trivial function to have no essential prime implicants, as the following example shows:

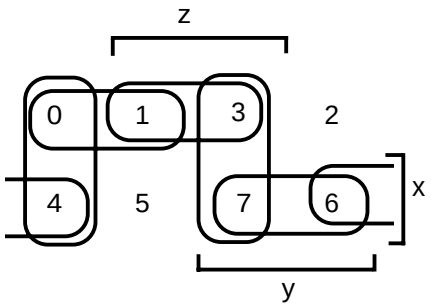


Figure 148: A Karnaugh map with no essential prime implicants

(Note that, because a map represents a hypercube, adjacency in the map extends to the "wrap-around" cases, such as xz' shown above.)Here there are six prime implicants, yet none is essential. In each SOP that covers the function, we can leave out one of the prime implicants. Thus we have six different implementations of the same complexity, five 2-variable prime implicants each.

The observations about redundancy are part of the reason that simplification of switching functions is complex. We cannot use a simple, straightforward, algorithm to get the optimum implementation. Instead it appears that we must generally resort to a more complex "backtracking" process. We do not pursue this further in these notes.

Karnaugh Maps of Higher Dimension

Karnaugh maps work well for representing hypercubes of dimension up to four. After that, they become harder to use for visualization. However, the principle on which they are based, called "consensus", can still be applied in a computer program, which is not limited to what can be visualized by human eyes. The figure below shows a 4-dimensional Karnaugh map obtained by juxtaposing two 3-dimension ones, one of which has been flipped over so that the cells with x = 1 are adjacent. This allows us to form the following sub-cubes:

- any 2 adjacent cells (horizontally or vertically, including wrap-around)
- any 4 adjacent cells in a 1x 4, 2 x 2, or 4 x 1 configuration, including wrap-around
- any 8 cells in a 2x 4 or 4 x 2 configuration, including wrap-around

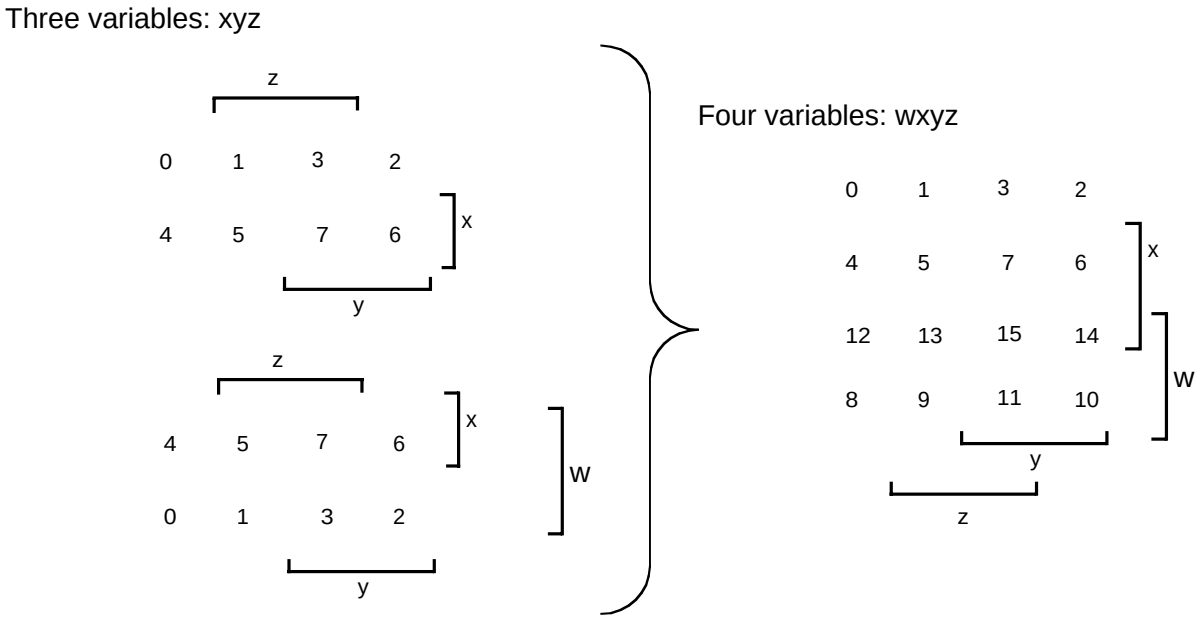


Figure 149: A 4-dimensional Karnaugh map

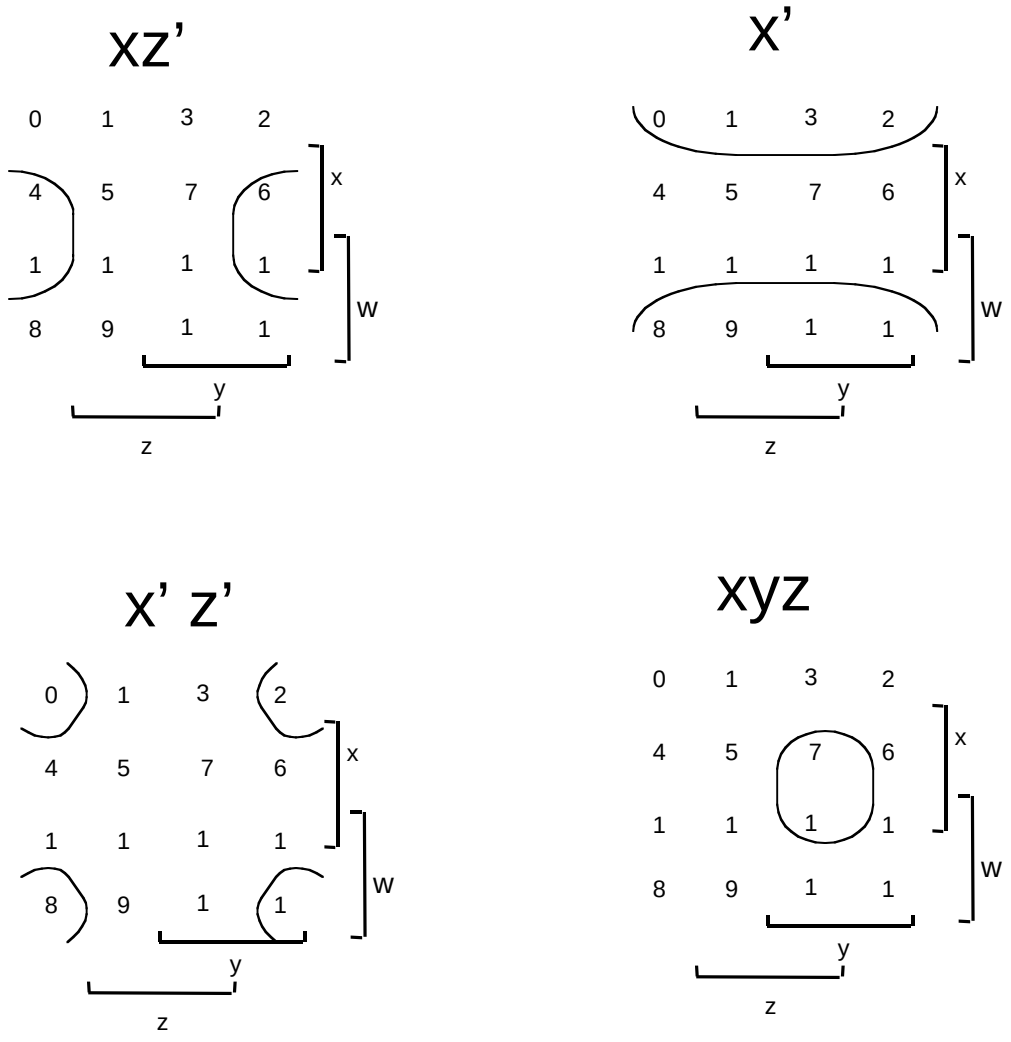


Figure 150: Sample sub-cubes on a 4-dimensional map

Functions and Karnaugh Maps with "Don't Cares"

In proceeding from a natural language problem statement to a switching function representation of the problem, the resulting function might not always be completely specified. That is, we will care about the function's results (0 or 1) for *some* combination of variables, but not care about its results for other combinations. One reason we might not care is that we know from the problem statement that these combinations cannot occur in practice.

Such "don't care" combinations often provide a bonus when it comes to finding

simplified SOP forms. Rather than stipulating an arbitrary choice of the function's value for these variables at the outset, we can wait until the simplification process begins. The technique is summarized as:

Choose the function value for don't care combinations to be 1 if it helps maximize the size of a covering sub-cube.

Example

Below we show a Karnaugh map for a function, where point 5, corresponding to term xy'z, is marked with a "d", indicating that we don't care about the function's output for that combination. In contract, the function's value is to be 0 for combinations x'yz' and xy'z', and 1 for all other combinations.

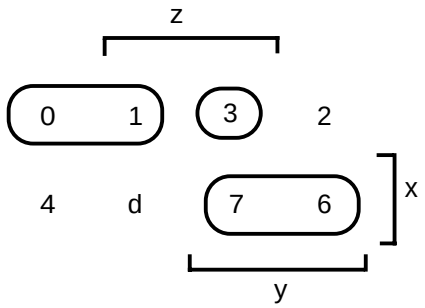


Figure 151: A Karnaugh map with "don't care" (marked d)

The choice of whether to cover any given cell marked "d" is up to us. Above, if we chose not to cover it (make the function have value 0 for xy'z), we would have the simplified implementation shown below, with SOP x'y' + x'z + yz + xy. Further simplification is possible in that one of the terms x'z or yz can be dropped without affecting the coverage: Either of x'y' + yz + xy or x'y' + x'z + xy both work.

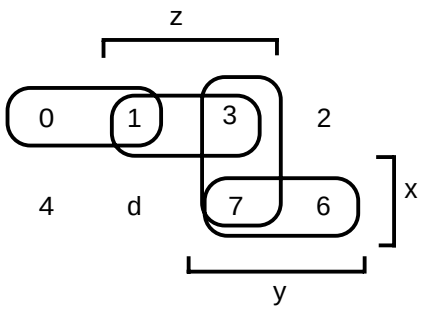


Figure 152: The prime implicants when d is assigned 0

If we choose to cover the cell marked "d" (make the function have value 1 for xy'z), we have the simplified implementation with SOP x'y' + xy + z, which is simpler than either of the simplest cases where we don't cover the d:

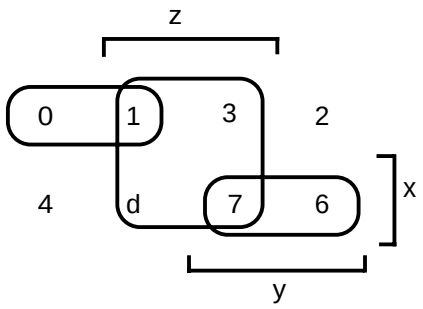


Figure 153: The prime implicants when d is assigned 1

In this case, it seems obvious that we should cover the d.

Iterated Consensus Principle for Finding Prime Implicants (Advanced)

Although what we are about to describe can be extended to include don't care cases, we choose not to do so for reasons of simplicity. When faced with functions with a large number of variables, we obviously would like to turn to the computer as a tool for simplification. Unfortunately, the technique presented so far for Karnaugh maps involves "eyeballing" the map to find the prime implicants. How can we express an equivalent technique in such a way that it can be represented in the computer? In short, what is the essence of the technique? This method is given the name "iterated consensus", and relies on two principles: consensus and subsumption.

The iterated consensus technique takes as input any set of product terms representing the function of interest. As output, it produces all of the prime implicants of the function. It is up to further analysis to use those prime implicants in constructing the simplest implementation.

The iterated consensus technique proceeds through a number of intervening states, where each state is a set of product terms. Initially this set is whatever is given as input. Finally, this set is the set of prime implicants. The consensus and subsumption principles are used to repeatedly modify the set until no further modifications are possible.

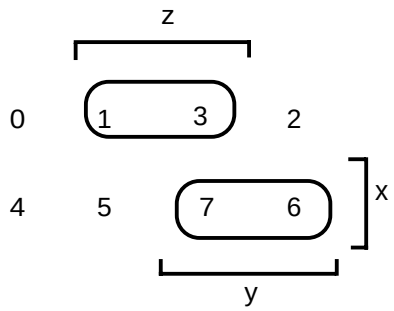
subsumption rule: If term U in the set is subsumed by a term V, then term U can be dropped from the set.

In terms of the Karnaugh map, this is equivalent to dropping a contained sub-cube.

consensus rule: If term U is a term not in the set, but which is the consensus of two

other terms V and W in the set, then term U can be added to the set. (However, terms that are subsumed by other terms in the set should not be added; they would just be removed by the subsumption rule anyway.) The exact definition of "consensus" will be given below; for now, we are discussing an informal example.

The consensus rule corresponds to introducing a new sub-cube on the map formed from points already covered by other sub-cubes. To see this, let us look at a typical map situation:



Clearly we can add the sub-cube 37 corresponding to the term yz. This term is the consensus of terms x'z and xy corresponding to the sub-cubes already covered. (We know that this sub-cube is not needed to cover the function, but the purpose of iterated consensus is to find prime implicants, and yz is certainly one.)

What we are saying by adding the consensus term is that

$$x'z + xy = x'z + xy + yz$$

To express the consensus in general, we note that the new term yz is found by the following considerations: For some variable, in this case x, one term has the variable complemented, the other uncomplemented. If we represent those terms as:

$$xF$$

and

$$x'G$$

then the consensus is just

$$FG$$

(in this case F is y, G is z, and therefore FG is yz).

Definition of Consensus: If U and V are two product terms, then:

If there is a single variable, say x, which appears uncomplemented in U and

complemented in V, then write U as xF and V as x'G (or the symmetric case, with U as x'F and V as xG), where both F and G are free of x. The consensus of U and V is defined to be FG. The operative identity in this case is:

xF + x'G = xF + x'G + FG

If the preceding case does not obtain, then the consensus is defined to be 0 (some authors would say it is "does not exist").

In terms of the Karnaugh map, the condition for the consensus to be non-zero is that the sub-cubes for F and G be "touching" on the map. The consensus term is the largest sub-cube that can be combined from sub-cubes of F and G, as suggested below.

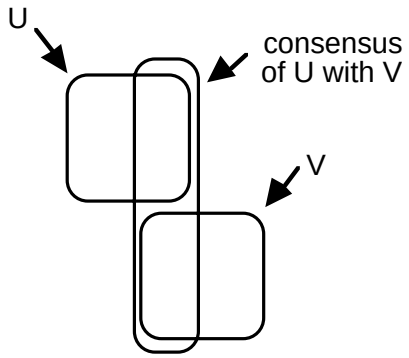


Figure 154: Showing consensus on a Karnaugh map

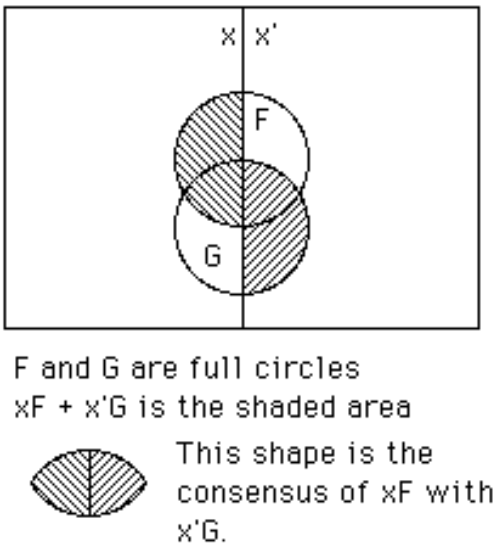


Figure 155: Showing consensus on a Venn diagram

Exercises

A good example of functions with don't cares can be found in various display encoders. For example, a seven-segment display consists of seven LEDs (light-emitting diodes), numbered s0 through s6, which display the digits from 0 through 9 as shown:

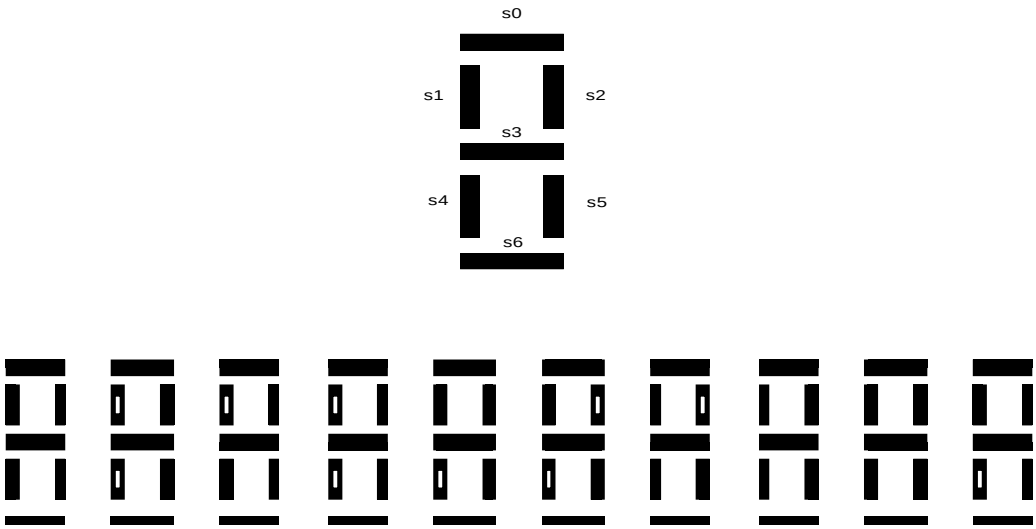


Figure 156: Display of 0 through 9 with a seven segment display

In the following, we assume that the digits are coded in BCD. This uses ten of sixteen possible combinations of four bits. The remaining combinations are don't cares. The seven segments correspond to seven switching functions.

1 • Give a Karnaugh map (with don't cares) for each of the switching functions.

2 •• Simplify each of the switching functions, using don't cares to advantage.

3 •• Construct gate realizations of the switching functions. Determine any possible sharing of product-terms among multiple functions.

4 •••• Develop a program that will input a logical expression and output the set of prime implicants for the corresponding function. Although different input formats are possible, a suggested internal format is to use a sequence of the symbols 0, 1, and x to represent a product term. For example, if we are dealing with a four-variable function, say f(u, v, w, x), then 01x0 represents u'vx'. (The x in the sequence represents that the absence of the corresponding letter, or equivalently, the union of two sub-cubes that are alike except for the values of that variable.) The reason for this suggestion is that the consensus of two such sequences is easy to compute. For example, the consensus of 01xx with 0x1x is 011x. This corresponds to u'v + u' w = u'vw.

9.6 Logic Modules

Although truth-tabular methods, maps, and the like are essential for understanding how computer logic works, they are not necessarily the best tools for building large systems, the problem being that the size of a truth table becomes overwhelming, even to a computer, when there are many inputs. The reason, of course, is that there are 2^N different input combinations for N input lines, and this number becomes large very fast. In order to handle this issue, designers structure systems using *modules* with understood behavior. At some level, truth-tabular methods are probably used to design aspects of these modules, but the modules themselves are understood using logic equations rather than tables.

Adder Module

A typical module found in a computer adds numbers represented by binary numerals. This module might be depicted as in the upper-left portion of the figure below. It could be realized by expanding it into the simpler FA ("full adder") modules shown in the main body of the figure. The term “full” is used for an adder module that adds three bits: two addend bits and a carry-in bit, to produce two bits: a sum and a carry-out bit. It contrasts with a “half adder”, which only adds two bits to produce a sum and carry-out. This type of adder structure is called a "ripple-carry" adder because the carry bits "ripple" through the FA gates. In the extreme case where all inputs, including the carry, are 1, the output carry production is delayed because it is a function of all of those input bits.

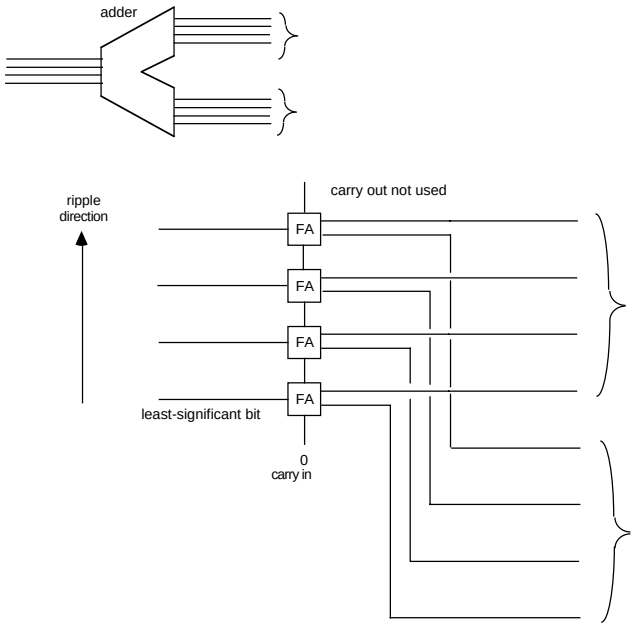


Figure 157: Expansion of the adder using ripple-carry; FA units are "full-adders"

The next figure shows a possible expansion of the FA modules using simpler logic functions. The M (majority) module has an output of 1 when 2 out of 3 inputs are 1. Therefore, its equation is:

carry-out = M(a, b, c) = ab +ac + bc

The $\oplus$ module is a 3-input exclusive-OR, i.e.

sum-out = a $\oplus$ b $\oplus$ c

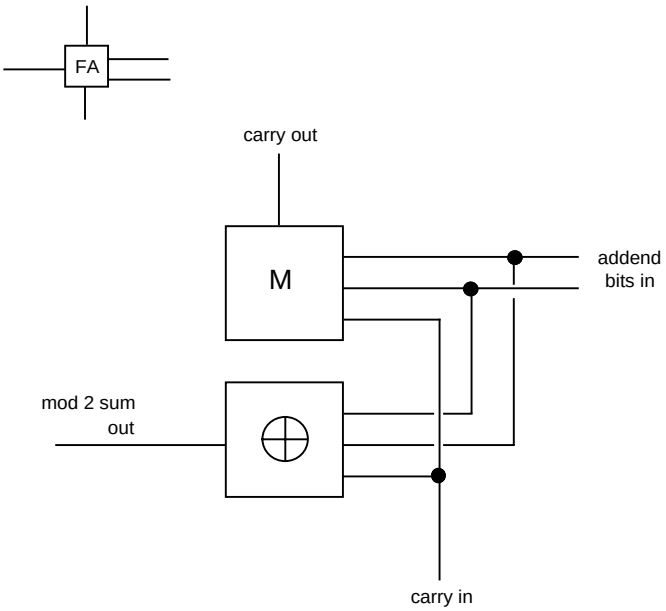


Figure 158: Expansion of the FA using M (majority) and 3-input exclusive-OR

Exercises

1 • Earlier we introduced the idea of a multiplexer, a module that has three inputs: two data inputs and an address input. The address input selects one or the other data input and reflects whatever is on that input to the output. Give a truth-table for the multiplexor. Although there are three input lines, we call this a "2-input" multiplexer, because selection is between two input liens.

2 •• Show the structure of a 4-input multiplexer. This unit will have inputs a, b, c, d and two address lines (assuming the address is encoded in binary). Such a device is shown below. (Hint: Use three 2-input multiplexers.)

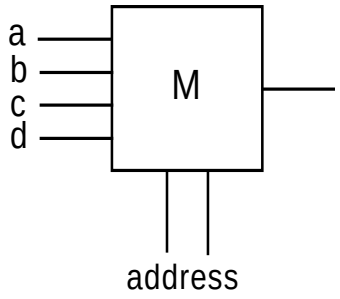


Figure 159: A 4-input multiplexer

3 ••• Show how to construct a 2n-input multiplexer from n-input multiplexers, for general n.

4 ••• Show how to construct recursively a 2^n-input multiplexer using only multiplexers with fewer inputs. If it is desired to build a multiplexer with 2^n inputs total, how many 2-input multiplexers would be required? (Construct a recurrence equation for the latter number and solve it.)

5 •• For the preceding problem, assuming that a singe 2-input multiplexer delays the input by 1 time unit, by how many units does your 2^n-input multiplexer delay its input.

6 ••• Show how a 4-input multiplexer can be used to implement an arbitrary combinational function of 2 logical variables. (Hint: Use the address lines as inputs and fix the *a*, *b*, *c*, *d* inputs.)

7 ••• Using the scheme of the previous problem, an 2^n-input multiplexer can be used to implement an arbitrary combinational function of how many logical variables?

8 ••• A *demultiplexer* (also called **DMUX**) reverses the function of a multiplexer, in that it has one input and several outputs. Based on the value of the address lines, the input is transmitted to the selected output line. The other output lines are held at 0. Show how to implement a 2-output and 4-output demultiplexer using simple logic gates.

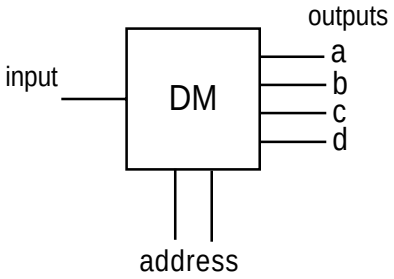


Figure 160: A 4-output demultiplexer

9 •• Show how to implement a 2^n-output demultiplexer for any n. How many simple gates are required in your construction?

10 •• A *decoder* is like a demultiplexer, except that the input is effectively held at a constant 1. Thus its outputs are a function of the address bits only. The decoder can be thought of as a converter from binary to a one-hot encoding. Suppose we have on hand an N-input decoder. What is the simplest way to build up an N-input multiplexer from this? What about building an N-output demultiplexer?

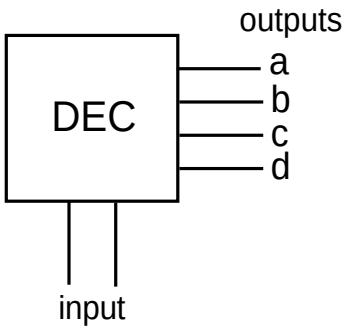


Figure 161: A 2-input, 4-output decoder, arranged to show the similarity to a demultiplexer.

11 •• Show that the outputs of a 2^n-output decoder are exactly the minterm functions on the address line variables.

12 •• An *encoder* reverses the role of outputs and inputs in a decoder. In other words, it converts a one-hot encoding to binary. Show how to build a 4-input encoder from simple logic gates.

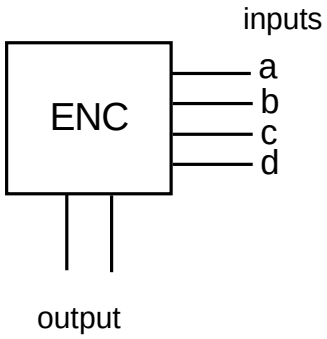


Figure 162: A 4-input, 2-output encoder.

13 ••• Show how to build a 3-input encoder from simple logic gates.

14 ••• Show how to build a 2^n-input encoder using a recursive construction.

15 •• Explore composing various combinations of encoder, decoder, multiplexer, demultiplexer together. What overall functions result?

16 ••• An "N-bean counter" counts the number of 1's among N input lines, presenting the output in binary. Design a logic circuit for a 3-bean counter (which will have 3 input and 2 output lines). Give a recursive construction for an N-bean counter, for arbitrary N.

9.7 Chapter Review

Define the following terms:

associative
binary code
binary-coded decimal
Boole/Shannon expansion
Cartesian encoding
combinational switching
commutative
conjunction
DeMorgan's laws
disjunction
don't-care condition
encoding
full adder
Gray code
half adder
hypercube
iff
implies
Karnaugh map
minterm expansion
one-hot code
parity
programmable logic array
proposition
subset code
substitution principle
tautology
universal set of switching functions
Venn diagram

9.8 Further Reading

George Boole, *An Investigation of the Laws of Thought*, Walton, London, 1854 (reprinted by Dover, New York, 1954). [Boole used 1 - *t* for the negation of *t*, working as if *t* were a number. The Boole/Shannon Expansion is stated: "If *t* be any symbol which is retained in the final result of the elimination of any other symbols from any system of equations, the result of such elimination may be expressed in the form

$$Et + E' (1-t)= 0$$

in which *E* is formed by making in the proposed system $t = 1$, and eliminating the same other symbols; and E' by making in the proposed system $t = 0$, and eliminating the same other symbols. Moderate.]

Frank Brown, *Boolean Reasoning*, Kluwer Academic Publishers, Boston, 1990. [Encyclopedic reference on Boolean algebra, with some applications to switching. Moderate to difficult.]

Augustus De Morgan, *On the Syllogism*, Peter Heath, ed., Yale University Press, New Haven, 1966. [DeMorgan's laws are stated: "(*A*, *B*) and *AB* have *ab* and (*a*, *b*) for contraries." Moderate.]

Martin Gardner, *Logic Machines and Diagrams*, University of Chicago Press, 1982. [Surveys diagrammatic notations for logic. Easy to moderate.]

Maurice Karnaugh, *The Map Method for Synthesis of Combinational Logic Circuits*, Transactions of the American Institute of Electrical Engineers, 72, 1, 593-599, November, 1953. [Introduction of the Karnaugh map.]

C.E. Shannon, *The synthesis of two-terminal switching circuits*, Trans. of the American Institute of Electrical Engineers, 28, 1, 59-98, 1949. [Gives a later version of the Boole/Shannon expansion.]

John Venn, *Symbolic Logic*, Chelsea, London, 1894. [Discourse on logic, introducing Venn diagrams, etc. Moderate]

Alfred North Whitehead and Bertrand Russell, *Principia Mathematica*, Cambridge University Press, London, 1910. [An original reference on logic. Moderate to difficult (notation).]

10. Predicate Logic

10.1 Introduction

Predicate logic builds heavily upon the ideas of proposition logic to provide a more powerful system for expression and reasoning. As we have already mentioned, a **predicate** is just a function with a range of two values, say `false` and `true`. We already use predicates routinely in programming, e.g. in conditional statements of the form

```
if( p(...args ...) )
```

Here we are using the two possibilities for the return value of p, (`true` or `false`). We also use the propositional operators to combine predicates, such as in:

```
if( p(....) && ( !q(....) || r(....) ) )
```

Predicate logic deals with the combination of predicates using the propositional operators we have already studied. It also adds one more interesting element, the "quantifiers".

The meaning of predicate logic expressions is suggested by the following:

Expression + Interpretation + Assignment = Truth Value

Now we explain this equation.

An **interpretation** for a predicate logic expression consists of:

a domain for each variable in the expression

a predicate for each predicate symbol in the expression

a function for each function symbol in the expression

Note that the propositional operators are not counted as function symbols in the case of predicate logic, even though they represent functions. The reason for this is that we do not wish to subject them to interpretations other than the usual propositional interpretation. Also, we have already said that predicates are a type of function. However, we distinguish them in predicate logic so as to separate predicates, which have truth values used by propositional operators, from functions that operate on arbitrary domains. Furthermore, as with proposition logic, the **stand-alone convention** applies with predicates: We do not usually explicitly indicate == 1 when a predicate expression is true; rather we just write the predicate along with its arguments, standing alone.

An **assignment** for a predicate logic expression consists of:

a value for each variable in the expression

Given an assignment, a truth value is obtained for the entire expression in the natural way.

Example

Consider the expression:

```
x < y || ( y < z  && z < x)
  ^          ^         ^      predicate symbols
```

Here || and && are propositional operators and < is a predicate symbol (in infix notation). An assignment is a particular predicate, say the *less_than* predicate on natural numbers, and values for x, y, and z, say 3, 1, and 2. With respect to this assignment then, the value is that of

```
3 < 1 || ( 1 < 2  && 2 < 3)
```

which is

```
false || ( true && true)
```

i.e.

```
true.
```

With respect to the same assignment for <, but 3, 2, 1 for x, y, z, the value would be that of

```
3 < 2 || ( 2 < 1  && 1 < 3)
```

which would be `false`. As long as we have assigned meanings to all variables and predicates in the expression, we can derive a `false` or `true` value. Now we give an example where function symbols, as well as predicate symbols, are present.

```
( (u + v) < y ) || ( (y < (v + w)) && v < x)
     ^                      ^              function symbols
```

would be an example of an expression with both function and predicate symbols. If we assign + and < their usual meanings and u, v, w, x, y the values 1, 2, 3, 4, 5 respectively, this would evaluate to the value of

```
( (1 + 2) < 4 ) || ( (4 < (2 + 3)) && 2 < 4 )
```

which is, of course, `true`.

Validity

It is common to be concerned with a fixed interpretation (of domains, predicates, and functions) and allow the assignment to vary over individuals in a domain. If a formula evaluates to true for all assignments, it is called **valid with respect to** the interpretation. If a formula is valid with respect to every interpretation, it is called **valid**. A special case of validity is where sub-expressions are substituted for proposition symbols in a tautology. These are also called *tautologies*. However, not every valid formula is a tautology, as is easily seen when we introduce quantifiers later on.

10.2 A Database Application

An important use of predicate logic is found in computer databases and the more general notion of "knowledge base", defined to be a database plus various computation rules. In this application, it is common to use predicate expressions containing variables as above as "queries". The predicates themselves represent the underlying stored data, computable predicates, or combinations thereof. A query asks the system to find all individuals corresponding to the variables in the expression such that the expression is **satisfied** (evaluates to 1). Next we demonstrate the idea of querying a database using the Prolog language as an example. Prolog is not the most widely-used database query language; a language known as SQL (Structured Query Logic) probably has that distinction. But Prolog is one of the more natural to use in that it is an integrated query language and programming language.

Prolog Database Example

There are many ways to represent the predicates in a database, such as by structured files representing tables, spreadsheet subsections, etc. In the language **Prolog**, one of the ways to represent a predicate is just by **enumerating** all combinations of values for which the predicate is true. Let us define the predicates *mother* and *father* in this fashion. These predicates provide a way of modeling the family "tree" on the right.

```
mother(alice, tom).
mother(alice, carol).
mother(carol, george).
mother(carol, heather).
mother(susan, hank).

father(john, tom).
father(john, carol).
father(fred, george).
father(fred, heather).
father(george, hank).
```

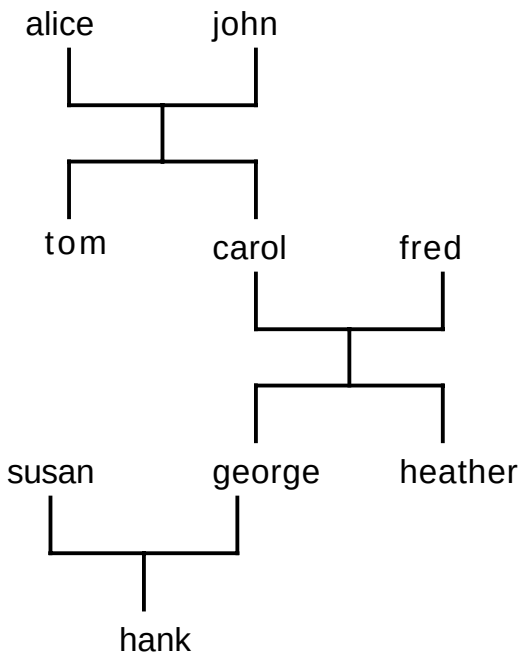


Figure 163: A family "tree" modeled as two predicates, `mother` and `father`.

It is possible for a query to contain no variables, in which case we would expect an answer of 1 or 0. For example,

```
mother(susan, hank)      ⇒ true

mother(susan, tom)       ⇒ false
```

More interestingly, when we put variables in the queries, we expect to get values for those variables that satisfy the predicate:

```
mother(alice, X)  ⇒ X = tom;  X = carol     (two alternatives for X)

father(tom, X)    ⇒ false                   (no such X exists)

mother(X, Y)      ⇒                         (several alternative combinations for X, Y)
      X = alice, Y = tom;
      X = alice, Y = carol;
      X = carol, Y = george;
      X = carol, Y = heather;
      X = susan, Y = hank
```

Note that the `X` and `Y` values must be in correspondence. It would not do to simply provide the set of `X` and the set of `Y` separately.

Defining *grandmother* using Prolog

The Prolog language allows us to present queries and have them answered automatically in a style similar to the above. Moreover, Prolog allows us to define new predicates using logic rather than enumeration.

Such a predicate is defined by the following logical expression:

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

Here :- is read as "if" and the comma separating *mother* and *parent* is read as "and". This says, in effect, "X is the grandmother of Y if X is the mother of (some) Z and Z is the parent of Y". We have yet to define *parent*, but let's do this now:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Here we have two separate logical assertions, one saying that "X is the parent of Y if X is the mother of Y", and the other saying a similar thing for father. These assertions are not contradictory, for the connective :- is "if", not "if and only if". However, the collection of all assertions with a given predicate symbol on the *lhs* exhausts the possibilities for that predicate. Thus, the two rules above together could be taken as equivalent to:

`parent(X, Y)` iff (`mother(X, Y)` or `father(X, Y)`)

Given these definitions in addition to the database, we now have a "knowledge base" since we have rules as well as enumerations. We can query the defined predicates in the same way we queried the enumerated ones. For example:

```
grandmother(alice, Y)    ⇒    Y = george; Y = heather

grandmother(X, Y)        ⇒    X = alice, Y = george;
                              X = alice, Y = heather
                              X = carol, Y = hank

grandmother(susan, Y)    ⇒    false
```

Quantifiers

Quantifiers are used in predicate logic to represent statements that range over **sets** or "domains" of individuals. They can be thought of as providing a very concise notation for what might otherwise be large conjunctive ($\wedge$) expressions and disjunctive ($\vee$) expressions.

Universal Quantifier ∀ ("for all", "for every")
(∀ x) P(x) means **for every x** in the domain of discourse P(x) is true.

Existential Quantifier ∃ ("for some", "there exists")
(∃ x) P(x) means **for some x** in the domain of discourse P(x) is true.

If the domain can be enumerated $\{d_0, d_1, d_2, ...\}$ (and this isn't always possible) then the following are suggestive

$$(\forall x)\, P(x) \equiv (P(d_0) \wedge P(d_1) \wedge P(d_2) \wedge ...)$$

$$(\exists x)\, P(x) \equiv (P(d_0) \vee P(d_1) \vee P(d_2) \vee ...)$$

This allows us to reason about formulas such as the of **DeMorgan's laws for Quantifiers**:

$$\neg (\forall x)\, P(x) \equiv (\exists x)\, \neg P(x)$$
$$\neg (\exists x)\, P(x) \equiv (\forall x)\, \neg P(x)$$

The definition of validity with respect to an interpretation, and thus general validity, is easily extended to formulas with quantifiers. For example, in the natural number interpretation, where the domain is {0, 1, 2, …} and > has its usual meaning, we have the following:

Formula	Meaning	Validity
$(\exists x)\ x > 0$	There is an element larger than 0.	valid
$(\forall x)\ x > x$	Every element is larger than itself.	invalid
$(\forall x)(\exists y)\ x > y$	Every element is larger than some element.	invalid
$(\forall x)(\exists y)\ y > x$	Every element has a larger element.	valid
$(\exists x)(\forall y)\ (y \mathrel{!}= x) \rightarrow x > y$	There is an element larger than every other.	invalid

Exercises

With respect to the interpretation in which:
The domain is the natural numbers
== is equality
> is greater_than
- is proper subtraction

which of the following are valid:

1 •• $x < y \rightarrow (x - z) < (y - z)$

2 •• $x < y \lor y < x$

3 •• $(x < y \lor x == y) \land (y < x \lor x == y) \rightarrow (x == y)$

Assuming that `distinct` are predicates such that `distinct(X, Y)` is true when the arguments are different, express rules with respect to the preceding Prolog database that define:

4 •• `sibling(X, Y)` means `X` and `Y` are siblings (different people having the same parents)

5 •• `cousin(X, Y)` means that `X` and `Y` are children of siblings

6 ••• `uncle(X, Y)` means that `X` is the sibling of a `Z` such that `Z` is the parent of `Y`, or that `X` is the spouse of such a `Z`.

7 ••• `brother_in_law(X, Y)` means that `X` is the brother of the spouse of `Y`, or that `X` is the husband of a sibling of `Y`, or that `X` is the husband of a sibling of the spouse of `Y`.

Which of the following are valid for the natural numbers interpretation?

8 • $(\exists x)\ (x \mathrel{!}= x)$

9 • $(\forall y)(\exists x)\ (x \mathrel{!}= y)$

10 • $(\forall y)(\exists x)\ (x = y)$

11 •• $(\forall y)(\forall x)\ (x = y) \lor (x > y) \lor (x < y)$

12 •• $(\forall y)\ [\ (y = 0) \lor (\exists x)\ (x < y)]$

13 •• $(\forall y)\ [\ (y = 0) \lor (\exists x)\ (x > y)]$

Bounded Quantifiers

Two variants on quantifiers are often used because they conform to conversational usage. It is common to find statements such as

> "For every x such that ...., P(x)."

For example,

> "For every even x > 2, not_prime(x)."

Here the .... represents a condition on x. The added condition is an example of a "bounded quantifier", for it restricts the x values being considered to those for which .... is true. However, we can put .... into the form of a predicate and reduce the bounded quantifier case to an ordinary quantifier. Let Q(x) be the condition "packaged" as a predicate. Then

"For every x such that Q(x), P(x)."

is equivalent to

$(\forall x)\ [Q(x) \rightarrow P(x)]$

Similarly, existential quantifiers can also be bounded.

"For some x such that Q(x), P(x)."

is equivalent to

$(\exists x)\ [Q(x) \wedge P(x)]$

Note that the bounded existential quantifier translates to an "and", whereas the bounded universal quantifier translates to an "implies".

Quantifiers and Prolog

Prolog does not allow us to deal with quantifiers in a fully general way, and quantifiers are never explicit in prolog. Variables that appear on the lefthand side of a Prolog rule (i.e. to the left of :-) are implicitly quantified with $\forall$. Variables that appear only on the righthand side of a rule are quantified as around the righthand side itself. For example, above we gave the definition of grandmother:

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

With explicit quantification, this would appear as:

$(\forall x)(\forall y)$ [grandmother(X, Y) **if** $(\exists Z)$ mother(X, Z) and parent(Z, Y)]

The reason that this interpretation is used is that it is fairly natural to conceptualize and that it corresponds to the *procedural interpretation* of Prolog rules.

Logic vs. Procedures

Although Prolog mimics a subset of predicate logic, the real semantics of Prolog have a **procedural** basis. That is, it is possible to interpret the logical assertions in Prolog as if

they were a kind of generalized procedure call. This duality means that Prolog can be used as both a procedural language (based on actions) and as a declarative language (based on declarations or assertions). Here we briefly state how this works, and in the process will introduce an important notion, that of *backtracking*.

To a first approximation, a Prolog rule is like an ordinary procedure: The lefthand side is like the header of a procedure and the righthand side like the body. Consider, then, the rule

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

Suppose we make the "call" (query)

```
grandmother(alice, Y)
```

Satisfying this predicate becomes the initial "goal". In this case, the call matches the *lhs* of the rule. The body is detached and becomes

```
mother(alice, Z), parent(Z, Y)
```

This goal is read: "Find a `Z` such that `mother(alice, Z)`. If successful, using that value of `Z`, find a `Y` such that `parent(Z, Y)`. If that is successful, `Y` is the result of the original query."

We can indeed find a `Z` such that `mother(alice, Z)`. The first possibility in the definition of *mother* is that `Z = tom`. So our new goal becomes `parent(tom, Y)`. We then aim to solve this goal. There are two rules making up the "procedure" for `parent`:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Each rule is tried in turn. The first rule gives a body of `mother(tom, Y)`. This goal will fail, since `mother` is an enumerated predicate and there is no `Y` of this form. The second rule gives a body of `father(tom, Y)`. This goal also fails for the same reason. There being no other rules for parent, the goal `parent(tom, Y)` fails, and that causes the body

```
mother(alice, Z), parent(Z, Y)
```

to *fail for the case* `Z = tom`. Fortunately there are other possibilities for `Z`. The next rule for mother indicates that `Z = carol` also satisfies `mother(alice, Z)`. So then we set off to solve

```
parent(carol, Y).
```

Again, there are two rules for parent. The first rule gives us a new goal of

```
mother(carol, Y)
```

This time, however, there is a `Y` that works, namely `Y = george`. Now the original goal has been solved and the solution `Y = george` is returned.

10.3 Backtracking Principle

The trying of alternative rules when one rule fails is called **backtracking**. Backtracking also works to find multiple solutions if we desire. We need only pretend that the solution previously found was not a solution and backtracking will pick up where it left off in the search process. Had we continued in this way, Y = heather would also have produced as a solution. The arrows below suggest the path of backtracking in the procedural interpretation of Prolog. One can note that the backtracking paradigm is strongly related to *recursive descent* and *depth-first search*, which we will have further occasion to discuss.

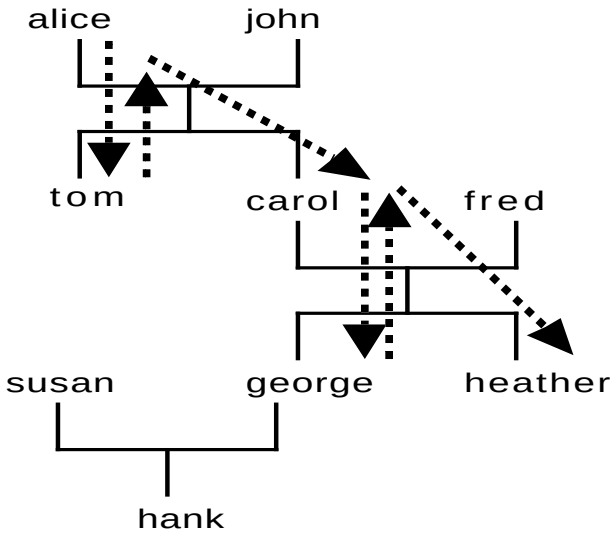


Figure 164: Backtracking in Prolog procedures

Recursive Logic

We close this section by illustrating a further powerful aspect of Prolog: rules can be recursive. This means that we can combine the notion of backtracking with recursion to achieve a resulting language that is strictly more expressive than a recursive functional language. At the same time, recursive rules retain a natural reading in the same way that recursive functions do.

Earlier we gave a rule for `grandmother`. Suppose we want to give a rule for `ancestor`, where we agree to count a parent as an ancestor. A primitive attempt of what we want to accomplish is illustrated by the following set of clauses:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z1), parent(Z1, Y).
ancestor(X, Y) :- parent(X, Z1), parent(Z1, Z2), parent(Z2, Y).
...
```

The only problem is that this set of rules is infinite. If we are going to make a program, we had better stick to a finite set. This can be accomplished if we can use `ancestor` recursively:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This pair of rules provides two ways for `X` to be an ancestor of `Y`. But since one of them is recursive, an arbitrary chain of parents can be represented. In the preceding knowledge, all of the following are true:

```
parent(alice, carol), parent(carol, george), parent(george, hank)
```

It follows logically that

```
ancestor(george, hank)
ancestor(carol, hank)
ancestor(alice, hank)
```

so that a query of the form `ancestor(alice, Y)` would have `Y = hank` as one of its answers.

Using Backtracking to Solve Problems

In chapter Compute by the Rules, we gave an example of a program that "solved" a puzzle, the Towers of Hanoi. Actually, it might be more correct to say that the *programmer* solved the puzzle, since the program was totally deterministic, simply playing out a pre-planned solution strategy. We can use backtracking for problems that are not so simple to solve, and relieve the programmer of some of the solution effort. Although it is perhaps still correct to say that the programmer is providing a strategy, it is not as clear that the strategy will work, or how many steps will be required.

Consider the water jugs puzzle presented earlier. Let us use Prolog to give some logical rules for the legal moves in the puzzle, then embed those rules into a solution mechanism that relies on backtracking. For simplicity, we will adhere to the version of the puzzle with jug capacities 8, 5, and 3 liters. The eight liter jug begins full, the others empty. The objective is to end up with one of the jugs containing 4 liters.

When we pour from one jug to another, accuracy is ensured only if we pour all of the liquid in one jug into the other *that will fit*. This means that there two limitations on the amount of liquid transferred:

(a) The amount of liquid in the source jug

(b) The amount of space in the destination jug

Thus the amount of liquid transferred is the minimum of those two quantities.

Let us use terms of the form

```
jugs(N8, N5, N3)
```

to represent the state of the system, with Ni liters of liquid in the jug with capacity i. We will define a predicate => representing the possible state transitions. The first rule relates to pouring from the 8 liter jug to the 5 liter jug. The rule can be stated thus:

```
jugs(N8, N5, N3) => jugs(M8, M5, N3) :-
    N8 > 0,
    S5 is 5 - N5,
    min(N8, S5, T),
    T > 0,
    M8 is N8 - T,
    M5 is N5 + T.
```

The conjunct `N8 > 0` says that the rule only applies if something is in the 8 liter jug. S5 is computed as the space available in the 5 liter jug. Then T is computed as the amount to be transferred. However, `T > 0` prevents the transfer of nothing from being considered a move. Finally, M8 is the new amount in the 8 liter jug and M5 is the new amount in the 5 liter jug. The 3 liter jug is unchanged, so N3 is used in both the "before" and "after" states. The predicate *min* yields as the third argument the minimum of the first two arguments. Its definition could be written:

```
min(A, B, Min) :-
    A =< B,
    Min is A.
min(A, B, Min) :-
    A > B,
    Min is B.
```

In a similar fashion, we could go on to define rules for the other possible moves. We will give one more, for pouring from the 3 liter to the 5 liter jug, then leave the remaining four to the reader.

```
jugs(N8, N5, N3) => jugs(N8, M5, M3) :-
    N3 > 0,
    S5 is 5 - N5,
    min(N3, S5, T),
    T > 0,
    M3 is N3 - T,
    M5 is N5 + T.
```

The rules we have stated are simply the constraints on pouring. They do not solve any problem. In order to do this, we need to express the following recursive rule:

> A solution from a final state consists of the empty sequence of moves.
>
> A solution from a non-final state consists of a move from the state to another state, and a solution from that state.

In order to avoid re-trying the same state more than once, we need a way to keep track of the fact that we have tried a state before. We will take a short-cut here and use Prolog's device of dynamically asserting new logical facts. In effect, we are building the definition of a predicate on the fly. Facts of the form `marked(State)` will indicate that *State* has already been tried. The conjunct `\+marked(State1)` says that we have not tried the state before. So as soon as we determine that we have not tried a state, we indicate that we are now trying it. Then we use predicate move as constructed above, to tell us the new state and recursively call solve on it. If successful, we form the list of moves by combining the move used in this rule with the list of subsequent moves.

```
solve(State1, []) :-
    final(State1).            % final state reached, success

solve(State1, Moves) :-
    \+marked(State1),               % continue if state not tried
    assert(marked(State1)),         % mark state as tried
    (State1 => State2),             % use transition relation
    solve(State2, More),                        % recurse
    Moves = [move(State1, State2) | More].      % record sequence
```

The following rules tell what states are considered final and initial:

```
initial(jugs(8, 0, 0)).

final(jugs(4, _N5, _N3)).

final(jugs(_N8, 4, _N3)).
```

When we call, in Prolog, the goal

```
    initial(State),
    solve(State, Moves).
```

two distinct move sequences are revealed, one shorter than the other, as shown be the following solution tree.

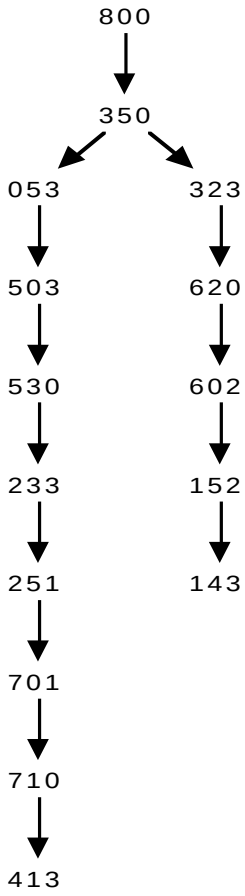


Figure 165: A tree of solutions for a water jug puzzle

We'll discuss further aspects of solving problems in this way in the chapter Computing with Graphs. Earlier, we stated the recursion manifesto, which suggests using recursion to minimize work in solving problems. The actual problem solving code in this example, exclusive of the rules for defining legal transitions, is quite minimal. This is due both to recursion and backtracking. So the **Prolog programmers' manifesto** takes things a step further:

> **Let recursion and backtracking do the work for you.**
>
> *Prolog programmers' manifesto*

Backtracking in "Ordinary" Languages

This is somewhat of a digression from logic, but what if we don't have Prolog available to implement backtracking? We can still program backtracking, but it will require some "engineering" of appropriate control. The basic feature provided by backtracking is to be

able to try alternatives and if we reach failure, have the alternatives available so that we may try others. This can be accomplished with just recursion and an extra data structure that keeps track of the remaining untried alternatives in some form. In many cases, we don't have to keep a list of the alternatives explicitly; if the alternatives are sufficiently well-structured, it may suffice to be able to generate the next alternative from the current one.

A case in point is the classical *N-queens problem*. The problem is to place N queens on an N-by-N chessboard so that no two queens attack one another. Here two queens attack each other if they are in the same row, same column, or on a common diagonal. Below we show a solution for N = 8, which was the output from the program we are about to present.

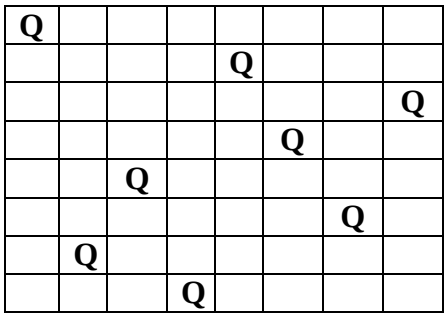


Figure 166: A solution to the 8-queens problem

To solve this problem using backtracking, we take the following approach: Clearly the queens must all be in different rows. We call these the "first" queen, "second" queen, etc. according to the row dominated by that queen. So it suffices to identify the columns for the queens in each row. Thus we can proceed as follows:

Place the first queen in the first unoccupied row.
Place the second queen in the next unoccupied row so that it doesn't attack the first queen.
Place the third queen in the next unoccupied row so that it doesn't attack the first two queens.
....

Continuing in this way, one of two things will eventually happen: We will reach a solution, or we will be unable to place a queen according to the non-attacking constraint. In the latter case, we *backup* to the most recent discretionary placement and try the *next* alternative column, and proceed forward from there. The current problem is well structured: the next alternative column is just the current alternative + 1. So we can accomplish pretty much all we need by the mechanism of recursion.

Here is the program:

```
import java.io.*;
import Poly.Tokenizer;

/* N Queens puzzle solver: The program accepts an integer N and produces a
 *  solution to the N Queens problem for that N.
 */

class Queens
  {
  int N;                // number of rows and columns

  int board[];          // board[i] == j means row i has a queen on column j

  static int EMPTY = -1;        // value used to indicate an empty row


  Queens(int N)                         // construct puzzle
    {
    this.N = N;
    board = new int[N];                 // create board
    for( int i = 0; i < N; i++ )
      board[i] = EMPTY;                 // initialize board
    }


  public static void main(String arg[]) // test program
    {
    Poly.Tokenizer in = new Poly.Tokenizer(System.in);
    int token_type;
    while( prompt() && (token_type = in.nextToken()) != Poly.Tokenizer.TT_EOF )
      {
      if( token_type != Poly.Tokenizer.TT_LONG || in.lval <= 0 )
        {
        System.out.println("Input must be a positive integer");
        continue;
        }

      Queens Puzzle = new Queens((int)in.lval);

      if( Puzzle.Solve() )      // solve the puzzle
        {
        Puzzle.Show();
        }
      else
        {
        System.out.println("No solutions for this size problem");
        }
      }
    System.out.println();
    }


  static boolean prompt()               // prompt for input
    {
    System.out.print("Enter number of queens: ");
    System.out.flush();
    return true;
    }


 boolean Solve()                       // try to solve the puzzle
    {
```

```
  return Solve(0);
  }

//
// Solve(row) tries to solve by placing a queen in row, given
// successful placement in rows < row. If successful placement is not
// possible, return false.
//

boolean Solve(int row)
  {
  if( row >= N )
    return true;                      // queens placed in all rows, success
  for( int col = 0; col < N; col++ )  // Try each column in turn
    {
    if( !Attack(row, col) )           // Can we place in row, col?
      {
      board[row] = col;               // Place queen in row, col.
      if( Solve(row+1) )              // See if this works for following rows
        return true;                  // success
      else
        board[row] = EMPTY;           // undo placement, didn't work
      }
    }
  return false;                       // no solution found for any column
  }


// see if placing in row, col results in an attack given the board so far.

boolean Attack(int row, int col)
  {
  for( int j = 0; j < row; j++ )
    {
    if( board[j] == col || Math.abs(board[j]-col) == Math.abs(j-row) )
      return true;
    }
  return false;
  }


// show the board

void Show()
  {
  int col;

  for( col = 0; col < N; col++ )
    System.out.print(" _");
  System.out.println();

  for( int row = 0; row < N; row++ )
    {
    for( col = 0; col < board[row]; col++ )
      System.out.print("|_");
    System.out.print("|Q");
    for( col++; col < N; col++ )
      System.out.print("|_");
    System.out.println("|");
    }
  System.out.println();
  }
}  // Queens
```

Functional Programming is a form of Logic Programming

Prolog includes other features beyond what we present here. For example, there are predicates for evaluating arithmetic expressions and predicates for forming and decomposing lists. The syntax used for lists in rex is that used in Prolog. We have said before that functions are special cases of predicates. However, functional programming does not use functions the way Prolog uses predicates; most functional languages cannot "invert" (solve for) the arguments to a function given the result. In another sense, and this might sound contradictory, functions are a special case of predicates: An n-ary function, of the form $D^n \rightarrow R$, can be viewed as an (n+1)-ary predicate. If *f* is the name of the function and *p* is the name of the corresponding predicate, then

$$f(x_1, x_2, ..., x_n) == y \text{ iff } p(x_1, x_2, ..., x_n, y)$$

In this sense, we can represent many functions as Prolog predicates. This is the technique we use for transforming rex rules into Prolog rules. A rex rule:

$$f(x_1, x_2, ..., x_n) => rhs.$$

effectively becomes a Prolog rule:

$p(x_1, x_2, ..., x_n, y)$:-
... expression determining y from *rhs* and $x_1, x_2, ..., x_n$, !.

The ! is a special symbol in Prolog known as "cut". Its purpose is to prevent backtracking. Recall that in rex, once we commit to a rule, subsequent rules are not tried. This is the function of cut.

Append in Prolog

In rex, the `append` function on lists was expressed as:

```
append([ ], Y) => Y;
append([A | X], Y) => [A | append(X, Y)];
```

In Prolog, the counterpart would be an `append` predicate:

```
append([ ], Y, Y) :- !.

append([A | X], Y, [A | Z]) :- append(X, Y, Z).
```

In Prolog, we would usually *not* include the cut (!), i.e. we *would* allow backtracking. This permits `append` to solve for the lists being appended for a given result list. For example, if we gave Prolog the goal append(X, Y, [1, 2, 3]), backtracking would produce four solutions for X, Y:

```
X = [ ], Y = [1, 2, 3];
X = [1], Y = [2, 3];
X = [1, 2], Y = [3];
X = [1, 2, 3], Y = [ ]
```

10.4 Using Logic to Specify and Reason about Program Properties

One of the important uses of predicate logic in computer science is specifying what programs are supposed to do, and convincing oneself and others that they do it. These problems can be approached with varying levels of formality. Even if one never intends to use logic to prove a program, the techniques can be useful in thinking and reasoning about programs. A second important reason for understanding the principles involved is that easy-to-prove programs are usually also easy to understand and "maintain"† . Thinking, during program construction, about what one has to do to prove that a program meets its specification can help guide the structuring of a program.

Program Specification by Predicates

A standard means of specifying properties of a program is to provide two **predicates** over variables that represent input and output of the program:

> **Input Predicate**: States what is assumed to be true at the start of the program.
>
> **Output Predicate**: States what is desired to be true when the program terminates.

For completeness, we might also add a third predicate:

> **Exceptions Predicate**: State what happens if the input predicate is not satisfied by the actual input.

For now, we will set aside exceptions and focus on input/output. Let us agree to name the predicates **In** and **Out**.

Factorial Specification Example

> int n, f; (This declares the types the variables used below.)

---

† The word "maintenance" is used in a funny way when applied to programs. Since programs are not mechanical objects with frictional parts, etc., they do not break or wear out on their own accord. However, they are sometimes unknowingly released with bugs in them and those bugs are hopefully fixed retroactively. Also, programs tend not to be used as is for all time, but rather evolve into better or more comprehensive programs. These ideas: debugging and evolution, are lumped into what is loosely called program "maintenance".

In(n): n >= 0 (States that n >= 0 is assumed to be true at start.)

Out(n, f): f == n! (States that f == n! is desired at end.)

Programs purportedly satisfying the above specification:

```
/* Program 1: bottom-up factorial*/

f = 1;
k = 1;
while( k <= n )
  {
  f = f * k;
  k = k + 1;
  }
```

The program itself is almost independent of the specification, except for the variables common to both. If we had encapsulated the program as a function, we could avoid even this relationship.

```
/* Program 2: top-down factorial*/

f = 1;
k = n;
while( k > 1 )
  {
  f = f * k;
  k = k - 1;
  }


/* Program 3: recursive factorial*/

f = fac(n);

where

long fac(long n)
{
if( n > 1 )
  return n*fac(n-1);
else
  return 1;
}
```

Each of the above programs computes factorial in a slightly different way. While the second and third are superficially similar, notice that the third is not tail recursive. Its multiplications occur in a different order than in the second, so that in some ways it is closer to the first program.

Proving Programs by Structural Induction

"Structural induction" is induction along the lines of an inductive data definition. It is attractive for functional programs. Considering program 3 above, for example, a structural induction proof would go as follows:

> Basis: Prove that *fac* is correct for n == 1 and n == 0.
>
> Induction: Assuming that *fac* is correct for argument value n-1, show that it is correct for argument value n.

For program 3, this seems like belaboring the obvious: Obviously fac gives the right answer (1) for arguments 0 and 1. It was designed that way. Also, if it works for n-1, then it works for n, because the value for n is just n times the value for n-1.

The fact that functional programs essentially are definitions is one of their most attractive aspects. Many structural induction proofs degenerate to observations.

In order to prove programs 1 and 2 by structural induction, it is perhaps easiest to recast them to recursive programs using McCarthy's transformation. Let's do this for Program 2:

```
fac(n) = fac(n, 1);

fac(k, f) => k > 1 ? fac(k-1, f*k) : f;
```

Again, for `n == 0` or `n == 1`, the answer is `1` by direct evaluation.

Now we apply structural induction to the 2-argument function. We have to be a little more careful in structuring our claim this time. It is that:

> (∀`f`) `fac(k, f)` ⇒ `f * k!`.

We arrived at this claim by repeated substitution from the rule for fac:

> fac(k, f) ⇒
> fac(k-1, f*k) ⇒
> fac(k-2, f*k*(k-1)) ⇒
> fac(k-3, f*k*(k-1)*(k-2)) ⇒...

Why we need the quantification of for all values of f is explained below. When called with `k == 0` or `k == 1` initially, the result f is given immediately. But `k! == 1` in this case, so `f == f*k!`.

Now suppose that `k > 1`, we have the inductive hypothesis

> (∀`f`) `fac(k-1, f)` ⇒ `f * (k-1)!`.

and we want to show

(∀f) `fac(k, f)` ⇒ `f * k!`.

For any value of f, the program returns the result of calling `fac(k-1, f*k)`. By the inductive hypothesis, the result of this call is `(f * k) *(k-1)!`. But this is equal to `f * (k * (k-1)!)`, which is equal to `f * k!`, what we wanted to show.

The quantification (∀f) was necessary so that we could substitute `f*k` for `f` in the induction hypothesis. The proof would not be valid for a fixed `f` because the necessary value of `f` is different in the inductive conclusion.

Now let's look at an example not so closely related to traditional mathematical induction. Suppose we have the function definition in rex:

```
shunt([ ], M) => M;

shunt([A | L], M) => shunt(L, [A | M]);
```

This definition is a 2-argument auxiliary for the reverse function:

```
reverse(L) = shunt(L, []);
```

We wish to show that `shunt` as intended, namely:

The result of `shunt(L, M)` is that of appending `M` to the reverse of `L`.

In symbols:

(∀`L`) (∀`M`) `shunt(L, M)` ⇒ `reverse(L) ^ M`

where ⇒ means *evaluates to* and `^` means *append*.

To show this, we structurally induct on one of the two arguments. The choice of which argument is usually pretty important; with the wrong choice the proof simply might not work. Often, the correct choice is the one in which the list dichotomy is used in the definition of the function, in this case the first argument `L`. So, proceeding with structural induction, we have

Basis `L == [ ]`: (∀`M`) `shunt([ ], M)` ⇒ reverse([ ]) `^ M`)

The basis follows immediately from the first rule of the function definition; `shunt([ ], M)` will immediately rewrite to `M`. and M == reverse([ ]) `^ M`.

Induction step `L == [A | N]`: The inductive hypothesis is:

(∀`M`) `shunt(N, M)` ⇒ `reverse(N) ^ M`

and what is to be shown is:

(∀M) `shunt([A | N], M) ⇒ reverse([A | N]) ^ M`

From the second definition rule, we see that the `shunt([A | N], M)` rewrites to

```
shunt(N, [A | M])
```

From our inductive hypothesis, we have

```
shunt(N, [A | M]) ⇒ reverse(N) ^ [A | M]
```

because of quantification over the argument M. Now make use of an equality

```
reverse(N) ^ [A | M] == reverse([A | N]) ^ M
```

which gives us what is to be shown

To be thorough, the equality used would itself need to be established. This can be done by appealing to our inductive hypothesis: Notice that the *rhs* of the equality is equivalent to `shunt([A | N], [ ]) ^ M`, by the equation that defines `reverse`. According to the second definition rule, this rewrites to `shunt(N, [A]) ^ M`. But by our inductive hypothesis, this evaluates to (`reverse(N) ^ [A]) ^ M`, which is equivalent to *lhs* of the equality using associativity of `^` and the equality `[A] ^ M == [A | M]`. If desired, both of these properties of `^` could be established by secondary structural induction arguments on the definition of `^`.

Proving Programs by Transition Induction

Transition induction takes a somewhat different approach from structural induction. Instead of an inductive argument on the data of a functional program, the induction proceeds along the lines of how many transitions have been undertaken from the start of the program to the end, and in fact, to points intermediate as well.

A common variation on the transition induction theme is the method of "loop invariants". A loop invariant is a logical assertion about the state of the program at a key point in the loop, which is supposed to be true whenever we get to that point. For a while loop or a for loop, this point is just before the test, i.e. where the comment is in the following program:

initialization

`while`(/* *invariant* */ *test*)

body

For example, in factorial program 2 repeated below with the invariant introduced in the comment, the loop invariant can be shown to be

```
k > 0 && f == n! / k!
```

```
/* Program 2: top-down factorial*/

f = 1;
k = n;
while( /* assert:  k > 0 && f == n! / k! */ k > 1 )
  {
  f = f * k;
  k = k - 1;
  }
```

There are two main issues here:

1. Why the loop invariant is actually invariant.

2. Why the loop invariant's truth implies that the program gives the correct answer.

Let us deal with the second issue first, since it is the main reason loop invariants are of interest. The loop will terminate only when the test condition, k > 1 in this case, is false. But since k is assumed to have an integer value and we have the assertion k > 0, this means that `k == 1` when the loop terminates. But we also have the assertion `f == n! / k!`. Substituting `k == 1` into this, we have `f == n!`, exactly what we want to be true at termination.

Now the first issue. Assume for the moment that `n > 0` when the program is started. (If `n == 0`, then the loop terminates immediately with the correct answer.) Essentially we are doing induction on the number of times the assertion point is reached. Consider the first time as a basis: At this time we know `f ==1` and `k == n`. But `n > 0`, so the `k > 0` part of the assertion holds. Moreover, `n! / k! == 1`, and f == 1 because we initialized it that way. So `f == n! / k!` and the full assertion holds the first time.

Inductively, suppose the assertion holds now and we want to show that it holds the next time we get to the key point, assuming there will be a next time. For there to be a next time, `k > 1`, since this is the loop condition. Let `f'` be the value of `f` and `k'` be the value of k the next time. We see that `f' == f*k` and `k' == k-1`. Thus `k' > 0` since `k > 1`. Moreover, `f' == f*k == (n! / k!)*k == n! / (k-1)! == n! / k'!`, so the second part of the invariant holds.

This completes the proof of program 2 by transition induction. Note one distinction, however. Whereas structural induction proved the program terminated and gave the correct answer, transition induction did not prove that the program terminated. It only proved that *if* the program terminates, the answer will be correct. We have to go back and give a second proof of the termination of program 2, using guess what? Essentially

structural induction! However, the proof is easier this time: it only needs to show termination, rather than some more involved logical assertion. We essentially show:

> The loop terminates for `k <= 1`. This is obvious.
>
> If the loop terminates for `k-1`, it terminates for `k`. This is true because `k` is replaced by `k-1` in the loop body.

Further Reflection on Program Specification

Note that the input specification for factorial above is `n >= 0`. Although we could run the programs with values of n not satisfying this condition, no claims are made about what they will do. A given program could, for example, do any of the following in case `n < 0`:

> a) Give a "neutral" value, such as 1, which is the value of `f(0)` as well.
>
> b) Give a "garbage" value, something that is based on the computation that takes place, but is relatively useless.
>
> c) Fail to terminate.

The problem with actually specifying what happens in these non-standard cases is that it commits the programmer to satisfying elements of a specification that are possibly arbitrary. It may well be preferable to "filter" these cases from consideration by an appropriate input specification, which is what we have done.

Another point of concern is that the output specification `f == n!` alludes to there being some *definition* of `n!`. For example, we could give a definition by a set of rex rules. But if we can give the rex rules, we might not need this program, since rex rules are executable. This concern can be answered in two ways: (i) Having more than one specification of the solution to a problem such that the solutions check with each other increases our confidence in the solutions. (ii) In some cases, the output specification will not specify the result in a functional way but instead will only specify properties of the result that could be satisfied by a number of different functions. Put another way, we are sometimes interested in a program that is just *consistent* with or *satisfies* an input-output *relation*, rather than computing a specific function.

Finally, note that specifying the factorial program in the above fashion is a sound idea only if we can be assured that `n` is a **read-only variable**, i.e. the program cannot change it. Were this not the case, then it would be easy for the program to satisfy the specification without really computing factorial. Specifically, the program could instead just consist of:

```
n = 1;
f = 1;
```

Certainly `f = n!` would then be satisfied at end, but this defeats our intention for this program. If we declared in our specification

```
read_only: n
```

then the above program would not be legitimate, since it sets n to a value.

Another way to provide a constraining specification by introducing an **anchor variable** that does not appear in the program. Such variables are read-only by definition, so we might declare them that way. For factorial, the specification would become, where n_0 is the initial value of n and does not occur in the program proper:

```
int n, n0, f;

read_only: n0

In(n0):  n == n0  ∧ n0 >= 0

Out(n0, f):  f = n0!
```

This doesn't look all that much different from the original specification, but now the "short-circuit" program

```
n = 1;
f = 1;
```

does *not* satisfy the specification generally. It only does so in the special case where n_0 `== 0` or n_0 `== 1`, since only then is n_0`! == 1`. We can remind ourselves that anchor variables are read-only

Array Summation Specification Example

A specification for a program summing the elements of an array `a`, from `a[0]` through `a[n-1]`.

```
float a[];
float max;
int n;

read_only: a, n

In(n, a): n >= 0

Out(n, a, max): max == sum(i = 0 to n-1, a[i])
```

Here we have introduced a notation *sum* to indicate the result of summing an array. As with some of the other examples, this specification would probably be enough to serve as

a second solution to the problem if sum were a valid programming construct. An example of a program purporting to satisfy the specification is:

```
s = 0;
k = 0;
while( k < n )
  {
  s = s + a[k];
  k++;
  }
```

Were it not for the `read_only` specification, we could satisfy the output predicate by merely setting `n` to `0` or by setting all elements of `a` to `0`, and setting `s` to `0`.

Using Quantifiers over Array Indices

An array is typically an arbitrarily-large collection of data values. As such, we cannot refer to each value by name in a specification; we must resort to quantifiers to talk about all of the elements of an array.

Array Maximum Example

As an example, consider the specification of a program for computing the maximum of an array in a variable `max`. Here two things are important for `max`:

> The value of `max` should be `>=` each array element.
>
> The value of `max` should be `==` some array element.

So the output assertions will be:

```
(∀i) max >= a[i]

(∃i) max == a[i]
```

where the array bounds are understood, or to make the bounds explicit:

```
(∀i) (i >= 0 && i < n) → max >= a[i]

(∃i) (i >= 0 && i < n) && max == a[i]
```

The complete specification would then be:

```
float a[];
float s;
int n;

read_only: a, n

In(n, a): n >= 0
```

```
Out(n, a, s): ( (∀i) (i >= 0 && i < n) → max >= a[i] )
              && ( (∃ i) (i >= 0 && i < n) && max == a[i] )
```

Array Sorting Example

The following is an example wherein the specification would not readily translate into a solution of the problem (e.g. using rex). Also, since we intend to rearrange the values of an array in place, we cannot use the read_only annotation for the array itself. We must instead introduce a new read-only variable that represents the original array contents. We will use equal(a, b, n) to designate that *a* and *b* have the same values, element-by-element, from 0 through n.

Array Sorting specification:

```
float a[], a0[];

int n;

read_only a0;

In(n, a, a0): n >= 0 && equal(a, a0, n)

Out(n, a0, a):  permutation(a, a0, n) && sorted(a, n)
```

For the sorting specification, we used two auxiliary predicates to express **Out**. By `permutation(a, a0, n)` we mean that the elements of `a` are the same as those of a_0, except possibly in a different order (their contents are the same when they are viewed as "bags"). By `sorted(a, n)` we mean that the elements of a are in non-decreasing order. We can express `sorted` in a logical notation as:

```
sorted(a, n) is (∀i) ( ( 0 <= i  ∧ i < n-1 ) → ( a[i] <= a[i+1] ) )
```

Expressing permutation is messier, due to the need to handle possibly duplicated elements. If we introduce a notation for counting the number of a given element, say #(e, a, n) meaning the **number of occurrences** of e in a, we could define:

`permutation(a, a0, n)` is

```
(∀i) (∀e)
     ( 0 <= i  ∧ i < n) → ( e == a[i] → #(e, a, n) == #(e, a0, n) )
```

We could give an appropriate rex-like rules for #(e, a, n):

```
#(e, a, -1) => 0;

#(e, a, i) => ( e == a[i] )? 1 + #(e, a, i-1);
```

```
#(e, a, i) => #(e, a, i-1);
```

The above rules would read: The number of times `e` occurs in `a[0]....a[n-1]` is `0`. For `i>= 0`, the number of times `e` occurs in `a[0]....a[i]` is 1 more than the number of times it occurs in `a[0]....a[n-1]` if `e == a[i]`. Otherwise it is the same as the number of times it occurs in `a[0]....a[n-1]`.

The fact that we need such recursive expressions, which are effectively programs themselves in an appropriate language, dampens our hope that specifications and programs can be totally distinct domains of endeavor. Indeed, writing a specification has much in common with writing a program. In the former, however, we hope for greater succinctness through the use of an appropriate specification language, such as the predicate calculus.

Correctness Defined

Given an input/output specification and a program intended to satisfy that specification, we now focus on what it *means* to satisfy a specification. Some terminology is helpful.

> **Partial Correctness**: A program P is said to be *partially correct* with respect to a specification (a pair of predicates **In**, **Out**) in case that:
>
> > If the program is started with variables satisfying **In** (and ip (instruction pointer) at the initial position), then *when and if the program terminates*, the variables will satisfy **Out**.

Notice the "when and if" disclaimer. Nothing is being claimed for cases where the program does not terminate,

> **Termination**: A program P is said to *terminate* with respect to a specification if
>
> > If the program is started with variables satisfying **In** (and ip at the initial position), then the program will terminate (i.e. will reach a state where its ip is at the final position).
> >
> > (Termination does not use the **Out** part of the specification.)
>
> **Total Correctness**: A program P is *totally correct* with respect to a specification if
>
> > The program is both partially correct and terminates with respect to that specification.

There are reasons why we separate partial correctness and termination in this way:

(i) Some programs cannot be guaranteed to terminate, so are partially correct at best.

(ii) Sometimes it is easier to prove partial correctness and termination separately.

Partial Correctness

The Floyd Assertion Principle

This principle is perhaps the easiest-to-understand way to prove partial correctness. (A special case of this method is the "**loop invariant**" idea introduced earlier.) We demonstrate it using the flowchart model for programs. Each of the nodes in the program flowchart is annotated with an **assertion**. The intent of the assertion is to **represent information about the state of the program at that particular node**, when and if the ip reaches that node.

Partial correctness is established by proving a set of **verification conditions** (**VC**s) associated with the invariants, the enabling conditions on the arcs, and the assignments on the arcs.

The beauty of this method is that, *if* the assertions are valid, the VCs can be proved individually in isolation without referring back to the original program. Here is how a VC relates to an arc in a program: Suppose that the following is a piece of the flowchart, where A_i and A_j are assertions, E is an enabling condition, and F represents the assignment being done.

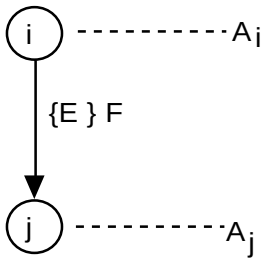


Figure 167: Flowchart fragment where nodes i and j have been annotated with assertions. E represents the enabling condition that must be satisfied for the ip (instruction pointer) to move from i to j, while F represents the change of state variables that will occur when the ip moves from i to j.

Specifically, express F as an equation between primed and unprimed versions of the program variables, representing the values before and after the statement is executed, respectively. Let A'_j be assertion A_j with all variables primed. Then the prototype **verification condition** for this arc is:

$$(A_i \wedge E \wedge F) \rightarrow A'_j$$

which is interpreted as follows: If the program's ip is at i with variables satisfying assertion A_i and the enabling condition E is satisfied, and if F represents the relation between variables before and after assignment, then A'_j holds for the new values. The names given to A_i and A'_j are **pre-condition** and **post-condition**, respectively.

Floyd Assertion Example

Consider the following fragment of a flowchart, which has been annotated with assertions at places i and j.

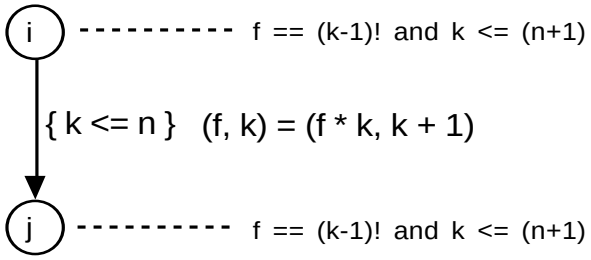


Figure 168: Fragment of a flowchart program, showing enabling condition, action, and possible assertions at nodes i and *j*

The verification condition $(A_i \wedge E \wedge F) \rightarrow A'_j$ in this case has the following parts:

A_i: f == (k-1)! and k <= (n+1)
E: k <= n
F: (f', k') == (f * k, k + 1)
A'_j: f' == (k'-1)! and k' <= (n+1)

Notes:

F represents a parallel assignment to f and k. The primed values indicate the values after the assignment, while the unprimed ones indicate the values before.

n is a read-only variable, so no primed value is shown for it.

A_i and A'_j are the same assertion, except that A'_j has its *k* and *f* variables primed, to denote their values after the assignment rather than before.

Spelled out more fully, if ξ represents the vector of all program variables, then the general enabling condition and assignment will take the form

$$\{ E(\xi) \}\ \xi = F(\xi)$$

while the verification condition for the arc is:

$$(A_i(\xi) \wedge E(\xi) \wedge \xi' == F(\xi)) \rightarrow A_j(\xi')$$

In summary, in the verification condition, we use an equality between primed variables and a function of unprimed variables to represent the effect of an assignment to one and the same set of program variables. The reason for choosing this approach is that we don't have any other way of relating assignment to a statement in predicate logic.

We continue the example by providing the verification conditions for all of the arcs of the compact factorial program, repeated here for convenience. The VC that was indicated above will be recognized as that for the arc going from node 1 to node 1.

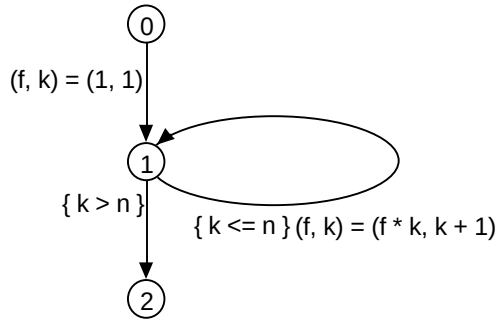


Figure 169: Compact flowchart for the factorial program

Using A_0, A_1, A_2 to represent the assertions at each node, the verification conditions, one per arc are:

arc $0 \rightarrow 1$: $(A_0 \wedge true \wedge (f', k') == (1, 1)) \rightarrow A'_1$

arc $1 \rightarrow 1$: $(A_1 \wedge k <= n \wedge (f', k') == (f*k, k+1)) \rightarrow A'_1$

arc $1 \rightarrow 2$: $(A_1 \wedge k > n \wedge (f', k') == (f, k)) \rightarrow A'_2$

To complete the proof, we also need to insure that:

$In \rightarrow A_0$ and $A_{exit} \rightarrow Out$

where A_{exit} is the assertion at the exit point. In this and most cases, this is implied by just *equating* A_0 to **In** and A_{exit} to **Out**.

Before we can actually conduct the proof, we must choose the remaining assertions A_i. The guidelines for doing this are as follows:

> A_i should be always true for the program state whenever the instruction pointer points to i (*i.e.* each A_i should be "invariant").

Let us try to choose appropriate assertions for the factorial example. If we equate A_0 to In and A_2 to Out, i.e.

A_0: n >= 0

A_2: f = n!

we only have to determine an appropriate A_1. Let us try as the value of A_1 the assertion

f == (k-1)! ∧ k <= (n+1)

By looking at the state-transition diagram, we can get support for the idea that this condition is invariant. The VCs can now be filled in:

VC_{01} arc 0 → 1: $(A_0 \wedge true \wedge (f', k') == (1, 1)) \rightarrow A'_1$

assertion at place 0; arc enabling condition; arc assignment; assertion at place 1 (primed variables)

i.e. (n >= 0 ∧ *true* ∧(f', k') == (1, 1))
→ (f' == (k'-1)! ∧ k' <= (n+1))

[*n* does not get primed, as it is a read_only variable.]

[VC_{11} was discussed earlier.]

VC_{11} arc 1 → 1: $(A_1 \wedge k <= n \wedge (f', k') == (f*k, k+1)) \rightarrow A'_1$

assertion at place 1; arc enabling condition; arc assignment; assertion at place 1 (primed variables)

i.e. (f == (k-1)! ∧ k <= (n+1) ∧k <= n ∧(f', k') == (f*k, k+1))
assertion at place 1

→ (f' == (k'-1)! ∧ k' <= (n+1))
assertion at place 1 (primed variables)

VC_{12} arc 1 → 2: $(A_1 \wedge k > n \wedge (f', k') == (f, k)) \rightarrow A'_2$

i.e. (f == (k-1)! ∧ k <= (n+1) ∧k > n ∧(f', k') == (f, k))
assertion at place 1

→ f' = n!
assertion at place 2 (primed variables)

Example Proofs of Verification Conditions

We have now translated the partial correctness of the program into three logical statements, the VCs. The proof of the three VCs is straightforward and we can take them in any order. Since each is of the form H $\rightarrow$ C (hypothesis implies conclusion), we shall assume the hypothesis and show the conclusion.

VC_{01} assume: (n >= 0 $\wedge$*true* $\wedge$(f', k') == (1, 1))
show: (f' == (k'-1)! $\wedge$ k' <= (n+1))

By the rightmost equation in the assumption (f' == 1, k' == 1), what we are to show follows from a simpler equivalent:

1 == (1 - 1)! $\wedge$ 1 <= (n + 1)

The left conjunct simplifies to 1 == 0! and is true since 0! == 1 by definition of factorial. The right conjunct 1 <= (n + 1) follows from n >= 0.

VC_{11} assume: (f == (k-1)! $\wedge$ k <= (n+1)$\wedge$k <= n $\wedge$(f', k') == (f*k, k+1))
show: (f' == (k'-1)! $\wedge$ k' <= (n+1))

By the rightmost equation in the assumption, what we are to show follows from a simpler equivalent:

f*k == ((k+1)-1)! $\wedge$ (k+1) <= (n+1)

i.e.

f*k == k! $\wedge$ (k+1) <= (n+1)

The left conjunct follows from the assumption that f == (k-1)! and the definition of factorial. The right conjunct follows from the assumption k <= n. [Note that the assumption k <= (n+1) was subsumed by k <= n in this VC, and therefore was not of any particular use.]

VC_{12} assume: (f == (k-1)! $\wedge$ k <= (n+1) $\wedge$k > n $\wedge$(f', k') == (f, k))
show: f' = n!

What we are to show is equivalent, using the equation f' == f, to f = n!. This will follow from the assumption that f == (k-1)! if we could establish that k = n+1. But we have k <= (n+1) and k > n in our assumption. Since we are working in the domain of *integers*, this implies k = n+1.

Having proved these VCs, we have established the partial correctness of our factorial program.

A Note on Choice of Assertions

Although A_i does not have to completely characterize the state whenever the instruction pointer points to i, it must characterize it sufficiently well that all of the VCs can be proved. The possible pitfalls are:

> If A_i is chosen to be too weak (i.e. too near to universally true), then some successor post-condition might not be provable.
>
> If A_i is chosen to be too strong (i.e. too near to false), then it might not be possible to prove it from some predecessor pre-condition.

Termination

Termination proofs proceed by an additional sort of reasoning from partial correctness proofs. One method, which we call the **energy function method** involves constructing an expression E in terms of the program variables that:

> E never has a value less than 0
>
> On each iteration of any loop, E decreases.

For the second factorial program,

```
f = 1;
k = n;
while( k > 1 )
  {
  f = f * k;
  k = k - 1;
  }
```

it is very clear that the expression `k-1` by itself decreases on each iteration and that 0 is its minimum, since any attempt to make `k-1` less than 0 (i.e. `k` less than `1`) will cause the loop to terminate.

For the first factorial program,

```
f = 1;
k = 1;
while( k <= n )
  {
  f = f * k;
  k = k + 1;
  }
```

the energy function we want is `n - k + 1`. The value of this expression decreases on each iteration, since k increase. Moreover, if n - k is small enough, the condition k <= n, which is the same as `n - k + 1 > 0`, is no longer true, so the loop will terminate.

Perspective

For most individuals, the real value of understanding the principles of program correctness is not mostly for proving programs. More importantly, constructing a program as if correctness had to be proved will give us better-structured programs. Poorly structured programs are not only hard to prove; they are hard to understand and hard to build upon.

10.5 Use of Assertions as a Programming Device

Some languages provide for the inclusion of assertions in the program text. The idea is that the program asserts a certain predicate should be true at the point where the assertion is placed. At execution time, if the assertion is found to be false, the program terminates with an error message. This can be useful for debugging. Various C and C++ libraries include assert.h, which provides such an assertion facility. The idea is that

```
assert( expression );
```

is an executable statement. The expression is evaluated. If it is not true (non-zero), then the program exits, displaying the line number containing the assert statement. This is a useful facility for debugging, but it is limited by the fact that the assertion must be expressed in the programming language. The kinds of assertions needed to make this generally useful require substantial functions that mimic predicate logic with quantifiers. The only way to achieve these is to write code for them, which sometimes amounts to solving part of the problem a second time.

10.6 Program Proving vs. Program Testing

There is no question that programs must be thoroughly tested before during development. However, it is well worth keeping mind a famous statement by E.W. Dijkstra:

Testing can demonstrate the presence of bugs, but it can never prove their absence.

The only situation in which this is not true is when a program can be exhaustively tested, for every possible input. But even for inputs restricted to a finite set, the number of possibilities is impractically large. Consider the number of combinations for a 32-bit multiplier, for example.

10.7 Using Assertional Techniques for Reasoning

Understanding the basis for assertion-based verification can help with reasoning about programs. The form of reasoning we have in mind includes

> "if an assertion is known to be true at one place in the program, what can we say is true at some other place?"
>
> "if an assertion must be true at one place in the program, what must be true at some other place in order to insure that the first assertion is true?"

Although we will use textual programs to illustrate the principles, it might be helpful to think in terms of the corresponding graph model.

Conditional statements

```
if( P )
  statement-1
```

Assuming that P as a procedure call has no side-effects, we know that P as an assertion is true before statement-1. More generally, if assertion A is also true before the *if* statement (and P has no side-effects), we know that A $\wedge$ P is true before statement-1.

```
if( P )
  statement-1
else
  statement-0
```

Under the same assumptions, we know that A $\wedge \neg$P before statement-0.

While Statements

```
while( P )
  statement-1
```

Assuming that P has no side-effects, P will be true before each execution of statement-1. Also, $\neg$P will be true *after* the overall *while* statement. (Even if A is true before the *while* statement, we cannot be sure that A is true before statement–1 the next time around, unless P implies A.)

In general, if B is true before the *while* statement, and statement-1 reestablishes B, then B will also be true on exit.

We can summarize this reasoning as follows:

If we can prove a verification condition

$(B \wedge P \wedge \text{statement-1}) \rightarrow B$

then we can infer the verification condition

$B \wedge (\text{while P statement-1}) \rightarrow (B \wedge \neg P)$

Here again, B is called a "loop invariant".

Example – While statement

A typical pattern occurs in the factorial program. The loop condition is k <= n. The invariant B includes, k+1 <= n. On exit, we have the negation of the loop condition, thus k > n. Together $(B \wedge \neg P)$ give k == n+1.

Reasoning by Working Backward

A somewhat systematic way to derive assertions internal to a program is to work backward from the exit assertion. In general, suppose we know that A is a post-condition that we want to be true after traversing an arc. We can derive a corresponding pre-condition that gives the minimal information that must be true in order for A to be true. This pre-condition will depend on the enabling predicate E and the assignment F for the corresponding arc, as well as on A. Since it depends on these three things, and entails the vector of program variables ξ, it is noted as wlp(A, E, F)(ξ), where wlp stands for **"weakest liberal precondition"**. A little thought will show that wlp(A, E, F)(ξ) can be derived as:

$$\text{wlp}(A, E, F)(\xi) \equiv (E(\xi) \rightarrow A(F(\xi)))$$

In other words, wlp(A, E, F)(ξ) is true just in case that whenever the enabling condition E(ξ) is satisfied, we must have A satisfied for the resulting state after assignment.

Notice that in relation to A, wlp(A, E, F) will always satisfy the verification condition for the corresponding arc, that is, we can substitute A for A_j and wlp(A, E, F) for A_i in the prototypical verification condition:

$(A_i \wedge E \wedge F) \rightarrow A'_j$

and end up with a true logical statement. Let's try it. Substituting the formula

claimed for wlp in place of A, we have:

$((E(\xi) \rightarrow A(F(\xi))) \wedge E(\xi) \wedge \xi' = F(\xi)) \rightarrow A(\xi')$

Suppose the overall hypothesis is true. Then from $E(\xi) \rightarrow A(F(\xi))$ and $E(\xi)$, we get $A(F(\xi))$. But from the equation $\xi' = F(\xi)$, we then have $A(\xi')$.

Weakest Liberal Precondition Examples

(given) statement	(given) post-condition	wlp
x = y + 5;	x > 0	true $\rightarrow$ y + 5 > 0, *i.e.* y > -5
x = x + 5;	x == 0	true $\rightarrow$ x + 5 = 0 *i.e.* x == -5
x = x + y;	x == 0	true $\rightarrow$ x + y = 0 *i.e.* x + y = 0
[x > y] x++;	x > 0	x > y $\rightarrow$ (x + 1) > 0
[x > y] x = x - y;	x > 0	x > y $\rightarrow$ (x -y) > 0 *i.e.* true
[x > y] x++;	y > x	x > y $\rightarrow$ y > (x + 1) *i.e.* false
[x > y] y++;	x > y	x > y $\rightarrow$ x > (y + 1) *i.e.* x > (y + 1)

A wlp of *false* says that the given post-condition cannot be achieved for that particular statement. A wlp of *true* says that the given post-condition can always be achieved, independent of the variable state before the statement.

Exercises

1 •• Consider the following program that computes the square of a number without using multiplication. Devise a specification and show that the program meets the specification by deriving an appropriate loop invariant.

```
static long square(long N)
{
long i, sum1, sum2;
sum1 = 0;
sum2 = 1;

for( i = 0; i < N; i++ )
  {
  sum1 += sum2;
  sum2 += 2;
  }
return sum1;
}
```

The technique shown in this and the next problem, generalizes to computing any polynomial using only addition. This is called "finite differences" and is the basis of Babbage's *difference engine*, an early computer design. It works based on the observation that an integer squared is always the sum of a contiguous sequence of odd numbers. For example,

25 == 1 + 3 + 5 + 7 + 9 (sum of the first 5 odd numbers)

This fact can be discovered by looking at the "first differences" of the sequence of squares: they are successive odd numbers. Furthermore, the first differences of those numbers (the "second differences" of the squares") are uniformly 2's. For any n-th degree polynomial, if we compute the n-th differences, we will get a constant. By initializing the "counter" variables differently, we can compute the value of the polynomial for an arbitrary argument by initializing these constants appropriately.

2 •• Consider the following program, which computes the cube of a number without using multiplication. Devise a specification and show that the program meets the specification by deriving an appropriate loop invariant.

```
static long cube(long N)
{
long i, sum1, sum2, sum3;
sum1 = 0;
sum2 = 1;
sum3 = 6;

for( i = 0; i < N; i++ )
  {
  sum1 = sum1 + sum2;
  sum2 = sum2 + sum3;
  sum3 = sum3 + 6;
  }
return sum1;
}
```

3 ••• Consider the following Java code:

```
// assert X == X0

polylist L = X;
polylist R = NIL;
while( /* */ !null(L) )
  {
  R = cons(first(L), R);
  L = rest(L);
  }

// assert R == reverse(X0)
```

Here *reverse* denotes the usual list reversal function. Note that we can apply reverse to both sides of the equality in the final assertion to get `R == reverse(X0)`, since for any list `R` , `reverse(reverse(R)) == R`. In other words, we are asserting that this code reverses the original list. What loop invariant would you assert at /* */ in order to establish that the final assertion follows from the initial assertion? (You may make use of the functions such as *reverse* and *append* in your loop invariant, as well as "obvious" identities for these functions.) Give an argument that shows that the final assertion follows from the loop invariant, and that the proposed invariant really is invariant.

4 ••• For any properties of functions such as *reverse* and *append* you used in the preceding problem, prove those properties by structural induction on appropriate functional programs for those functions. An example of such a property is:

$(\forall X)$ `reverse(reverse(X)) == X`

where it is assumed that the domain of `X` is that of lists.

5 ••• Devise a square-root finding program based on the squaring program above. Provide a specification and show the correctness of the program.

6 •• Show that the array summation program is totally correct with respect to its specification.

7 ••• Show that the array maximum program is totally correct with respect to its specification.

8 •••• Show that the sorting program is totally correct with respect to its specification.

10.8 Chapter Review

Define the following terms:

assert library
assignment
backtracking
DeMorgan's laws for quantifiers
energy function
existential quantifier
Floyd assertion principle
interpretation
N-queens problem
partial correctness
post-condition
pre-condition

predicate
quantifier
structural induction
termination
total correctness
transition induction
universal quantifier
valid
verification condition

10.9 Further Reading

Babbage, H.P. (ed.) *Babbage's Calculating Engines*. London, 1889.

Jon Barwise and John Etchemendy, *The Language of First-Order Logic*, Center for the Study of Language and Information, Stanford, California, 1991. [Introduction to proposition and predicate logic, including a Macintosh™ program "Tarski's World. Easy to moderate.]

W.F. Clocksin and C. S. Mellish, *Programming in Prolog*, Third edition, Springer-Verlag, Berlin, 1987. [A readable introduction to Prolog. Easy.]

R.W. Floyd, *Assigning meanings to programs*, Proc. Symp. Appl. Math., 19, in J.T. Schwartz (ed.), Mathematical Aspects of Computer Science, 19-32, American Mathematical Society, Providence, R.I., 1967. [Introduction of the Floyd assertion principle.]

Cordell Green, *The Application of Theorem Proving to Question Answering Systems*, Ph.D. Thesis, Stanford University Computer Science Department, Stanford, California, 1969. [Seminal work on the connection between logic and program proving.]

R.M. Keller. *Formal verification of parallel programs*. Communications of the ACM, **19**, 7, 371-384 (July 1976). [Applies assertions to general transition-systems. Moderate.]

Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974. [Examples using Floyd's Assertion Principle. Moderate.]

11. Complexity

11.1 Introduction

This chapter focuses on issues of program running time, including how to measure and analyze programs for their running time, as well as provide examples of techniques for improving program performance. Examples are taken from the areas of sorting and searching.

The pragmatic aspects of computing require one to be cognizant of the resource-usage aspects of a program. While such concerns should be secondary to those of the correctness of the program, they are nonetheless concerns that, like correctness, can make the difference between success and failure in computer problem solving. The general term used by computer scientists to refer to resource usage is "complexity". This term refers not to how complex the *program* is, i.e. how difficult it is to understand, but rather how much *resources* are consumed when the program is *executed.* Indeed, the least difficult to understand program might be fairly profligate in its use of resources.

The process of making a program more "efficient" unfortunately often has the effect of making it harder to understand. To develop a program *to a first approximation*, the following axiom might be applied.

Get it right first, *then* make it faster.

In particular, this axiom should be applied when considering small incremental improvements in code, which can shave off some fraction of execution time, but which make the program obscure and more difficult to debug.

The greater thrust of this chapter, however, is algorithmic improvements, that is make a program faster by choice or development of a better algorithm. Coding a new algorithm can be like starting afresh with respect to "getting it right" however. For this reason, it is best to have designed the overall program as a set of modules, with "plug replaceability" between a simple but slower module and a faster one.

11.2 Resources

By "resource", we typically are concerned with one or more of the following:

> **Execution time**: This is the time it takes a program to process a given input. Time is considered a resource for at least two reasons:

The time spent waiting for a solution (by a human, or by some other facet of automation) is time that could be put to other productive uses. In this sense, time is not the actual resource, but is instead reflective of resources that might go unused while waiting.

Thinking of the computer as providing a service, there is a limitation on the amount of service that can be provided in a given time interval. Thus programs that execute longer use up more of this service.

Memory space: This is the space used by a program to process a given input. Memory space used translates into cost of computation in the sense that memory costs money and the ability to use a certain amount of memory directly depends on the memory available.

Memory space could be further sub-divided along the lines of a *memory hierarchy*, some form of which is found in most computer systems:

Main memory: Semiconductor memory in which most of the program and data are stored when the program is running.

Cache memory: Very high-speed semiconductor memory that "caches" frequently-used program and data from main memory.

Paging memory: Slower memory, usually disk, which in simplistic terms serves as kind of "overflow" or "swapping" area for the main memory.

File memory: Disk or tape memory for file objects used by a program .

In these notes, our primary focus will be on execution time as the resource Some consideration will be given to memory requirements as well. As we shall see, it is often possible to "trade off" time resources for memory resources and vice-versa.

11.3 The Step-Counting Principle

Most often we will be interested in relative execution-time comparisons between two or more algorithms for solving a given problem. Rather than dealing with the actual times a computer would spend on an algorithm, we try to use a measure that is relatively insensitive to the particular computer being used. While the speeds of various primitive operations, such as addition, branching (change of control from one point in the program to another), etc. may vary widely, we make the assumption that for purposes of comparing algorithms on a given computer, we can just count the **number** of each kind of operation during execution, rather than be concerned with the actual times of those operations. This is not to say that every occurrence of a given kind of operation takes the same time; there will be a general dependency on the values of the arguments as well. However, for purposes of getting started, we make the assumption that the count is an

adequate measure. We could then get an overall time estimate by multiplying the counts of various operations by the time taken by those operations and summing over all n different kinds of operations:

$$\text{Execution time} = \sum_{i=1}^{n} \text{count(Operation i)*time(Operation i)}$$

Straight-line Programs

Straight-line programs are programs with no branching: every operation in the program is executed. Thus the execution time is the same regardless of data. For example, consider the straight-line program:

```
a = b*c + d;
c = d/e + f;
f = a*c;
```

Here there are two multiply operations, one divide, and two additions. Thus the total time would be computed as

execution time =

```
  2*time(multiply)
+ 1*time(divide)
+ 2*time(add)
+ 3*time(assign)
```

where time(assign) refers to the time to assign a value to a variable explicitly.

Loop Programs

Very few programs of substance will be straight-line. Typically we have loops, the execution of which will depend on the data itself. In this case, the total time depends on the data. Consider

```
sum = 0;
for( i = 0; i < N; i++ )
  sum = sum +  i*i;
```

Here the number of times the loop body is executed will be N. Therefore, there will be N multiply operations. There will also be N additions in the loop body, as well as N additions of the form i++, and N comparisons i < N. We have as total execution time:

execution time =
2*time(assign) +
N*[time(multiply) + time(add) + time(compare) + time(increment) + time(assign)]

Recursive Programs

As we know, loop programs can be represented as recursive programs. However, recursive programs also have the possibility of "non-linear" recursion, making them sometimes more of a challenge to analyze. Consider the computation of the rex program `sum(1, N)` where `N >= 0`.

```
sum(M, N) => M >= N ? M;
sum(M, N) => K = (M+N)/2, sum(M, K) + sum(K+1, N);
```

`sum(M, N)` computes the sum of integers `M` through `N` by dividing the range into two, until the range is empty. The value of sum is obtained by summing the results of recursive calls to the function.

The following tree shows how the recursion decomposes for `sum(1, 6)`:

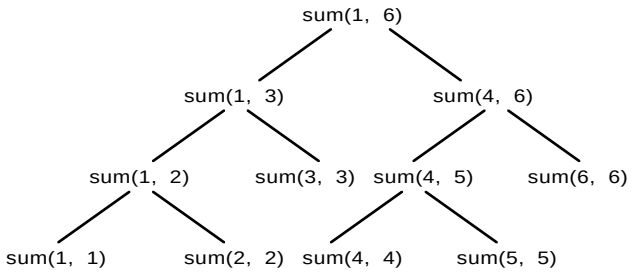


Figure 170: Tree of a recursive decomposition

The tree representation for the program's execution of `sum(1, N)` will have `N` leaves and therefore `N-1` interior nodes, i.e. `2*N-1` nodes altogether. For each node there will be a comparison to determine which rule to use, for a total of `2*N-1` comparisons. For each interior node, there will be 3 additions, and one division by 2. So overall we have

execution time =
(N-1)*[3*time(add) + time(divide)]
+ (2*N-1)*time(compare)

Here we are ignoring any overhead required to do function calls, and are assuming that the times to do the basic operations are constant, i.e. independent of the argument sizes. This is only an approximation to reality, especially if we have to deal with arbitrarily-large arguments. If time(add) = time(divide) = time(compare) = 1, then the total time is

4*(N - 1) + 2*N - 1

= 6*N - 5

The analysis above assumes that we already understand the algorithm well enough to see that a tree is involved, and that we know how to analyze a tree. An alternative approach that doesn't make such assumptions is to derive a recurrence formula for time patterned after the rules, but with the data size as an argument. In this case, the "size" is the range of numbers to be summed. For the basis case, there is only a comparison, so we have:

T(1) => time(compare);

For the induction rule, we make the simplifying assumption that the range is of even length, so we can divide it in half:

```
T(2*N) =>
        time(compare) + 3*(time add) + 1*time(divide) + 2*T(N);
```

Again assuming that all operations take the same time, we get

```
T(1) => 1;

T(2*N) => 5 + 2*T(N);
```

For example, to sum 8 numbers,

```
T(8)  ==> 5 + 2*T(4)
      ==> 5 + 2*(5 + 2*T(2))
      ==> 5 + 2*(5 + 2*(5 +2*T(1)))
      ==> 5 + 2*(5 + 2*(5 +2*1))
      ==> 43
```

which agrees with our earlier calculation of 6*N-5 when N = 8. We can also see that there is agreement for general N that is repeatedly divisible by 2. Such a number must be a power of 2, $N = 2^k$. Let $S(k) = T(2^k)$. Then we have the equivalent recurrence

```
S(0) ==> 1;

S(k+1) ==> 5 + 2*S(k);
```

We can "solve" this recurrence by successive substitutions:

```
S(k)  ==> 5 + 2*S(k-1)
      ==> 5 + 2*(5 + 2*S(k-2))
      ==> 5 + 2*(5 + 2*(5 + 2*S(k-3)))
      ==> ...
```

until the argument to S is reduced to 0. This will obviously occur after k substitutions, so

$$
\begin{aligned}
S(k) &= 5 + 2*S(k-1) \\
&= 5 + 2*(5 + 2*S(k-2)) \\
&= 5 + 2*5 + 2^2*S(k-2) \\
&= 5 + 2*5 + 2^2*5 + 2^3*S(k-3) \\
&= 5 + 2*5 + 2^2*5 + 2^3*5 + 2^4*S(k-4) \\
&\quad\ \ .... \\
&= 5*(1 + 2 + 2^2 + 2^3 + ... + 2^{k-1}) + 2^k * S(0) \\
&= 5*(2^k - 1) + 2^k \\
&= 6*2^k - 5
\end{aligned}
$$

11.4 Profiling

Many systems provide a software tool known as a "profiler". Such a tool counts executions of procedures and the places from which they are called. Using it, one can get an idea of how much time overall is being spent in various procedures, and thus possibilities for where to devote attention in improving run time.

A specific example, using the Java interpreter with -prof option will put profile results from the run in a file `java.prof`.

Let's suppose that we have the following break down of the time devoted to various pieces of code A, B, C:

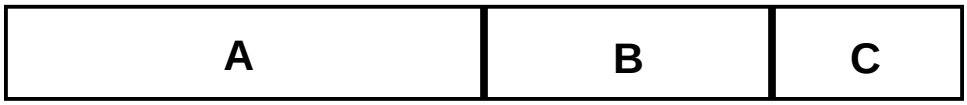


Figure 171: Execution time profile

The suggestion is that A takes about 50% of the time, B 30%, and C 20%. The question is where to concentrate code improvements to reduce the execution time? Intuitively we should concentrate on A, because there we stand to achieve the biggest reduction. But how much improvement can we get from A alone? In the very best case, we could eliminate A altogether. This would result in a 50% reduction in execution time. On the other hand, if we eliminated C altogether, we would still have 80% of the time we did before, or only a 20% reduction.

Such considerations are quantified in a rule known as Amdahl’s law. In general, if chunk *A* takes fraction *f* of the overall time, then the speedup achieved by reducing *A* to 0 is at most 1/(1-*f*). So, execution of *A* would have to occupy about 90% of the execution to enable a 10-fold reduction in execution time if *A* were eliminated completely. Amdahl's law was originally derived for application to parallel computing and we'll see more about it in the chapter *Limitations of Computing*.

11.5 Suppressing Multiplicative Constants

Quite often we make the further simplifying assumption that the operation times are the same for all operations. While this may seem like a drastic oversimplification, it is useful for comparative purposes. If every operation requires a constant time, then each time can be expressed as some factor times one of the operations. Thus, in assuming all operations have the same time, the resulting time estimate will be off by a factor that is at most the maximum of these factors. For reasons to be explained, it is defensible to make the assumption that all times are the same, so long as it is clear that this assumption is being made. With this assumption, we would have the following for the above examples:

straight-line example: execution time = 8 steps

loop example: execution time = 5*N + 2 steps

recursive example: execution time = 6*N - 5 steps

11.6 Counting Dominant Operations

In many cases, we can get an idea of the execution time by simply focusing on the number of dominant operations. For example, in the loop program, we could focus on the number of multiplies or the number of times the loop body is executed. In both cases, we would end up with an execution time of N steps. In the recursive program, we could count the number of times the recursive rule is used, which would give us N-1 steps.

11.7 Growth-Rate

Although we may, on occasion, engage in estimating time for a *specific* input to a program, in general we will be interested in a much broader measure to give an idea of the quality of the program or its algorithm. Such a measure is found in the form of *growth-rate comparisons*, as we now discuss.

Most programs are designed to work with not just a single input, but rather with a wide, and usually infinite, set of input possibilities. We often can associate a measure of the input, usually in the form of a parameter that implies the *size* of the input. Some examples are:

Program application	**Possible measure(s) of input**
word processing	number of characters in the document, or number of editing commands
solving linear equations	number of equations, and/or number of unknowns
sorting an array	number of elements in the array
displaying a scene graphically	number of polygons in the scene

With each type of program, we try to focus on one key measure in which to express the program's performance. We try to express the programs resource usage, e.g. execution time, as a function of this measure. For example, in the loop program above, we could use the value N as the measure. We derived that

execution time = 5*N + 2 steps

With slight modification, we can convert that program into one that sums the squares of an array of N elements:

```
sum = 0;
for( i = 0; i < N; i++ )
      sum = sum + a[i]*a[i];
```

Now the input measure is equated to the *size* of the array.

Now consider sorting an array, using the following minimum-selection sort algorithm expressed in Java. (Here calling the constructor on an array of doubles sorts the array in place; the object created can then be discarded).

```
class minsort
{
private double array[];       // The array being sorted
int N;                        // The length of the prefix to be sorted

  //  Calling minsort constructor on array of doubles sorts the array.
  //  Parameter N is the number of elements to be sorted (which might
  //  be fewer than are in the array itself).

  minsort(double array[], int N)
    {
    this.array = array;
    this.N = N;

    for( int i = 0; i < N; i++ )
      {
      swap(i, findMin(i));
      }
    }


   //  findMin(M) finds the index of the minimum among
   //  array[M], array[M+1], ...., array[N-1].

  int findMin(int sortFrom)
    {
    // by default, the element at minSoFar is the minimum
    int minSoFar = sortFrom;

    for( int j = sortFrom+1; j < N; j++ )
      {
      if( array[j] < array[minSoFar] )
        {
        minSoFar = j;   // a smaller value is found
```

```
      }
    }
  return minSoFar;
  }


 //  swap(i, j) interchanges the values in array[i] and array[j]

void swap(int i, int j)
  {
  double temp = array[i];
  array[i] = array[j];
  array[j] = temp;
  }
```

If we count comparisons, as in `array[j] < array[minSoFar],` as the dominant operation, then we could derive

execution time = n*(n-1)/2 steps

To see this, let us extract the loop structure essence of the program:

```
for( i = 0; i < n; i++ )
  {
  for( j = i + 1; j < n ; j++ )
    *** one step ***
  }
```

Here one step represents the comparison operation that we are counting. Now examine the number of times the inner loop body executes as a function of the outer loop index:

i = 0	j = 1, 2, ..., n - 1	n - 1 steps
i = 1	j = 2, 3, ..., n - 1	n - 2 steps
...		
i = n-1	j = n, ..., n - 1	0 steps

In total, we have 0 + 1 + 2 + ... + (n-1) steps, which sums to n*(n-1)/2. This summation can be shown by induction. This is a special case of the sum of an *arithmetic series*.

In terms of the topic of this section, we would say that the sorting program's growth-rate is represented by the function

$n \rightarrow n*(n-1)/2$

that is, the function that, with argument *n*, yields the value of n*(n –1)/2. It is important to keep in mind that the growth rate is a *function,* even though it is often written just as an *expression*

n*(n –1)/2

with the argument *n* being implicit for simplicity.

Not all programs run the same amount of time for a given input measure. For those that do not, it is common to use the *maximum* over all inputs having a given value of the measure as the growth rate function. For example, suppose we had a program that inputs strings of 0's and 1's, with the following observed execution times:

Input	Time
λ	0
0	1
1	1
00	1
01	4
10	4
11	2
000	1
001	9
010	9
011	9
100	8
101	6
110	4
111	9
...	

If we use the *length* of the input as the measure, then a growth-rate of $n \rightarrow n^2$ is suggested, even though not all inputs of length n require n^2 time. Thus, we are often content with focusing on the **worst-case** among inputs of a given value of the input measure, rather than considering all inputs, in order to get an idea of the complexity. Another way of looking at it is that the derived function forms an **envelope** around the actual executions times, or is an **upper bound** on the execution time of the algorithm. The figure below demonstrates this for the example at hand.

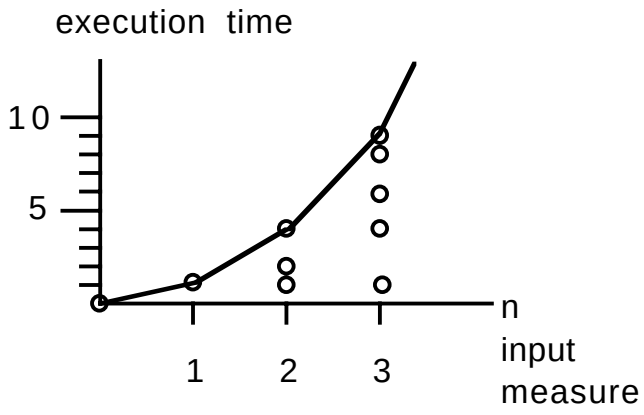


Figure 172: Execution times for a string-processing program, plotted vs. input string length. The quadratic curve is an upper-bound on the times.

11.8 Upper Bounds

In general, a function can be an **upper bound** on execution time without having all of its points correspond to actual execution times. For example, if the above hypothetical algorithm used at most 12 steps to process any input of length 4, then the upper bound of $n \rightarrow n^2$ would still be consistent. Likewise, the function $n \rightarrow n^3$ would also be an upper bound, although qualitatively a poorer one with respect to the given data.

Informally, when an upper bound fits the data points closely, we say it is a **tight** upper bound. Obviously, the tighter an upper bound is, the more information is conveyed by the statement that it is an upper bound. That is, saying that $n \rightarrow n^2$ is an upper bound conveys more information than saying that $n \rightarrow n^3$ is.

For convenience, it is common to omit the argument part of functional expressions when talking about growth rates. Thus n^3 would actually stand for the function $n \rightarrow n^3$. We will be taking this approach from here on in the discussion, except at points where it is useful to make it clear that we are talking about a function rather than just an expression.

11.9 Asymptotic Growth-Rate

A coarse, but useful, measure for comparing algorithms is based on asymptotic growth-rate. This measure has the benefit of being relatively easy to derive, since it is impervious to the making of many approximations in the derivation process. Asymptotic growth rate is a measure of goodness of the time taken by an algorithm as the value of the input measure *n* grows without bound. In computing asymptotic growth rate, we often ignore multiplicative constants in the complexity function and focus on the "rate" itself. Thus,

while an execution time measure of n^2 (i.e. the function $n \rightarrow n^2$) is obviously better than one of n^3, the asymptotic comparison would also rank $1000n^2$ (i.e. the function $n \rightarrow 1000n^2$) as being better than n^3, even though the latter is better (i.e. lower) for values of $n < 1000$, called the *crossover point*. The reason to prefer $1000n^2$ is that n^3 is only better than it for a *finite* number of values of n (assuming the input measure is an integer). For the remaining infinite number of inputs, $1000n^2$ is better. Of course, this sort of reasoning is most meaningful when the crossover point is within the range of values of n to which the algorithm is actually going to be applied.

We can simplify the task of asymptotic comparisons by observing that, in a function the value of which is a sum of terms, the sum is often **asymptotically dominated** by one of those terms. Consider for example, the function

$$n \rightarrow 1000n^2 + n^3$$

For large n, the second term dominates, in the sense that the first term becomes relatively insignificant the larger n becomes. Thus, for purposes of comparing this function to another, we can simply neglect the term $1000n^2$ in the limit. The first function, now approximated by

$$n \rightarrow n^3$$

is clearly seen to grow faster than the second function.

11.10 The "O" Notation

For purposes of comparing asymptotic growth rates, the "O" (for "order") notation has been invented†. In considering a function such as

$$n \rightarrow 1000n^2 + n^3$$

it is natural to indicate that the growth rate of that function is "on the order of" the growth-rate of the function $n \rightarrow n^3$, or for short, the function is "order of" n^3. A simple way of accomplishing this is to define a set of functions, the growth rate of each of which is no more than a certain metric times an arbitrary constant. For example,

$$O(n^3)$$

means the set of functions growing no faster than does the function $n \rightarrow cn^3$, where c is an arbitrary constant.

---

† The "O" notation is due to P.G.H. Bachmann, *Zahlentheorie*, vol. 2: *Die analytische Zahlentheorie*, B.G. Teubner, Leipzig, 1894.

If f and g are two functions,

$$f \in O(g)$$

means that f is bounded from above by g times a constant.

It is also common to see in the literature

$$f = O(g)$$

which is a slight abuse of notation, but one having the same meaning as $f \in O(g)$. It is also common to use **expressions in place of functions**. Thus, one often sees something like

$$n^2 \in O(n^3)$$

when what is really meant is the following relationship:

$$(n \rightarrow n^2) \in O(n \rightarrow n^3)$$

Examples

We have already seen that $n^2 \in O(n^3)$. We also mentioned that $1000n^2 \in O(n^3)$. As will be seen, $cn^r \in O(n^s)$ whenever $r < s$, for any constant c. The rationale for all of these can be seen by looking at the slopes of the curves as n increases without limit. Even if c is very large, cn^r will eventually be overtaken by n^s for large enough n if $r < s$. The following diagram show the case where $f \in O(g)$ even though for low values of n, *f*'s value is a significant multiple of *g*'s value.

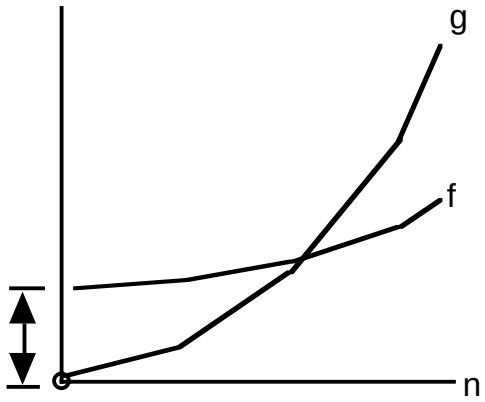


Figure 173: f ∈ O(g), assuming the indicated trends in f and g continue

We can use *g*'s algorithm for small values of n and *f*'s algorithm for large values to get the best of both worlds. We would choose between the two algorithms depending on whether $n < n_0$ where n_0 is the breakpoint, and the resulting execution time would then appear as in the following diagram.

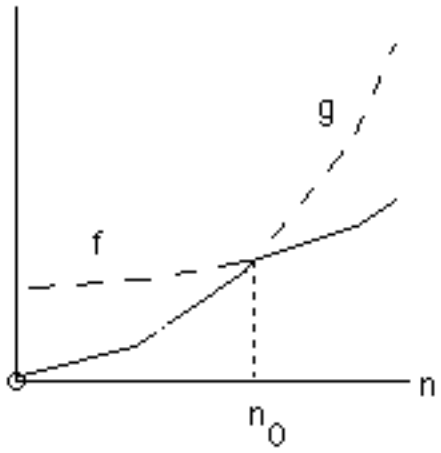


Figure 174: Combining two algorithms to get a superior execution time function

Simplified Definition of "O"

We give a definition for the **special case of functions with natural number arguments**, which allows most resource measures to be modeled, due to the fact that we almost always base the measure on the size of some input facet and the size is in integral units. Later (in the exercises) we give the more general definition, and indicate that the two definitions are consistent on the domain of natural numbers.

Let f: N $\rightarrow$ R and g: N $\rightarrow$ R be two functions with domain being the natural numbers and range being the positive real numbers. Then

$f \in O(g)$

[typically read "f is oh of g" or "f is big-oh of g"]

means

$(\exists\ c)(\forall\ n)\ \ f(n) \leq cg(n)$

This says: "there exists a constant c such that for all n, f(n) is less than or equal to c times g(n).

For the case of R as a domain, we would need to use a more complex definition for $f \in O(g)$:

$(\exists\ c)(\exists\ n_0)(\forall n > n_0\ )\ \ f(n) \leq cg(n)$

For the case of N as a domain, the two definitions are equivalent.

If we are given *f* and *g*, then in order to show that $f \in O(g)$, we need only to exhibit an appropriate c. We show this in the following examples.

Examples

$n^2 \in O(n^3)$ Take c = 1. Obviously $(\forall\ n)\ n^2 \leq 1n^3$.

$1000n^2 \in O(n^3)$ Take c = 1000. Obviously $(\forall\ n)\ 1000n^2 \leq 1000n^3$.

$n^2 + 10^6n \in O(n^2)$ Take c =$2*10^6$. We have

$$(\forall\ n)\ n^2 + 10^6n \leq 10^6n^2 + 10^6n \leq 10^6n^2 + 10^6n^2 = 2*10^6n^2$$

since $(\forall\ n)\ n \leq n^2$.

11.11 O Notation Rules

Fortunately, it is not necessary to return to first principles for every comparison of two functions. We can establish some rules that help us reason about relationships between asymptotic growth rates:

Transitivity Rule

If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.

Proof: Suppose that $f \in O(g)$ and $g \in O(h)$. Then by definition, we know that for some constants *c* and *d*:

$(\forall\ n)\ \ f(n) \leq cg(n)$

and

$(\forall\ n)\ \ g(n) \leq dh(n)$

i.e. the existence of *c* and *d* is guaranteed by our supposition and the definition of O. We must then show

$(\exists\ e)(\forall\ n)\ \ f(n) \leq eh(n)$

We can do this by exhibiting a constant *e* that makes this statement true. It appears that choosing *e* to be the product of *c* and *d* will do the job: We need to show that, for arbitrary *n*,

$$f(n) \leq cdh(n)$$

But we already have

$$f(n) \leq cg(n)$$

and

$$g(n) \leq dh(n)$$

Putting these two inequalities together gives exactly what we need.

Sum Rule

If $f \in O(h)$ and $g \in O(k)$, then $f+g \in O(\max(h, k))$.

Here we use f + g as an abbreviation for the *function* $n \rightarrow f(n) + g(n)$ and max(h, k) as an abbreviation for the function $n \rightarrow \max(h(n), k(n))$.

Proof: For convenience, define *m* to be the function $n \rightarrow \max(h(n), k(n))$. That is, for all n, $m(n) = \max(h(n), k(n))$. Assume that $f \in O(h)$ and $g \in O(k)$, to show that $(n \rightarrow f(n) + g(n)) \in O(m)$. Let c and d be constants such that

$$(\forall\ n)\ \ f(n) \leq ch(n)$$

and

$$(\forall\ n)\ \ g(n) \leq dk(n)$$

Then we have

$$(\forall\ n)\ \ f(n) + g(n) \leq \max(ch(n), dk(n))$$

Thus

$$(\forall\ n)\ \ f(n) + g(n) \leq \max(c, d)\ m(n)$$

Therefore we have found a constant, namely max(c, d), which establishes what we want to show.

The sum rule is frequently applied in program analysis. If a program consists of two parts in sequence, P; Q, with the complexity of each in terms of the input measure being

represented by functions *f* and *g*, respectively, and *h* and *k* are known upper bounds on *f* and *g*, then the function $n \rightarrow \max(h(n), k(n))$ is an upper bound on the overall program complexity. Put another way, the complexity of the combination P; Q is dominated by the part with the greater complexity. Quite often, the same part dominates for all values of the input measure.

Polynomial Rule

By applying the sum rule inductively, we can get the following:

> Let f(n) be any polynomial in n, with k being the highest exponent.
> Then $f \in O(n^k)$.

Caution: We need to be aware that polynomials have a *fixed* number of terms, i.e. the set of terms cannot be a function of n, as in the following:

Example of a Fallacious Argument: $n^3 \in O(n^2)$
Bogus Proof: $n^3 = n^2 + n^2 + \ldots + n^2$ where the sum is taken over n terms. By the polynomial rule, since the highest exponent is 2, we have $n^3 \in O(n^2)$.

Constant Absorption Rule

It is never necessary to write $f(n) \in O(dg(n))$ where *d* is a constant. It is always considered preferable to write this as $f(n) \in O(g(n))$. The argument here is that the constant d can be "absorbed into" the constant that exists by definition of $f(n) \in O(g(n))$.

Proof: Assume that $f(n) \in O(dg(n))$ where *d* is a constant, to show $f(n) \in O(g(n))$. By supposition, there is a *c* such that

$$(\forall\ n)\ \ f(n) \leq cdg(n)$$

But letting e = cd, *e* is also a constant, so

$$(\exists\ e)(\forall\ n)\ \ f(n) \leq eg(n)$$

Therefore $f(n) \in O(g(n))$.

Meaning of O(1)

When we say that $f \in O(1)$, we mean that f is O of the function $n \rightarrow 1$, i.e. the constant function 1. By the constant absorption rule, any function bounded above by a constant is O(1). So **saying f ∈ O(1) is just another way of saying that f is bounded by a constant.** To say that $f \in O(c)$ where c is some other constant is the same as saying $f \in (1)$, so we *always* write the latter.

For example, the **linear addressing principle** says that any element within an array can be accessed, given its index, in time O(1). This is an important advantage of arrays over linked lists, for which the access can only be bounded by O(n) where n is the number of elements in the list.

Multiplication Rule

The proper form of argument when the number of terms is a function of n is given by the following:

If $f \in O(g)$, then $m(n)*f(n) \in O(m(n)*g(n))$.

In terms of programs, if g provides an upper bound on the execution of the body of a loop, where n is the value of the input measure, and the loop executes m(n) times, then the function $n \rightarrow m(n)*g(n)$ provides an upper bound for the overall program.

Example – The following program has O(ng(n)) as an upper bound, assuming that the loop body does not change i.

```
for( i = 0; i < n; i++)
        .... some O(g(n)) computation ....
```

Here m(n) = n.

11.12 Analyzing Growth of Exotic Functions

The rules above give us ability to analyze some basic functions, but it does not help us handle cases that will be encountered frequently, such as ones involving *log* n [all *log*s will be assumed base 2 unless otherwise noted. However, since *log*s of different bases differ by constant factors, it would not be worthwhile differentiating them in the O notation anyway, due to the constant absorption rule.]

As an overview, it is worth establishing a framework of a few functions in a "hierarchy" of functions that often come up in algorithms:

$1 < \log n < .... < n^{1/4} < n^{1/3} < n^{1/2} < n < n \log n < n^2 < n^3 < n^4 < .... < 2^n < n!$

Each "function" in this chain is O of the next function, but not conversely. We will establish some of these relations in this section. First we show a few comparative plots to remind ourselves of how some of these functions behave.

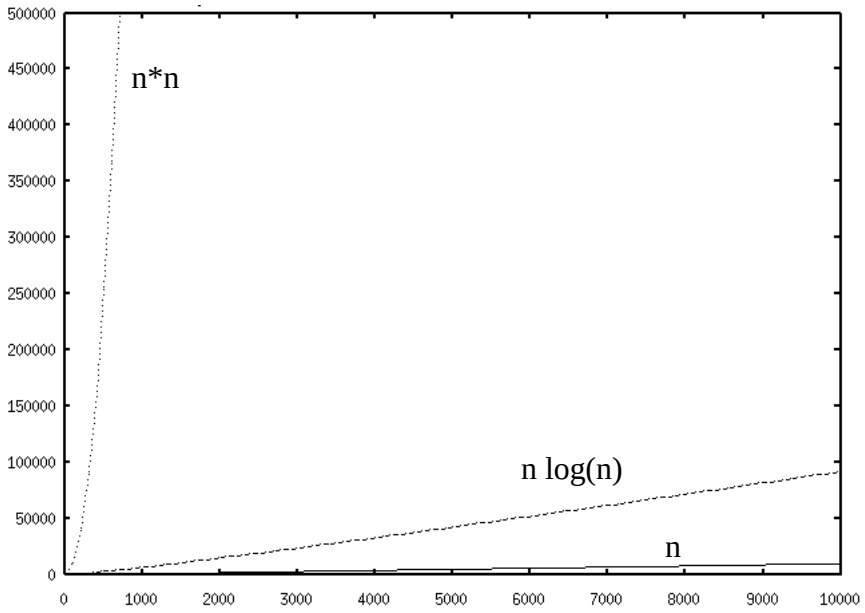


Figure 175: n^2 *vs.* n log n *vs.* n

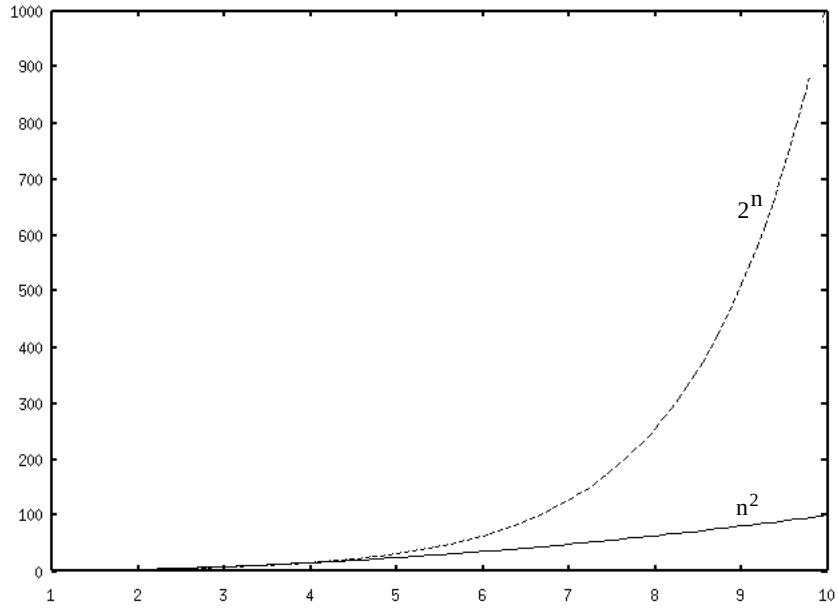


Figure 176: 2^n *vs.* n^2

A convenient way to approach analysis of some functions is through the *derivative*. Suppose we are trying to establish $f \in O(g)$. Even though we are working with functions on a natural number domain, suppose that each function has an analytic counterpart *F* and *G* on the real domain. If *G* maintains a greater derivative than *F* for sufficiently large n, then at some point the slope of the curve for *F* will stay less than the slope of the curve for *G*. By extrapolating from this point, we can see that *G* will ultimately overtake *F*. This is illustrated in the following diagram.

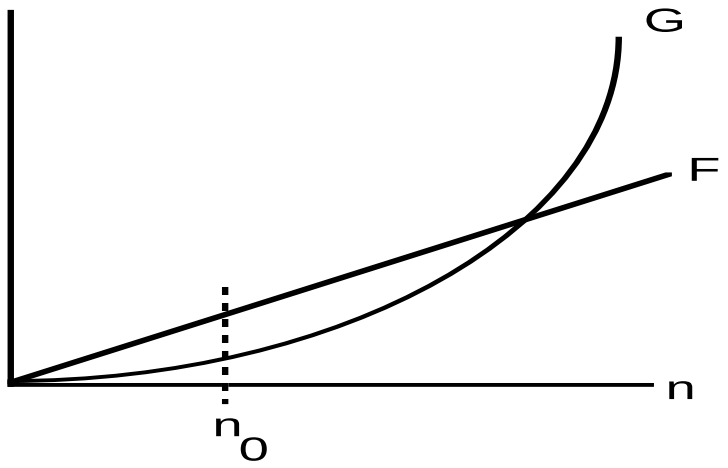


Figure 177: Showing that f ∈ O(g) through knowledge of the derivatives of corresponding analytic functions. At point n_0 the derivative of G becomes greater than that of F.

11.13 Derivative Rule

A sufficient condition for $f \in O(g)$, where *f* and *g* are restrictions of analytic functions *F* and *G* to the natural number domain, is that

$$(\exists\ n_0)(\forall\ n > n_0)\ \ F'(n)\ \leq G'(n)$$

where F' and G' denote the first derivatives of F and G respectively.

Caution: The condition above is only sufficient for $f \in O(g)$. It is not necessary. For example, one can easily construct examples using functions where each function is O of the other, yet there are no corresponding analytic functions.

Example: log n ∈ O(n)

Here we are comparing two functions: *log*, and the identity function $n \rightarrow n$. Let us call the analytic counterparts *F* and *G*. Then from calculus the derivatives have the property that

F'(n) = c / n where c is an appropriate constant†

G'(n) = 1

Thus if we choose n_0 to be the next integer above c, we have the conditions set forth in the derivative rule: $(\forall\ n > n_0)\ \ c / n \leq 1$.

Example: log n ∈ $O(n^{1/2})$

$n^{1/2}$ is, of course, another way of writing the square root of n. From calculus, the derivative of this function is $1/(2n^{1/2})$. This derivative will overtake the derivative of log n, which is c / n. Equality occurs at the point where

$$c / n = 1/(2n^{1/2})$$

i.e. n = ceiling($4c^2$).

11.14 Order-of-Magnitude Comparisons

Below is a table of some common functions and their approximate values as n ranges over 6 orders of magnitude.

log n	3.3219	6.6438	9.9658	13.287	16.609	19.931
$\log^2 n$	10.361	44.140	99.317	176.54	275.85	397.24
sqrt n	3.162	10	31.622	100	316.22	1000
n	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
n log n	33.219	664.38	9965.8	132877	$1.66*10^6$	$1.99*10^7$
$n^{1.5}$	31.6	10^3	$31.6*10^4$	10^6	$31.6*10^7$	10^9
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	1024	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}
n!	3 628 800	$9.3*10^{157}$	10^{2567}	10^{35659}	10^{456573}	$10^{5565710}$

Values of various functions vs. values of argument n.

Such tables can give hints to the growth rate of functions, although are by no means to be considered a proof. For such things we should rely on analytic methods. In any case, such tables are instructive. For example, the table above shows that we can run a problem with a factor of 10^6 larger in its input measure using an O(log n) algorithm in only 20 times

† Recall that for any two bases, a and b, $\log_b x = \log_b a\ \log_a x$.

longer to execute. For an O(n log n) algorithm, we would require only $20*10^6$ times longer, as compared to 10^{12} times longer for an $O(n^2)$ algorithm, a factor of $5*10^4$.

An "inverted" version of the table can be used determine the relative sizes of problem that can be run in a *fixed* time using algorithms of various orders.

Time Multiple	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
log n	1024	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{30000}
$\log^2 n$	8	1024	$3*10^9$	$1.2*10^{30}$	$1.5*10^{95}$	$1.1*10^{301}$
sqrt n	100	10^4	10^6	10^8	10^{10}	10^{12}
n	10	10^2	10^3	10^4	10^5	10^6
n log n	4.5	22	140	1000	$7.7*10^3$	$6.2*10^4$
$n^{1.5}$	4	21	100	210	2100	10000
n^2	3	10	32	100	320	1000
n^3	2	4	10	21	46	100
2^n	3	6	9	13	16	19
n!	3	4	6	7	8	9

Increase in size of problem that can be run based on increase in allowed time, assuming algorithm runs problem size 1 in time 1.

This table tells us, for example, that if we have 1000 times more time, if our algorithm is O(n!) we can only run a problem 6 times as large. On the other hand, if we have an n log n algorithm, we could run a problem 140 times as large in the same time.

11.15 Doubling Comparisons

Perhaps a handier way to remember how various functions grow is to consider what happens if we double the input size. If the function is O(n), then doubling the input size will at most double the execution time. If the function is log(n), then doubling the input size will only add a constant to the execution time, and so on. We can summarize these sorts of observations in the following table, where k is a constant.

Complexity	**Doubling the input causes execution time to**
O(1)	stay the same
O(log n)	increase by an additive constant
$O(n^{1/2})$	increase by a factor of sqrt(2)
O(n)	double
O(n log n)	double, plus increase by a constant factor times n
$O(n^2)$	increase by a factor of 4
$O(n^k)$	increase by a factor of 2^k
$O(k^n)$	square

This table can be used to help intuition in algorithm design. For example, if the algorithm is observed to double in time plus add a constant factor times the input, then we can infer that the algorithm is O(n log n). An example of this kind of behavior is quicksort, under ideal circumstances.

11.16 Estimating Complexity Empirically

Given access to the code of a program, the complexity can often be determined by analysis. However, it may be desirable to check our analysis empirically, or we might wish to estimate the complexity of a program the code of which we do not have. A way to proceed experimentally is to run the program on inputs of a variety of values of the input measure and record the time in each case. This will give us a set of size-time pairs (S_i, T_i). A good choice would be to have input sizes differ by successive powers of two. Of course we do not know that our chosen inputs achieve the maximum time among inputs of that size; we are just assuming that all will be about the same, an assumption that is not always valid. If this assumption is in question, we can run the program with multiple inputs of each size.

The size-time pairs derived above constitute an approximation to the time complexity function T of the program. Now we form a hypothesis that T is O(*f*) for some function *f*. For example, if we were analyzing a sorting program, we might hypothesize that T is $O(n^2)$. If our hypothesis is correct, then there must be a constant c such that

$$\text{(for all S)} \qquad T(S) \leq cf(S).$$

We can compute the ratios T(S)/f(S). If there is a noticeable upper bound on this ratio, then that would make a possible value of c. If the ratios seem to hover around c, especially as S gets larger, then there is a good chance that our hypothesis is correct. On the other hand, if the ratios tend to decrease with S, our hypothesis is probably correct, but it might have been too conservative. That is, there might be a tighter bound, such as *n log n* in the case of the sorting program. The third possibility is that there is no bound evident. In this case, our hypothesis is probably incorrect and we should try again with a new hypothesis of a faster growth rate.

Empirical Comparison of Sorting Procedures

The various sorts described were tested on our local computer with varying sizes of arrays, doubling the sizes from 16 through 4096. The algorithms themselves will be described in a later section. As we timed each run, we also computed the time divided by n, n^2, and n log n, in an effort to ascertain the asymptotic performance and compare it to our analytic results. The results are shown below. The reader should examine each table and conclude:

(a) which bound best describes the performance of that particular algorithm

(b) an appropriate multiplicative constant for each bound

(c) an estimate of the time each algorithm would take for one million elements

The reader should also try to get a sense of how a good asymptotic bound does not necessarily indicate the best algorithm for small data sizes.

minsort

elements	time (sec)	time/n	time/(n*n)	time/(n*log n)
16	0.00085938	0.00005371	0.00000336	0.00001937
32	0.00304688	0.00009521	0.00000298	0.00002747
64	0.01109375	0.00017334	0.00000271	0.00004168
128	0.03687500	0.00028809	0.00000225	0.00005937
256	0.14187500	0.00055420	0.00000216	0.00009994
512	0.56750000	0.00110840	0.00000216	0.00017768
1024	2.24750000	0.00219482	0.00000214	0.00031665
2048	8.98000000	0.00438477	0.00000214	0.00057508
4096	35.89000000	0.00876221	0.00000214	0.00105343

insertSort

elements	time (sec)	time/n	time/(n*n)	time/(n*log n)
16	0.00058594	0.00003662	0.00000229	0.00001321
32	0.00210938	0.00006592	0.00000206	0.00001902
64	0.00781250	0.00012207	0.00000191	0.00002935
128	0.03000000	0.00023437	0.00000183	0.00004830
256	0.11875000	0.00046387	0.00000181	0.00008365
512	0.47125000	0.00092041	0.00000180	0.00014754
1024	1.88000000	0.00183594	0.00000179	0.00026487
2048	7.50500000	0.00366455	0.00000179	0.00048062
4096	34.86000000	0.00851074	0.00000208	0.00102320

quicksort

elements	time (sec)	time/n	time/(n*n)	time/(n*log n)
16	0.00082031	0.00005127	0.00000320	0.00001849
32	0.00171875	0.00005371	0.00000168	0.00001550
64	0.00328125	0.00005127	0.00000080	0.00001233
128	0.00906250	0.00007080	0.00000055	0.00001459
256	0.01625000	0.00006348	0.00000025	0.00001145
512	0.03500000	0.00006836	0.00000013	0.00001096
1024	0.07500000	0.00007324	0.00000007	0.00001057
2048	0.17000000	0.00008301	0.00000004	0.00001089
4096	0.35000000	0.00008545	0.00000002	0.00001027

```
heapsort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00144531   0.00009033   0.00000565   0.00003258
      32   0.00242187   0.00007568   0.00000237   0.00002184
      64   0.00640625   0.00010010   0.00000156   0.00002407
     128   0.01218750   0.00009521   0.00000074   0.00001962
     256   0.02875000   0.00011230   0.00000044   0.00002025
     512   0.06375000   0.00012451   0.00000024   0.00001996
    1024   0.14250000   0.00013916   0.00000014   0.00002008
    2048   0.31000000   0.00015137   0.00000007   0.00001985
    4096   0.68000000   0.00016602   0.00000004   0.00001996

radixSort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00335937   0.00020996   0.00001312   0.00007573
      32   0.00546875   0.00017090   0.00000534   0.00004931
      64   0.01093750   0.00017090   0.00000267   0.00004109
     128   0.02187500   0.00017090   0.00000134   0.00003522
     256   0.04312500   0.00016846   0.00000066   0.00003038
     512   0.08500000   0.00016602   0.00000032   0.00002661
    1024   0.18000000   0.00017578   0.00000017   0.00002536
    2048   0.35000000   0.00017090   0.00000008   0.00002241
    4096   0.69000000   0.00016846   0.00000004   0.00002025
```

Use of Limits

Sometimes a quick way to check whether $f \in O(g)$, $g \in O(f)$, etc. is to look at the limit of f(n)/g(n) as n increases without bound. If this limit exists (i.e. is finite), then $f \in O(g)$. This follows from the definition of "limit":

$$\lim_{n \to \infty} h(n) = c$$

means that

$$(\forall\, \varepsilon > 0)(\exists\, n_0)\ (\forall\, n > n_0)\ \ |\, h(n) - c\,| < \varepsilon$$

If this limit exists, where the h(n) of interest is f(n)/g(n), then

$$(\forall\, n > n_0)\ |\, f(n)/g(n) - c\,| < \varepsilon$$

Thus (since functions of interest to us are positive)

$$(\forall\, n > n_0)\ f(n) < (c + \varepsilon)g(n)$$

By letting d be the maximum of the values of f(n) for $n \leq n_0$, we get

$$(\forall\, n)\ f(n) < \max(d, (c + \varepsilon))g(n)$$

By choosing the constant in the definition of O to be max(d, (c + ε)), we have shown

Limit Rule

If a finite limit c of the ratio of f(n)/g(n) exists

$$\lim_{n \to \infty} f(n)/g(n) = c$$

then $f \in O(g)$.

L'Hopital's Rule

Sometimes the limit may exist but is not easy to derive. This happens when the ratio f(n)/g(n) cannot be simplified in an obvious way. An example is log(n) / sqrt(n). Both the numerator and denominator go to ∞ as n increases. In such cases, the notion of derivative can again be used:

$$\lim_{n \to \infty} f(n)/g(n) = \lim_{n \to \infty} f'(n)/g'(n)$$

where f' and g' denote the first derivatives of f and g, provided that

$$\lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty$$

We can continue applying this rule iteratively until a reducible form is obtained, since

$$\lim_{n \to \infty} f'(n)/g'(n) = \lim_{n \to \infty} f''(n)/g''(n)$$

Additional Complexity Notation

Here is some additional notation that is used in the literature (see the references by Knuth and by Brassard).

f ∈ Ω(g) is used to designate that $g \in O(f)$. That is, *g* is a *lower bound* on *f*, within the confines of some multiplicative constant.

f ∈ Θ(g) is used to abbreviate that $f \in O(g)$ *and* $g \in O(f)$, i.e. the two functions have the *same* growth-rate.

f ∈ o(g) [*f* is "little-oh" of *g*] means that $f \in O(g)$ and that the limit of f(n)/g(n) is 0. In other words, f(n) becomes insignificant compared to g(n) as n gets large.

In particular, use of Θ notation indicates that the given bound is *tight*.

Exercises

1 •• Show that for any positive constant ε, $\log n \in O(n^{\varepsilon})$.

2 •• Show that $\sum_{i=1}^{n} i^2 \in O(n^3)$

3 ••• Derive a closed form expression for $\sum_{i=1}^{n} i^2$. Prove that your expression is correct by induction. (Hint: From the preceding problem, it might be reasonable to try a 3rd order polynomial. If there is such a polynomial, you could solve for its coefficients by constructing 4 equations with the coefficients as unknowns.)

4 ••• Show that for any fixed k, and any $c > 1$, $n^k \in O(c^n)$.

5 •• Which of the following are true, and why? $2^n \in O(2^{n+1})$. $2^{n+1} \in O(2^n)$.

6 • Suppose that a and b are positive integers with $a < b$. Which of the following are true, and why? $n^a \in O(n^b)$. $n^b \in O(n^a)$.

7 •• Suppose that a and b are positive constants with $1 < a < b$. Which of the following are true, and why? $a^n \in O(b^n)$. $b^n \in O(a^n)$.

8 •••• Suppose that f and g are functions such that $f(n) \in O(g(n))$. Let c be a positive constant. Is it necessarily true that $f(cn) \in O(g(n))$? Justify your answer.

9 ••• Let b be a positive integer. Show that

$(1 + b + b^2 + b^3 + \ldots + b^n) \in O(b^n)$.

[Hint: There is a closed form for the left-hand side: It is the sum of a geometric series.]

10 •• Show that for any positive integer k, $(\log n)^k \in O(n)$.

11 •• Prove the multiplication rule: If $f \in O(g)$, then $n \rightarrow n*f(n) \in O(n \rightarrow n*g(n))$.

12 ••• As mentioned earlier, our definition of $f \in O(g)$ applies to the restricted case that the domains are natural numbers. For the more general case of real domains, the definition needs to be modified. In fact, the standard definition of $f \in O(g)$ is

$$(\exists\, c)(\exists\, n_0)(\forall\, n > n_0)\ \ f(n) \leq cg(n)$$

That is, there exist constants c and n_0 such that for all n beyond n_0, $f(n) \leq cg(n)$. Show that on the domain of natural numbers, this definition is equivalent to the definition given in these notes. [Hint: For a given n_0, the set of natural numbers less than n_0 is finite. Thus one can use the maximum of the values of f(n) over this set in the construction of the constant c.]

13 •••• Using the general definition in the previous exercise, re-prove each of the rules for O that have been derived in the notes.

14 ••• For each of the following program outlines, provide the best "O" upper bound on the time complexity growth rate as a function of parameter N, where P(); is some constant-time computation. In all cases i and j are declared as `int`. Unless otherwise stated, assume single arithmetic operations are O(1).

a.
```
for( i = N; i > 0; i-- )
      P();
```

b.
```
for( i = N; i > 0; i-- )
      for(j = 0; j < i; j++ )
            P();
```

c.
```
for( i = N; i > 0; i-- )
      for( j = i; j < i+1000000; j++ )
            P();
```

d.
```
for( i = N; i > 0; i-- )
      for( j = i; j > 1; j = j/2 )
            P();
```

e.
```
for( i = N; i > 0; i = i/2 )
      for( j = i; j > 1; j = j/2 )
            P();
```

f.
```
for( i = N; i > 0; i /= 2 )
  for( j = i; j > 0; j-- )
      P();
```

g.
```
int i, j;
for( j = N; j > 0; j-- )
  for( i = j; i > 0; i /= 2 )
      P();
```

h.
```
double F = 1;
```

```
for( i = N; i > 0; i-- )
      F *= N;
```

15 ••• Rank the following functions in such a way that f precedes g in the ranking whenever $f(n) \in O(g(n))$. Show your rationale.

$a(n) = n^{1.5}$

$b(n) = n * \log n$

$c(n) = n/ \log n + n^3$

$d(n) = (\log n)^2 + \log(\log n)$

$e(n) = n + 10^9$

$f(n) = n!$

$g(n) = 2^n$

11.17 Case Studies Using Sorting

As already mentioned, the problem of arranging the elements of a sequence into increasing order is called *sorting*. Because the problem is easy to understand and there is a large number of methods, sorting provides a good set of examples for analyzing complexity. Earlier we saw that sorting by repeatedly selecting the minimum element from an array gives an upper bound of $O(n^2)$ for an n-element array. Here we look at other methods for sorting with hopes that good algorithm design can improve this bound. This is important when n gets large. For a sequence of size 10^6, if the complexity of sorting were n^2 microseconds, it would take 10^6 seconds to sort the sequence, i.e. it would take over 11.5 days. If we had a supercomputer that ran 100 times this fast, then the time still might be prohibitive, over 2.7 hours. If we were able to improve the algorithm to n log n microseconds, then a sequence of the same size would take less than 12 milliseconds, or less than .12 milliseconds on the supercomputer.

Although they may be expressed using numbers as data elements, a typical sorting application actually sorts *structs*, data objects consisting of several components or *fields*. Typically only one or two fields comprise the values upon which records are compared. The aggregate of these fields is called the *key* of the record. For example, records representing people might have a combination of last name and first name as the key. If the keys consist only of integers in a relatively small range, the best algorithm is apt to be quite different than in a case where the key is a chemical formula of arbitrary size.

Bucket Sort

For sorting elements chosen from a small range of integers, a **bucket sort** is a good choice. Suppose the range is 0 to N-1 where N is fixed. Then we can sort by constructing an array of N items into which the data are placed. The elements of this array are thought of as *buckets.* The general principle here is called **distribution sorting**, since we distribute the items into buckets. If the data items are just numbers, then since all numbers are alike, the buckets can just be counters of the number of times each number occurs. If the data items are general records, then the buckets would need to be containers for a sufficient number of records. A linked list of records would be a good candidate implementation.

The advantage of bucket sort is that it runs in O(n) time, because we need only make one pass through the input data to put all of the data in buckets, then a pass over the buckets to create the sorted data. This analysis assumes that most buckets are non-empty. If there is a much larger number of buckets than records and many of the buckets are empty, then the time to scan the buckets could dominate. Our O(n) figure assumes that almost all buckets have something in them.

Radix Sort

Radix sort is a variant of bucket sorting which uses fewer buckets by exploiting radix representations of the integer keys. The reason that the number of buckets is of concern is that if there are many more buckets than items to be sorted, and many buckets end up empty, handling the number of buckets could dominate the sorting time.

If the range of numbers is very large, we can conduct the distribution sort recursively, by dividing up the range into sub-ranges, performing an initial distribution, then sorting within the buckets. A variation on this idea is to use the **radix principle**. It is applicable when the values are represented as integer numerals in some base (radix). We sort on each digit of the numerals, starting with the least-significant. If the radix is b, then there are b buckets. We repeat this process, progressing toward the most-significant digit. After each distribution, we regroup the items anew, taking care to preserve their order from the previous distribution. After the last regrouping, the items are sorted.

The radix sorting principle was used on automatic punch-card sorters, and can also be used in hand punch-card sorting. The following illustrates how a home-made indexing system amenable to radix sorting can be constructed using cards. Obtain a deck of cards. Estimate n, the maximum range of values to be used in sorting. Punch log n holes along a specific edge of each of the cards, as suggested by the following diagram. For a card numbered m, cut a channel from the hole to the edge of the card for each hole corresponding to a 1 bit in the binary representation of m.

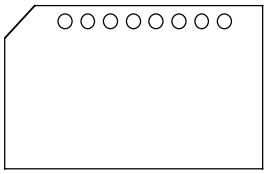


Figure 178: The card for number 0 in the indexing system

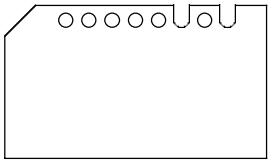


Figure 179: The card for number 5 in the indexing system

To sort the cards, insert a spindle into the holes representing the lowest-order bit. Lift the spindle, separating the cards without channels at that bit to those with channels. Restack the cards with the non-channel cards in front. Repeat this process on the next significant bit, and so on, up to the most significant bit. At the conclusion, the cards will be sorted.

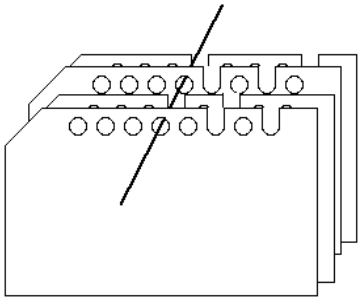


Figure 180: Using a spindle to select the cards having their fifth bit 0, a step in the radix sorting process

The following Java code simulates the sorting process outlined above.

```
class radixSort
{
/**
  *  Calling radixSort constructor on array of floats sorts the array.
  *  Parameter N is the number of elements to be sorted.
  */

// radixSort works using binary representation of numbers being sorted.
// radixSort first sorts on the least-significant bit, then the next least,
// and so on until there are no more bits which have 1 as a value.
// On each pass, it counts the number of words with a 0 in the current
// bit position. It then copies the elements from the array into a
// buffer so that all words with a 0 precede all with a one. It then
// copies the buffer back to the array for the next pass.
```

```
radixSort(int array[], int N)
  {
  int buffer[] = new int[N];    // place to put result of one pass

  boolean done = false;         // indicates whether sorting completed

  for( int shiftAmount = 0; !done; shiftAmount++ )
    {
    // one pass consists of the following:

    int count = 0;              // count of number of 0 bits

    done = true;

    // first phase: determine number of words with 0 bit

    for( int i = 0; i < N; i++ )
      {
      int shifted = (array[i] >> shiftAmount);  // move bit to low-order

      if( shifted > 0 )                         // is anything left?
        done = false;

      if( shifted % 2 == 0 )
        count++;                                // count this 0
      }

    if( done )
      break;


    // second phase: redistribute words with 0 vs. 1

    int lower = 0, upper = count;       // positions for redistribution

    for( int i = 0; i < N; i++ )
      {
      int shifted = (array[i] >> shiftAmount);

      if( shifted % 2 == 0 )
        {
        buffer[lower++] = array[i];
        }
      else
        {
        buffer[upper++] = array[i];
        }
      }

    for( int i = 0; i < N; i++ )
      {
      array[i] = buffer[i];
      }
    }
  }
```

The time to sort a set of numerals using radix sort is proportional to the number of numerals times the number of digits in the longest numeral. If the latter number is bounded by a constant K, then n numerals can be sorted in time proportional to Kn, i.e.

O(n) time. Again, this sort works only in the case that data can be represented as numerals in some radix.

We now turn to sorting methods that work on general keys, using only the assumption that two keys can be compared, but nothing further. A number of obvious sorting algorithms repeat the bound of $O(n^2)$ given by **minsort** discussed earlier. These include:

Simple insertion sort:

Repeat the following process: Begin with a prefix of the array containing just one element as a sorted array. From the remaining elements, choose the next one and find its position in the sorted array. Insert the element by moving the higher elements upward one. The algorithm is expressed in Java is shown below. As before, calling the constructor is what causes the array to be sorted, in place.

```
class insertSort
  {
  private double array[];     // The array being sorted
  int N;                      // The length of the prefix to be sorted

  // Calling insertSort constructor on array of doubles sorts it.
  // Parameter N is the number of elements to be sorted (which might
  // be fewer than are in the array itself).

  insertSort(double array[], int N)
    {
    this.array = array;
    this.N = N;

    for( int i = 1; i < N; i++ )
      {
      insert(i, findPosition(i));
      }
    }


   //  insert(i, j) inserts array[i] into an array at position j,
   //  shifting to the right the elements
   //  array[j+1], array[j+2], ...., array[i-1]

  void insert(int i, int j)
    {
    double hold = array[i];
    for( int k = i; k > j; k-- )
      {
      array[k] = array[k-1];
      }
    array[j] = hold;
    }


   //  findPosition(i) finds the position at which to insert array[i]
   //  in array[0] .... array[i-1]
   //
```

```
  int findPosition(int i)
    {
    double item = array[i];
    for( int k = i-1; k >= 0; k-- )
      {
      if( array[k] <= item )
        return k+1;
      }
    return 0;
    }
 }
```

That the above insertion sort is $O(n^2)$ can be seen by analyzing the programs using step counting. Intuitively, minsort and insertion sorts get their $O(n^2)$ linear nature of their attack on the problem: we have an outer loop that runs n steps, and the cost of that loop ranges from 1 to n. If we are to break through $O(n^2)$ to a lower upper bound, we must find an approach that is not so linear. Here is where the following principle suggests itself:

> **Divide-and-Conquer Principle**: Try to break the problem in half, rather than paring off one element at a time.

Perhaps the most obvious divide-and-conquer sorting algorithm is **Quicksort**. At least, it is obvious that the approach is correct. Quicksort is easiest to state recursively:

Quicksort: Sorting by Divide-and-Conquer

Basis: If the sequence consists of at most 1 element, it is sorted.

Recursion:

Break a sequence of more than 1 element into two, as follows:

Chose an element from the sequence as a pivot value. All elements less than the pivot value are selected as subsequence L and all elements greater than or equal the pivot value are selected as subsequence R.

Sort the subsequences L and R (recursively). Then form the sequence consisting of L (sorted) followed by the pivot, followed by R (sorted).

In Java this could be expressed as follows:

```
class quicksort
  {
  float a[];

   // Calling quicksort constructor on array of floats sorts the array.
   // Parameter N is the number of elements to be sorted.
```

```
quicksort(float array[], int N)
  {
  a = array;
  divideAndConquer(0, N-1);
  }

// sort the segment of the array between low and high

void divideAndConquer(int low, int high)
  {
  if( low >= high )
    return;                          // basis case, <= 1 element

  float pivot = a[(low + high)/2];    // select pivot

  // shift elements <= pivot to bottom
  // shift elements >= pivot to top

  int i = low-1;
  int j = high+1;

  for( ; ; )
    {                                // find two elements to exchange
    do { i++; } while( a[i] < pivot );  // slide i up
    do { j--; } while( a[j] > pivot );  // slide j down

    if( i >= j ) break;              // break if sliders meet or cross

    swap(i, j);                      // swap elements and continue
    }

  divideAndConquer(low, i-1);        // sort lower half
  divideAndConquer(j+1, high);       // sort upper half
  }


 // swap(i, j) interchanges the values in a[i] and a[j]

void swap(int i, int j)
  {
  float temp = a[i];
  a[i] = a[j];
  a[j] = temp;
  }
```

Under ideal circumstances, the dividing phase of quicksort will split the elements into two equal-length subsequences. In this case, there will be *log* n levels of recursive calls to quicksort. At each level, O(n) steps must be done to split the array. So the running time is on the order of n *log* n. Unfortunately this is only in ideal circumstances, although by a probabilistic argument that we do not present, it also represents an *average* case performance under reasonable assumptions. The worst case performance however causes the array to split very unevenly, resulting in a worst case of $O(n^2)$, which is the worst case for the other sorting algorithms presented. In a worst-case sense, we have made no progress.

Heapsort: Using a Tree to Sort

Now that we have a hint from Quicksort that O(n log n) *might* be achievable in the best case, we seek an algorithm that has this performance. Whenever we are trying to make improvements over algorithms that deal with linear sequences, the following approach is worth trying:

> **Tree Structuring Principle:**
>
> Rather than dealing with the sequence linearly, try to employ a tree structure to cut sequence traversal needs from n to *log* n.

How about doing insertions within a tree rather than in a linear array, as is done by the simple insertion sort? If there are n elements and we can do each insertion in O(log n) time, we might be able to achieve our goal. Of course there are details to be worked out concerning how we can do the insertions so as to maintain the balance of the tree.

A linear array, as used in *select_min* sort and simple insertion sort, maintains the items sorted thus far in a strict order. By relaxing this condition, we can keep the information "partially ordered" and gain a faster insertion. The original sort of this nature, as presented by Williams, 1964, was called heapsort, reflecting a "heap" structure, a particular type of tree structure.

Note: The heap in this section should not be confused with the *heap* used for general storage of dynamic data structures.

The Heap Invariant

An example of a heap is shown below. A heap has the following defining property (or invariant)

> **The children of any node cannot be greater than the node itself**.
>
> *heap invariant*

This means that there is a tendency for increasing values as we move toward the root. As a corollary, we see that the progeny of a node cannot be greater than the node itself, and further the root must be the greatest element in the tree.

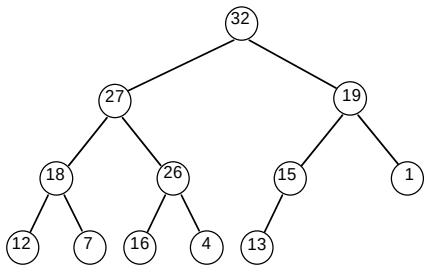


Figure 181: A heap

The following diagram shows the standard implementation of a heap using an array. The array is viewed as being indexed starting at 1. A node can have 0, 1, or 2 children. The children of a node with index p have indices 2*p and 2*p+1.

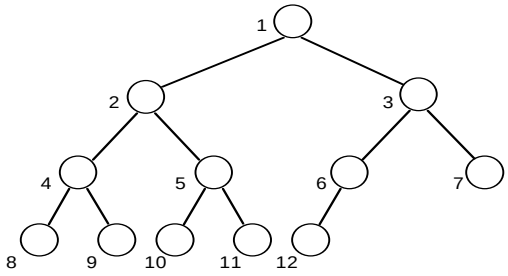


Figure 182: A heap mapped onto the locations of an array.

The numbers show indices of locations, not data values.

The reasoning for choosing 1-origin indexing is just so the relationship between parent and child indices is simple arithmetically. If we use 0-origin instead, it becomes slightly more complicated.

We can get a rough idea of how sorting is done using a heap by first explaining another structure that can be implemented as a heap: a **priority queue**. This term was introduced earlier. Recall that in a priority queue discipline, the *largest* item is always the next to be removed. Evidently, the largest item in a heap is always the root, so it is easy enough to locate. However, in removing the root, we must fill the vacancy thus left in such a way that the result is still a heap. Our goal will be to show that the heap can be formed in time O(n log n) and reformed, after removing an element, in time O(log n). Given this, the following code indicates how a heap can be used to achieve an O(n log n) sorting algorithm.

```
.... create an empty heap ....
for( i = 0; i < n; i++ )
      .... add a[i] to the heap ....

for( i = n-1; i >= 0; i-- )
      .... remove a[i] from the heap ....
```

The heapsort algorithm also uses a space-saving trick: using the vacated locations in the heap for storage of the final sorted array. This means that no additional memory space is needed.

Deleting the Maximum from a Heap

The ability to find the maximum quickly within a heap is based on the "partial ordering" of the node values. To preserve this property, it is not enough to simply remove the root.

> When removing the max from a heap, we must adjust the tree immediately afterward so that the max so that the heap invariant once again holds.
>
> *preservation of the heap invariant*

Below we show our original heap example after removing the max. Obviously there is a "hole" at the root that needs to be filled. Also obviously, the value that must fill this hole is 27, since it is the maximum of the remaining nodes.

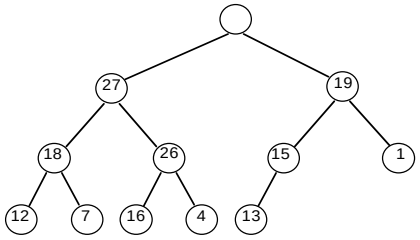


Figure 183: Original heap example after removing the maximum, but before restoring the heap invariant

However, if we move 27 to the root, that leaves another hole, etc. Continuing this process, we would eventually have a hole in the top row. This situation is undesirable, for it means that the heap can no longer be represented as a contiguous array.

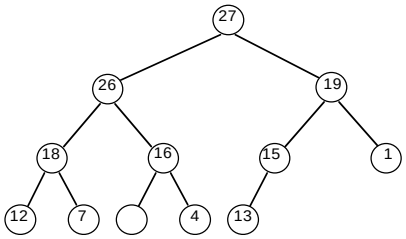
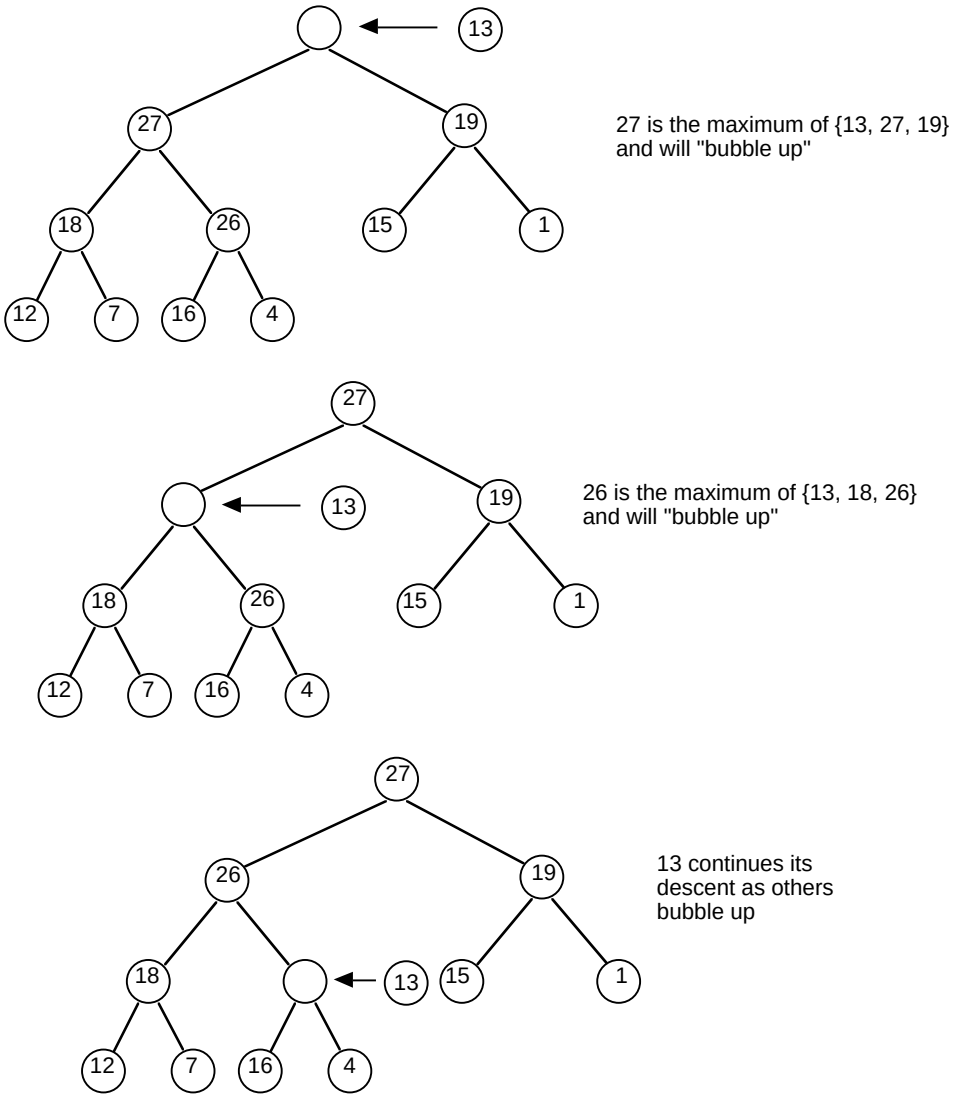


Figure 184: Undesirable situation: after filling the hole at the root with the maximum, and continuing this process down the tree, we have left a hole at the bottom level. This heap can no longer be represented as a contiguous array.

In order to avoid this situation, we shall have to plan in advance to preserve the contiguity of the heap. We can do this by removing that *last* item in the heap, i.e. the one that appears in the lower right node of the tree, 13 in this case. So we temporarily make an orphan out of 13, to find a new home somewhere in the array.

We fill the hole at the root with this former leaf value (13), then adjust the array by "bubbling up" that value. In this case, bubbling up means to repeatedly adjust a combination of three nodes so that the maximum is the parent, as shown in the following sequence.



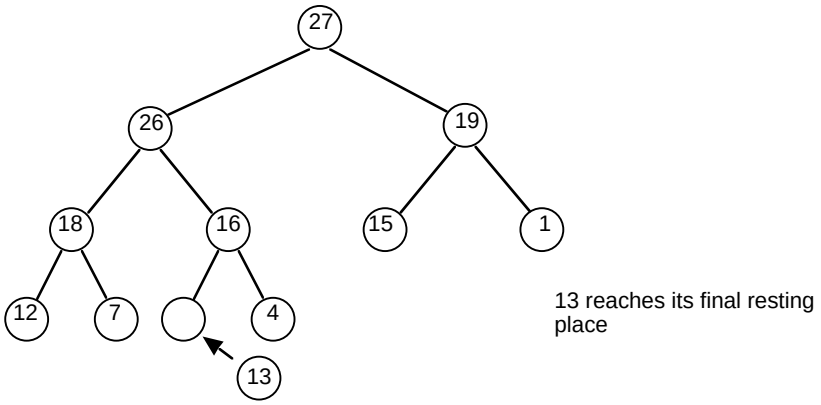


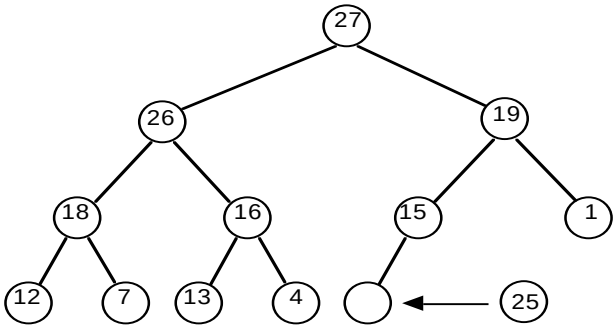
Figure 185: Restoring the heap invariant

At this point, the heap invariant is restored and the corresponding heap is again contiguous. The algorithm for delete_max then is a mechanization of this process. We notice that the algorithm for deleting max runs in O(*log* n), since all accesses to nodes are made along a single path from root to some leaf. We only look at nodes directly on that path and single nodes to one side or the other.

Initial Creation of a Heap

A heap can be created by starting with an empty heap and repeatedly adding new elements. As with removal, we must show how to maintain the heap invariant when inserting, as well as making sure that insertions are also O(*log* n).

Let us work with the heap above as an example, and suppose that the value 25 is to be added. As before, to maintain contiguity as a heap, we will need to put into play the node shown vacant below. We use a technique similar, but not identical, to the one used for removing the maximum: We put the new element 25 into the vacancy, then let nodes above it change places until an appropriate level is reached. This is simpler than deleting the maximum, because it only involves a 2-way comparison, rather than a 3-way one.



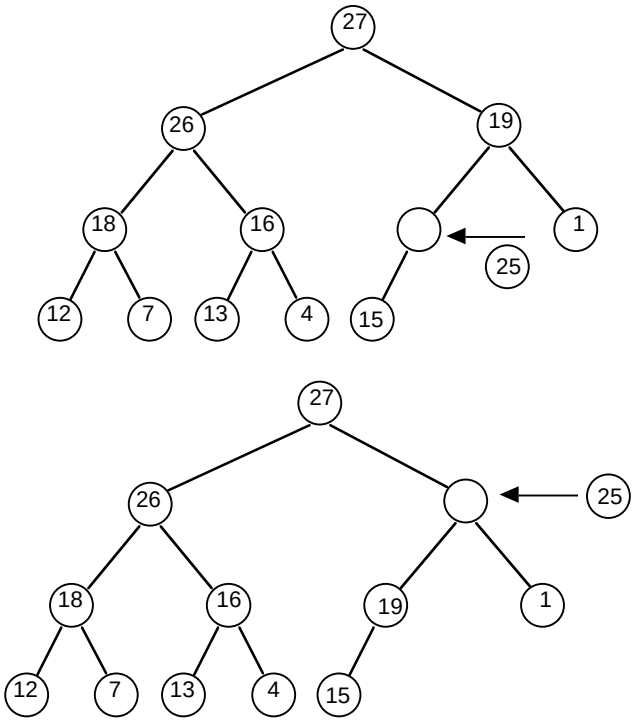


Figure 186: Bubbling up in the initial creation of a heap

At this point, the proper position for the new element has been found, so bubbling stops, leaving us with a new heap. Again we notice that the heap modification is accomplished in O(log n), since only nodes on the path from the created vacancy to the root are examined.

The following code shows the combination of these ideas in the heapsort procedure, which sorts an array in place:

```
class heapsort
  {
  private float array[];        // The array being sorted

  // Calling heapsort constructor on array of floats sorts the array.
  // Parameter N is the number of elements to be sorted.

  heapsort(float array[], int N)
    {
    this.array = array;
    int Last = N-1;

    // A heap is a tree in which each node is smaller than either of its
    // children (and thus than any of its descendants). All sub-trees
    // of a heap are also heaps. In this program, a heap is stored as
    // an array, with the root at element 0. In general, if a node is
    // at element I, its children are at elements 2*I+1 and 2*I+2.
```

```
  // phase 1: form heap
  // Construct heap bottom-up, starting with small trees just above
  // leaves and coalescing into larger trees near the root.

  for( int Top = Last/2; Top >= 0; Top-- )
    {
    adjust(Top, Last);
    }


  // phase 2: use heap to sort
  // Move top element (largest) out of heap, swapping with last
  // element and changing the heap boundary, until only one element
  // remains.

  while( Last > 0 )
    {
    swap(0, Last);
    adjust(0, --Last);
    }
  }


//  adjust(Top, Last) adjusts the tree between Top and Last

void adjust(int Top, int Last)
  {
  float TopVal = array[Top];                  // Set aside top of heap
  int Parent, Child;

  for( Parent = Top; ; Parent = Child )       // Iterate down tree
    {
    Child = 2*Parent+1;                       // Child is left child

    if( Child > Last )
      break;                                  // No left child exists

    if( Child+1 <= Last                       // Right child exists
         && array[Child] < array[Child+1] )   // and is larger
      Child++;                                // Child is larger child

    if( TopVal >= array[Child] )
      break;                                  // Location for TopVal

    array[Parent] = array[Child];             // Move larger child up
    }

  array[Parent] = TopVal;                     // Install TopVal
  }


 // swap(i, j) interchanges the values in array[i] and array[j]

void swap(int i, int j)
  {
  float temp = array[i];
  array[i] = array[j];
  array[j] = temp;
  }
}
```

Merge Sort

One natural method of sorting is to sort by repeated merging. A set of rules for this form of sort was given in the chapter *Low-Level Functional Programming*. Our analysis of merge_sort as given in that chapter follows:

a. The time to merge a pair of lists to get an *N* element list is O(N) since each recursive call "retires" an element to the result list and each element of the result gets retired only once.

b. Thus, the time to *merge_pairs* on a list of lists, the summed total length of which is N, is O(N), since from a. the time to merge a single pair is proportional to the number of elements in the result.

c. The time to use *repeat* on a list of *N* 1-element lists is the number of times *repeat* is called recursively times O(N), from b.

d. The number of times *repeat* is called is O(log N), since each call sees a list that is about half as long as the previous call.

e. The time to merge_sort an *N* element list is O(N) + time to repeat an *N* element list, since mapping an *N* element list is O(N). From c and d, the time to repeat is O(N log N).

Therefore the overall time to *merge_sort* an *N* element list is O(N log N).

11.18 Complexity of Set Operations and Mappings

A recurring theme in computer problem solving is the need to deal with various kinds of sets and mappings. Frequently we have need for sets of integers, strings, and more complex data types, as well as mappings from a wide variety of types to integers, etc. Thus it is worthwhile using these as one of the foci in our discussion of complexity. A wide variety of different representations for these abstractions is available and they differ in the types of data they handle, the complexity, space requirements, and coding difficulty. It is helpful to taxonomize the methods based on these points.

Thinking of a set as a class, the following operations are typical:

find Find out whether a member is in the set. If so, return a locator for it. A locator is a kind of reference to the member. It is used in deleting the member or changing it.

add Add a new member to the set (assuming it is not present already)

delete Given a locator for a member, remove the member from the set.

new Create a new set.

Set Implementation Using an Unordered Array

Perhaps the simplest way to represent a set is as an array, the elements of which are in no particular order. The implementation is similar to that for a stack using an array, with an index indicating the last element in the set. Adding to the set (assuming there is space available) is trivial, essentially the same as a push onto the stack. Finding a member requires a search through the array up until the point the element is found or the end is reached. In the worst case, all elements in the array are examined. Delete, given a locator, is simple: since order is not important, we can fill the hole left by deletion by putting the last element in its place (unless, of course, we wish to maintain the order of insertion, but once order becomes important, we no longer have a set).

Letting *N* represent the current set size, we can see from the above discussion that the following upper bounds hold on the set operations:

find	O(N)
add	O(1)
delete	O(N)
new	O(1)

Here we assume that the delete accounts only for the time to do the deletion, not to find the element being deleted as well. A similar discussion and set of bounds holds for using a **linked-list** to represent a set.

Binary Search Principle

Some improvement can be obtained, at the expense of slightly greater coding complexity, by keeping the members of the set *in order*. This requires that an order be available for the domain of the set, which is not always the case. The ordering assumption is not at odds with the definition of a set being unordered, since its use is for internal purposes.

By keeping the set elements in order, the technique of **binary search** becomes available for the find operation. Here is how binary search works for find: Using index computation to find the index of the mid-point of the array, we take a "stab" at that point and compare it with the element to be found. If we have equality, we return the index as the locator. If the element to be found is less than the mid-point, then we search in the half-array below the mid-point; if it is greater, we search in the half-array above. This process is repeated until the element is either found, or we are left with an empty array to search.

Each step of the binary search procedure reduces the number of elements still under consideration by half. Therefore, the number of steps is proportional to the number of halvings it takes to reduce the original number of elements, *N*, to less than 1. This number of steps is, of course, *log N*.

The superiority of binary search for find in an ordered array is tempered somewhat by the costs of addition and deletion. In order to maintain the ordering, we have to shift elements one way or the other when we insert or delete. In the worst case, we might have to shift all of the elements. The complexity for the ordered array case would thus appear as follows:

find	O(log N)
add	O(N)
delete	O(N)
new	O(1)

If most of the activity involving the set is of the *find* type, with few additions or deletions, then binary search on an ordered set is preferred over using an unordered set.

Binary Search Trees

Does using an ordered linked-list help in a similar way? If we have a linked structure, additions and deletions usually become less complex. Unfortunately a linked-list doesn't provide a good way to do the index computation required for binary search. That is, we'd like to have a way to find the mid-point of a list similar to what was used for an array. But without adding additional structure, there is no such way. The desire to have something approaching a linked structure on which binary search can be done has motivated the use of various tree structures, the most obvious of which is the **binary search tree**.

A binary search tree enables binary searching on a linked structure by approximating access to successive mid-points, similar to the way that binary search was described. To do this, each node in the structure has two links, one to the elements below the current one and one to the elements above. The figure shows a binary search tree that holds the set {1, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15}

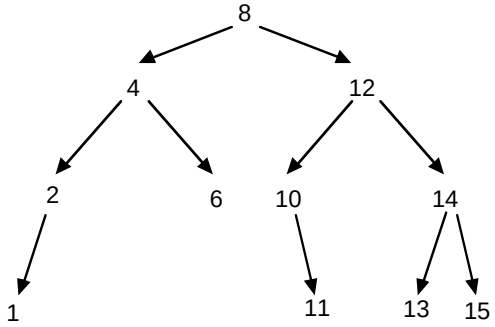


Figure 187: A binary search tree

The defining property of a binary search tree is that all elements in the left sub-tree of a node are less than the node itself, while all elements in the right sub-tree are greater than or equal to the node. If the binary search tree is ***balanced***, meaning that for a tree with N nodes, the length of the longest path is at most 1 + log N, then the search can be done in time O(*log N*). We can also insert and delete in time O(*log N*). However, it is not obvious that we can insert and delete in this amount of time and still maintain the balance needed for O(*log N*) search. In fact, several extensions of binary search trees have been developed that maintain the balance. These include AVL-trees, 2-3 trees, 2-3-4 trees, red-black trees, and B-trees. The reader might explore these interesting possibilities in future courses.

Bit Vectors

If the domain of a set's elements consists of integers, or can be easily mapped into integers (for example, character strings can be considered as large-radix numerals and are thus identifiable as integers), then a very fast method of set representation is available. A bit vector is an array with one array element per domain element. The value of an element is either 1, indicating that the corresponding domain element is a member of the set being represented, or 0 indicating that it is not. For example, if the domain is {0, ..., 15}, then a bit vector representation of the set {1, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15} is

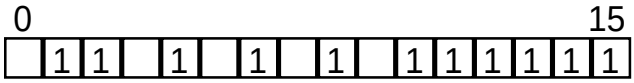


Figure 188: A bit vector representation of the set

{1, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15}
with empty entries representing 0

The advantage of a bit vector is that *find* is extremely fast. We simply index the position corresponding to the element we are trying to find. If the value is 1, the element is in the set, otherwise it is not. By the linear addressing principle, *find* is O(1) time. Similarly, add and delete are also O(1) time. In a bit vector, the actual elements are not stored, so the representation can be compact in terms of space. However this compactness only occurs if the elements in the set are relatively **dense** in the range of possible elements. The amount of space required by a bit vector is O(M) where M is the domain size. If the set is sparse and M is large, much space will be wasted. Also, it requires time proportional to M to *initialize* a bit vector, compared to time proportional to the size of the set for the other representations studied so far. To summarize, **for a bit vector**:

find	O(1)
add	O(1)
delete	O(1)
new	O(M)

Analysis of Hashing

Hashing was introduced in the chapter *Implementing Information Structures*. Under nominal conditions, the time for *find* using hashing is best regarded as O(1). This assumes that

- the elements of the set are distributed relatively evenly into buckets,
- the size of any one bucket is bounded by a constant, and
- the time to compute the hashing function is bounded by a constant

Any of these assumptions can be violated, but there are ways to remedy any tendency to violate them. For example, the first assumption can be satisfied by devising an appropriate hashing function, such as *hash_pdg* presented earlier. The size of buckets are bounded, assuming the first assumption holds, by making the table large enough. If the number of elements in the set is not known in advance, there are ways to extend the table size dynamically. This is not a trivial problem, but it is a solvable one, using techniques such as "extendible hashing" (see Fagin, *et al*. 1979). The third assumption is true if there is a bounded number of digits in the representation of each element. Obviously this would not be true if we used arbitrarily-long strings, but it is approximately true for many common cases. Since we have to look at every character of the string to be matched anyway, and the time to compute a typical hashing function is bounded by a constant times the length of the string, this time can be considered to be part of the cost of looking at the elements to be found.

The Trie Principle

Trie representation is a form of tree demonstrated in *Information Structures*. Unlike binary search trees, but similar to radix sorting, tries can exploit situations where the data can be represented as numerals. Since the linear addressing principle applies at each level of a trie, the access time to any element of a trie is O(1) if the number of levels is bounded, or O(L) where L is the number of levels, in general.

Sets vs. Bags and Mappings

So far we have emphasized structures for storing sets. However, most of these structures can also be adapted to implement bags and mappings as well. Usually it is a matter of storing additional information with the element inside the representation. For example, with bags we store a number indicating the *multiplicity* of the element in the bag; the absence of an element implies multiplicity 0. With mappings, the elements stored in the set are the domain values, and with each element in the domain we store the corresponding range value.

Exercises

1 ••• Rewrite the radix-sort procedure using a queue abstract data type. In particular, store the initial data on one queue and dequeue each item, placing it in one of two other queues depending on the least significant bit. Repeat this process for bits of successively-increasing significance. (Note: You do not have to implement the queue data type; that was discussed in Computing Objectively.) Hopefully your algorithm is easier to understand now that more abstraction has been employed.

2 ••• Try to devise an O(n) sort for numbers of fixed precision using a trie.

3 ••• Design a data abstraction for *"bignums"*, integers of arbitrary precision, using internal arrays of fixed-precision items (such as `short int`). Implement the operations of addition, subtraction, and multiplication at a minimum. Give O bounds on the complexity of your operations as a function of argument size.

4 ••• Using your implementation in the previous problem, code the Russian peasants' method of raising a bignum to a power. Analyze the complexity of raising a fixed constant to a power. See how well your analysis agrees with empirical observation.

5 •••• Explore the possibility of speeding up bignum multiplication using the divide-and-conquer strategy.

6 ••• Empirically compare the performance of a spell checker using hashing against ones using (a) binary search, and (b) a trie. Assume that you do not count the time taken to create the ordered array or the trie.

7 ••• Conduct a literature search on methods for keeping binary search trees in balance, so as to ensure an O(log n) search time.

8 •• Write a program that will do a fast spelling check by using a dictionary stored as a hash table. Populate the table from a dictionary files, such as `/usr/dict/words` available in most UNIX® systems. Compare the speed of your program to one that searches the dictionary sequentially.

9 ••• Suppose you wish to treat arbitrary Polys as keys. Develop a hash function for this application. Use recursion and avoid converting the Poly into text first.

10 ••• Implement merge sort. Use linked lists. Verify empirically the *n log n* upper bound

11.19 Chapter Review

Define the following terms:

Amdahl's law
asymptotically dominated
binary search
binary search tree
bit vector
bucket sort
complexity
distribution sorting
divide and conquer
growth-rate
hashing
heapsort
insertion sort
L'Hopital's rule
merge sort
"O" notation
profiling
quicksort
radix sort
tight
trie
upper bound

Describe how you would estimate the complexity of a program empirically.

Describe the role of limits of sequences in interpreting complexity bounds.

11.20 Further Reading

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, 1983. [Moderate.]

Jon Bentley, *Programming Pearls*, Addison-Wesley, Reading Mass., 1986. [A series of articles on algorithms and programming. Easy to moderate.]

Jon Bentley, *More Programming Pearls*, Addison-Wesley, Reading Mass., 1988. [A continuation of Bentley, 1986. Easy to moderate.]

G. Brassard. *Crusade for a better notation*. ACM SIGACT News, **17**, 1, 60-64 (1985).

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990. [Moderate.]

E. Fredkin, *Trie memory*, Comm. ACM, **3**, 4, 490-500, 1960.

R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, *Extendible hashing – a fast access method for dynamic files*, ACM Trans. on Database Systems, **4**, 315-344, 1979. [Shows an interesting way to expand hashing tables by splitting buckets. Moderate to difficult.]

R.W. Floyd, *Algorithm 245: Treesort 3*, Comm. ACM, **7**, 12, 345, 1964 [Improvements on the original heapsort.]

C.A.R. Hoare, *Quicksort*, Computer Journal, 5, 1, 10-15, 1962.

D.E. Knuth. *Big omicron and big omega and big theta*. ACM SIGACT News, **8**, 2, 18-24 (1976).

D.E. Knuth, *The Art of Computer Programming*, 3 volumes, Addison-Wesley, Reading Mass., 1973, 1981. [Comprehensive reference on algorithms and analysis. Moderate to difficult.]

Williams, J.W.J., *Algorithm 232: Heapsort*, Comm. ACM, **7**, 6, 347-348, 1964 [Original paper on heapsort.]

12. Finite-State Machines

12.1 Introduction

This chapter introduces finite-state machines, a primitive, but useful computational model for both hardware and certain types of software. We also discuss regular expressions, the correspondence between non-deterministic and deterministic machines, and more on grammars. Finally, we describe typical hardware components that are essentially physical realizations of finite-state machines.

Finite-state machines provide a simple computational model with many applications. Recall the definition of a Turing machine: a finite-state controller with a movable read/write head on an unbounded storage tape. If we restrict the head to move in only one direction, we have the general case of a finite-state machine. The sequence of symbols being read can be thought to constitute the input, while the sequence of symbols being written could be thought to constitute the output. We can also derive output by looking at the internal state of the controller after the input has been read.

Finite-state machines, also called *finite-state automata* (singular: *automaton*) or just finite *automata* are much more restrictive in their capabilities than Turing machines. For example, we can show that it is not possible for a finite-state machine to determine whether the input consists of a prime number of symbols. Much simpler languages, such as the sequences of well-balanced parenthesis strings, also cannot be recognized by finite-state machines. Still there are the following applications:

- Simple forms of pattern matching (precisely the patterns definable by "regular expressions", as we shall see).

- Models for sequential logic circuits, of the kind on which every present-day computer and many device controllers is based.

- An intimate relationship with directed graphs having arcs labeled with symbols from the input alphabet.

Even though each of these models can be depicted in a different setting, they have a common mathematical basis. The following diagram shows the context of finite-state machines among other models we have studied or will study.

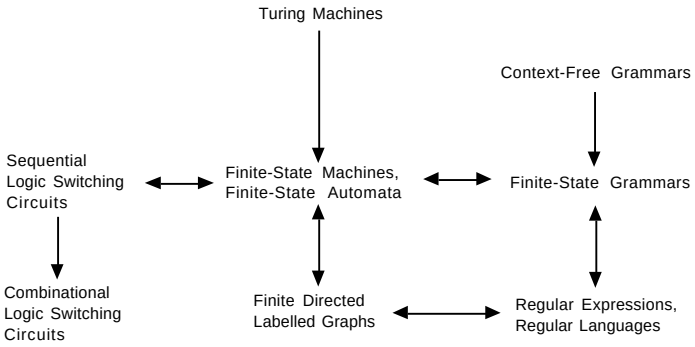


Figure 189: The interrelationship of various models with respect to computational or representational power. The arrows move in the direction of restricting power. The bi-directional arrows show equivalences.

Finite-State Machines as Restricted Turing Machines

One way to view the finite-state machine model as a more restrictive Turing machine is to separate the input and output halves of the tapes, as shown below. However, mathematically we don't need to rely on the tape metaphor; just viewing the input and output as sequences of events occurring in time would be adequate.

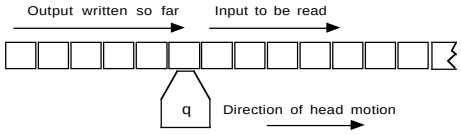


Figure 190: Finite-state machine as a one-way moving Turing machine

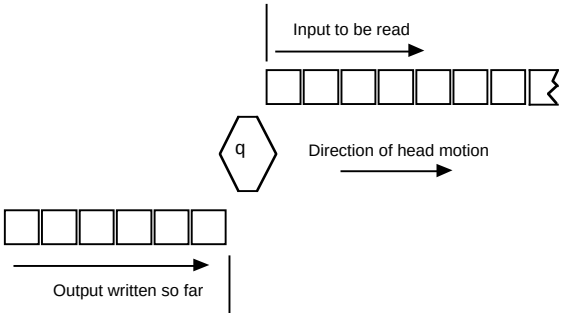


Figure 191: Finite-state machine as viewed with separate input and output

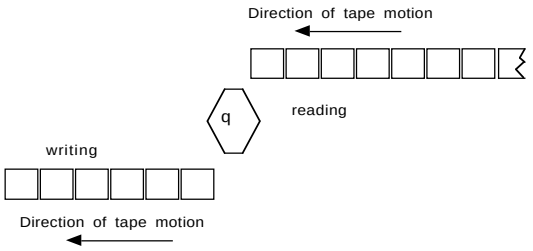


Figure 192: Finite-state machine viewed as a stationary-head, moving-tape, device

Since the motion of the head over the tape is strictly one-way, we can abstract away the idea of a tape and just refer to the input *sequence* read and the *output* sequence produced, as suggested in the next diagram. A machine of this form is called a ***transducer***, since it maps input sequences to output sequences. The term *Mealy machine*, after George H. Mealy (1965), is also often used for transducer.

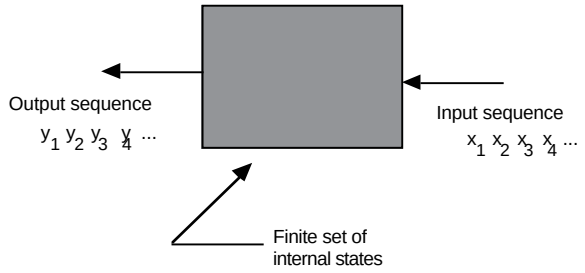


Figure 193: A transducer finite-state machine viewed as a tapeless "black box" processing an input sequence to produce an output sequence

On the other hand, occasionally we are not interested in the sequence of outputs produced, but just an output associated with the current state of the machine. This simpler model is called a *classifier*, or *Moore machine*, after E.F. Moore (1965).

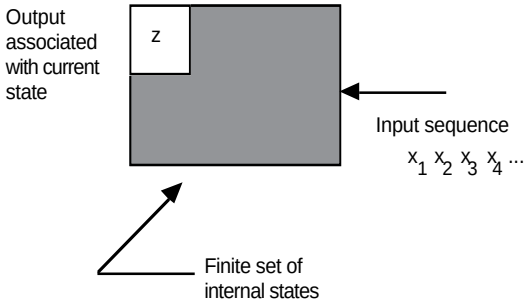


Figure 194: Classifier finite-state machine.
Output is a function of current state, rather than being a sequence

Modeling the Behavior of Finite-State Machines

Concentrating initially on transducers, there are several different notations we can use to capture the behavior of finite-state machines:

- As a functional program mapping one list into another.
- As a restricted imperative program, reading input a single character at a time and producing output a single character at a time.
- As a feedback system.
    - Representation of functions as a table
    - Representation of functions by a directed labeled graph

For concreteness, we shall use the sequence-to-sequence model of the machine, although the other models can be represented similarly. Let us give an example that we can use to show the different notations:

Example: An Edge-Detector

The function of an edge detector is to detect transitions between two symbols in the input sequence, say 0 and 1. It does this by outputting 0 as long as the most recent input symbol is the same as the previous one. However, when the most recent one differs from the previous one, it outputs a 1. By convention, the edge detector always outputs 0 after reading the very first symbol. Thus we have the following input output sequence pairs for the edge-detector, among an infinite number of possible pairs:

input	output
0	0
00	00
01	01
011	010
0111	0100
01110	01001
1	0
10	01
101	011
1010	0111
10100	01110
etc.	

Functional Program View of Finite-State Machines

In this view, the behavior of a machine is as a function from lists to lists.

Each state of the machine is identified with such a function.

The initial state is identified with the overall function of the machine.

The functions are interrelated by mutual recursion: when a function processes an input symbol, it calls another function to process the remaining input.

Each function:

looks at its input by one application of *first*,

produces an output by one application of *cons*, the first argument of which is determined purely by the input obtained from first, and

calls another function (or itself) on rest of the *input*.

We make the assumptions that:

The result of cons, in particular the first argument, becomes partially available even before its second argument is computed.

Each function will return NIL if the input list is NIL, and we do not show this explicitly.

Functional code example for the edge-detector:

We will use three functions, `f`, `g`, and `h`. The function `f` is the overall representation of the edge detector.

```
f([0 | Rest]) => [0 | g(Rest)];
f([1 | Rest]) => [0 | h(Rest)];
f([]) => [];

g([0 | Rest]) => [0 | g(Rest)];
g([1 | Rest]) => [1 | h(Rest)];
g([]) => [];

h([0 | Rest]) => [1 | g(Rest)];
h([1 | Rest]) => [0 | h(Rest)];
h([]) => [];
```

Notice that `f` is never called after its initial use. Its only purpose is to provide the proper output (namely `0`) for the first symbol in the input.

Example of `f` applied to a specific input:

f([0, 1, 1, 1, 0]) ==> [0, 1, 0, 0, 1]

An alternative representation is to use a single function, say `k`, with an extra argument, treated as just a symbol. This argument represents the *name* of the function that would have been called in the original representation. The top-level call to `k` will give the initial state as this argument:

```
k("f", [0 | Rest]) => [0 | k("g", Rest)];
k("f", [1 | Rest]) => [0 | k("h", Rest)];
k("f", []) => [];

k("g", [0 | Rest]) => [0 | k("g", Rest)];
k("g", [1 | Rest]) => [1 | k("h", Rest)];
k("g", []) => [];

k("h", [0 | Rest]) => [1 | k("g", Rest)];
k("h", [1 | Rest]) => [0 | k("h", Rest)];
k("h", []) => [];
```

The top level call with input sequence `x` is `k("f", x)` since `"f"` is the initial state.

Imperative Program View of Finite-State Machines

In this view, the input and output are viewed as streams of characters. The program repeats the processing cycle:

read character,
select next state,
write character,
go to next state

ad infinitum. The states can be represented as separate "functions", as in the functional view, or just as the value of one or more variables. However the allowable values must be restricted to a finite set. No stacks or other extendible structures can be used, and any arithmetic must be restricted to a finite range.

The following is a transliteration of the previous program to this view. The program is started by calling `f()`. Here we assume that `read()` is a method that returns the next character in the input stream and `write(c)` writes character `c` to the output stream.

```
void f()          // initial function
{
switch( read() )
  {
  case '0': write('0'); g(); break;
  case '1': write('0'); h(); break;
  }
```

```
}

void g()    // previous input was 0
{
switch( read() )
  {
  case '0': write('0'); g(); break;
  case '1': write('1'); h(); break; // 0 -> 1 transition
  }
}

void h()    // previous input was 1
{
switch( read() )
  {
  case '0': write('1'); g(); break; // 1 -> 0 transition
  case '1': write('0'); h(); break;
  }
}
```

[Note that this is a case where all calls can be "tail recursive", i.e. could be implemented as *goto*s by a smart compiler.]

The same task could be accomplished by eliminating the functions and using a single variable to record the current state, as shown in the following program. As before, we assume `read()` returns the next character in the input stream and `write(c)` writes character c to the output stream.

```
static final char f = 'f';                      // set of states
static final char g = 'g';
static final char h = 'h';

static final char initial_state = f;

main()
{
char current_state, next_state;
char c;

current_state = initial_state;
```

```
while( (c = read()) != EOF )
  {
  switch( current_state )
    {
    case f:                    // initial state
      switch( c )
      {
      case '0': write('0'); next_state = g; break;
      case '1': write('0'); next_state = h; break;
      }
      break;

    case g:                    // last input was 0
      switch( c )
      {
      case '0': write('0'); next_state = g; break;
      case '1': write('1'); next_state = h; break; // 0 -> 1
      }
      break;

    case h:                    // last input was 1
      switch( c )
      {
      case '0': write('1'); next_state = g; break; // 1 -> 0
      case '1': write('0'); next_state = h; break;
      }
      break;
    }
  current_state = next_state;
  }
}
```

Feedback System View of Finite-State Machines

The feedback system view abstracts the functionality of a machine into two functions, the next-state or state-transition function F, and the output function G.

F: States X Symbols → States state-transition function

G: States X Symbols → Symbols output function

The meaning of these functions is as follows:

F(q, σ) is the state to which the machine goes when currently in state q and σ is read

G(q, σ) is the output produced when the machine is currently in state q and σ is read

The relationship of these functions is expressible by the following diagram.

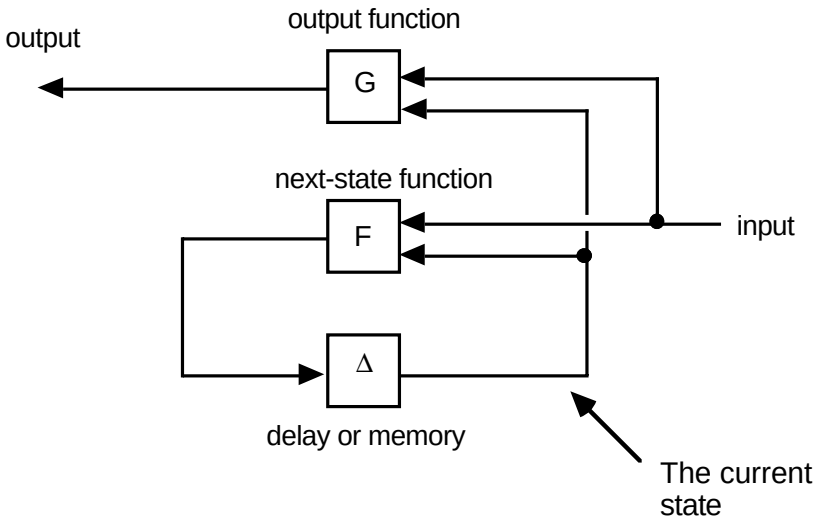


Figure 195: Feedback diagram of finite-state machine structure

From F and G, we can form two useful functions

F*: States X Symbols* → States extended state-transition function

G*: States X Symbols* → Symbols extended output function

where Symbols* denotes the set of all *sequences* of symbols. This is done by induction:

$F^*(q, \lambda) = q$

$F^*(q, x\sigma) = F(F^*(q, x), \sigma)$

$G^*(q, \lambda) = \lambda$

$G^*(q, x\sigma) = G^*(q, x)\, G(F^*(q, x), \sigma)$

In the last equation, juxtaposition is like cons'ing on the right. In other words, F*(q, x) is the state of the machine after all symbols in the sequence x have been processed, whereas G*(q, x) is the sequence of outputs produced along the way. In essence, G* can be regarded as the *function computed by* a transducer. These definitions could be transcribed into rex rules by representing the sequence xσ as a list [σ | x] with λ corresponding to [ ].

Tabular Description of Finite-State Machines

This description is similar to the one used for Turing machines, except that the motion is left unspecified, since it is implicitly one direction. In lieu of the two functions F and G, a

finite-state machine could be specified by a single function combining F and G of the form:

States X Symbols → States X Symbols

analogous to the case of a Turing machine, where we included the motion:

States X Symbols → Symbols X Motions X States

These functions can also be represented succinctly by a table of 4-tuples, similar to what we used for a Turing machine, and again called a *state transition table*:

$State_1, Symbol_1, State_2, Symbol_2$

Such a 4-tuple means the following:

> If the machine's control is in $State_1$ and reads $Symbol_1$, then machine will write $Symbol_2$ and the next state of the controller will be $State_2$.

The state-transition table for the edge-detector machine is:

	current state	**input symbol**	**next state**	**output symbol**
start state	f	0	g	0
	f	1	h	0
	g	0	g	0
	g	1	h	1
	h	0	g	1
	h	1	h	0

Unlike the case of Turing machines, there is **no particular halting convention**. Instead, the machine is always read to proceed from whatever current state it is in. This does not stop us from assigning our own particular meaning of a symbol to designate, for example, end-of-input.

Classifiers, Acceptors, Transducers, and Sequencers

In some problems we don't care about generating an entire sequence of output symbols as do the **transducers** discussed previously. Instead, we are only interested in categorizing each input sequence into one of a finite set of possibilities. Often these possibilities can be made to derive from the current state. So we attach the result of the computation to the state, rather than generate a sequence. In this model, we have an output function

c: Q → C

which gives a category or class for each state. We call this type of machine a **classifier** or **controller**. We will study the conrtoller aspect further in the next chapter. For now, we focus on the classification aspect. In the simplest non-trivial case of classifier, there are two categories. The states are divided up into the "accepting" states (class 1, say) and the "rejecting" states (class 0). The machine in this case is called an **acceptor** or **recognizer**. The sequences it accepts are those given by

$$c(F^*(q_0, x)) = 1$$

that is, the sequences *x* such that, when started in state q_0, after reading x, the machine is in a state q such that c(q) = 1. The set of all such x, since it is a set of strings, is a **language** in the sense already discussed. If A designates a finite-state acceptor, then

$$L(A) = \{ x \text{ in } \Sigma^* \mid c(F^*(q_0, x)) = 1\}$$

is the **language accepted by** A.

The structure of a classifier is simpler than that of a transducer, since the output is only a function of the state and not of both the state and input. The structure is shown as follows:

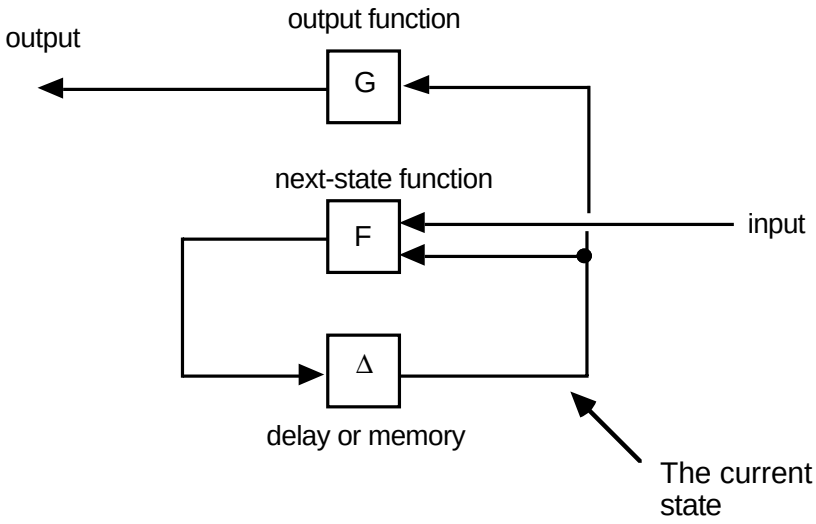


Figure 196: Feedback diagram of classifier finite-state machine structure

A final class of machine, called a **sequencer** or **generator**, is a special case of a transducer or classifier that has a single-letter input alphabet. Since the input symbols are unchanging, this machine generates a fixed sequence, interpreted as either the output sequence of a transducer or the sequence of classifier outputs. An example of a sequencer is a MIDI (Musical Instrument Digital Interface) sequencer, used to drive electronic musical instruments. The output alphabet of a MIDI sequencer is a set of 16-bit words, each having a special interpretation as pitch, note start and stop, amplitude, etc. Although

most MIDI sequencers are programmable, the program typically is of the nature of an initial setup rather than a sequential input.

Description of Finite-State Machines using Graphs

Any finite-state machine can be shown as a graph with a finite set of nodes. The nodes correspond to the states. There is no other memory implied other than the state shown. The start state is designated with an arrow directed into the corresponding node, but otherwise unconnected.

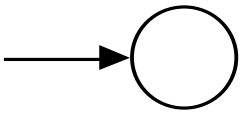


Figure 197: An unconnected in-going arc indicates that the node is the start state

The arcs and nodes are labeled differently, depending on whether we are representing a transducer, a classifier, or an acceptor. In the case of a **transducer**, the arcs are labeled σ/δ as shown below, where σ is the input symbol and δ is the output symbol. The state-transition is designated by virtue of the arrow going from one node to another.

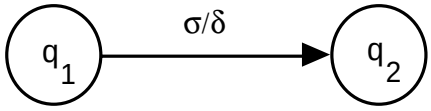


Figure 198: Transducer transition from q_1 to q_2, based on input σ, giving output δ

In the case of a **classifier**, the arrow is labeled only with the input symbol. The categories are attached to the names of the states after /.

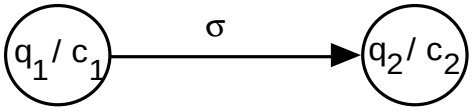


Figure 199: Classifier transition from q_1 to q_2, based on input σ

In the case of a **acceptor**, instead of labeling the states with categories 0 and 1, we sometimes use a double-lined node for an accepting state and a single-lined node for a rejecting state.

Figure 200: Acceptor, an accepting state

Transducer Example

The edge detector is an example of a transducer. Here is its graph:

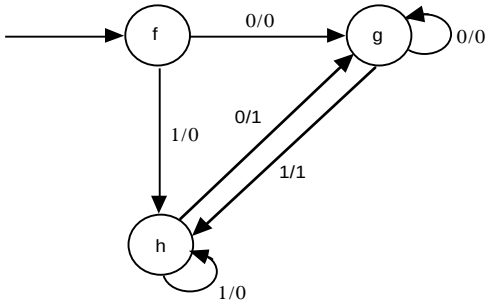


Figure 201: Directed labeled graph for the edge detector

Let us also give examples of classifiers and acceptors, building on this example.

Classifier Example

Suppose we wish to categorize the input as to whether the input so far contains 0, 1, or more than 1 "edges" (transitions from 0 to 1, or 1 to 0). The appropriate machine type is classifier, with the outputs being in the set {0, 1, more}. The name "more" is chosen arbitrarily. We can sketch how this classifier works with the aid of a graph.

The construction begins with the start state. We don't know how many states there will be initially. Let us use a, b, c, ... as the names of the states, with *a* as the start state. Each state is labeled with the corresponding class as we go. The idea is to achieve a finite closure after some number of states have been added. The result is shown below:

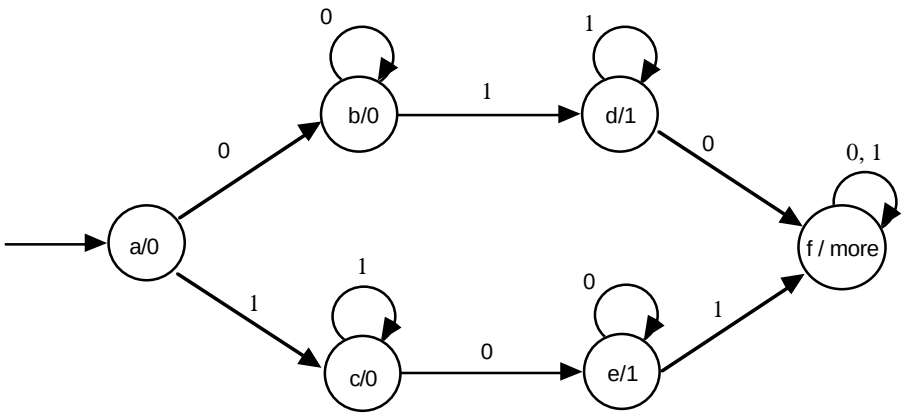


Figure 202: Classifier for counting 0, 1, or more than 1 edges

Acceptor Example

Let us give an acceptor that accepts those strings with exactly one edge. We can use the state transitions from the previous classifier. We need only designate those states that categorize there being one edge as accepting states and the others as rejecting states.

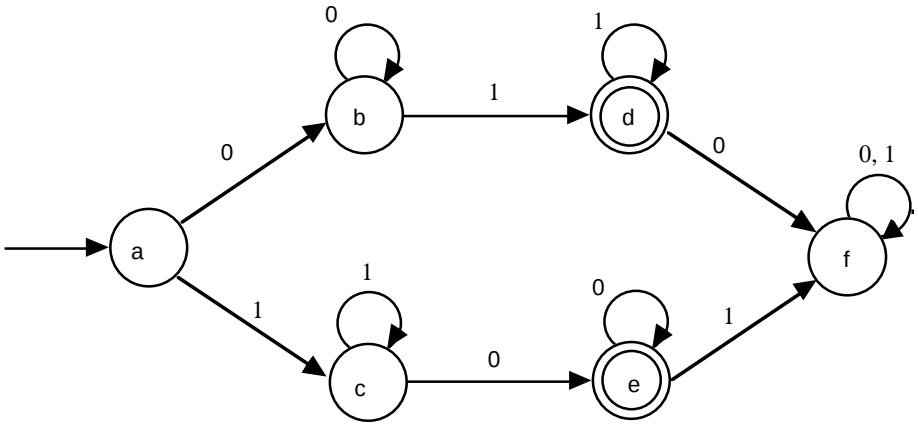


Figure 203: Acceptor for strings with exactly one edge. Accepting states are d and e.

Sequencer Example

The following sequencer, where the sequence is that of the outputs associated with each state, is that for a naive traffic light:

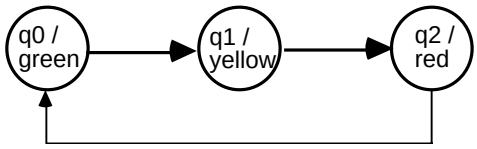


Figure 204: A traffic light sequencer

Exercises

1 •• Consider a program that scans an input sequence of characters to determine whether the sequence as scanned so far represents either an integer numeral, a floating-point numeral, unknown, or neither. As it scans each character, it outputs the corresponding assessment of the input. For example,

Input Scanned	Assessment
1	integer
+	unknown
+1	integer
+1.	floating-point
1.5	floating-point
1e	unknown
1e-1	floating-point
1e.	neither

Describe the scanner as a finite-state transducer using the various methods presented in the text.

2 •• Some organizations have automated their telephone system so that messages can be sent via pushing the buttons on the telephone. The buttons are labeled with both numerals and letters as shown:

1	**2** **a b c**	**3** **d e f**
4 **g h i**	**5** **j k l**	**6** **m n o**
7 **p r s**	**8** **t u v**	**9** **w x y**
*****	**0**	**#**

Notice that certain letters are omitted, presumably for historical reasons. However, it is common to use * to represent letter q and # to represent letter z. Common schemes do not use a one-to-one encoding of each letter. However, if we wanted such an encoding, one method would be to use two digits for each letter:

> The first digit is the key containing the letter.
>
> The second digit is the index, 1, 2, or 3, of the letter on the key. For example, to send the word "cat", we'd punch:
>
> ```
> 2 3   2 1   8 1
>  c     a     t
> ```
>
> An exception would be made for 'q' and 'z', as the only letters on the keys '*' and '#' respectively.

Give the state-transition table for communicating a series of any of the twenty-six letters, where the input alphabet is the set of digits {1, ..., 9, *, #} and the output alphabet is the set of available letters. Note that outputs occur only every other input. So we need a convention for what output will be shown in the transducer in case there is no output. Use λ for this output.

3 •• The device sketched below is capable of partially sensing the direction (clockwise, counterclockwise, or stopped) of a rotating disk, having sectors painted alternating gray and white. The two sensors, which are spaced so as to fit well within a single sector, regularly transmit one of four input possibilities: wg (white-gray), ww (white-white), gw (gray-white), and gg (gray-gray). The sampling rate must be fast enough, compared with the speed of the disk, that artifact readings do not take place. In other words, there must be at least one sample taken every time the disk moves from one of the four input combinations to another. From the transmitted input values, the device produces the directional

information. For example, if the sensors received wg (as shown), then ww for awhile, then gw, it would be inferred that the disk is rotating clockwise. On the other hand, if it sensed gw more than once in a row, it would conclude that the disk has stopped. If it can make no other definitive inference, the device will indicate its previous assessment. Describe the device as a finite-state transducer, using the various methods presented in the text.

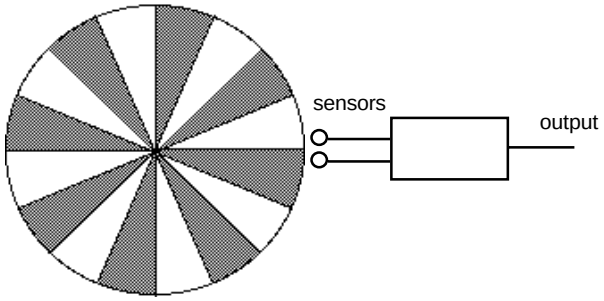


Figure 205: A rotational direction detector

4 •• Decide whether the wrist-watch described below is best represented as a classifier or a transducer, then present a state-transition diagram for it. The watch has a chronograph feature and is controlled by three buttons, *A*, *B*, and *C*. It has three display modes: the time of day, the chronograph time, and "split" time, a saved version of the chronograph time. Assume that in the initial state, the watch displays time of day. If button *C* is pressed, it displays chronograph time. If *C* is pressed again, it returns to displaying time of day. When the watch is displaying chronograph time or split time, pressing *A* starts or stops the chronograph. Pressing B when the chronograph is running causes the chronograph time to be recorded as the split time and displayed. Pressing B again switches to displaying the chronograph. Pressing B when the chronograph is stopped resets the chronograph time to 0.

5 ••• A certain vending machine vends soft drinks that cost $0.40. The machine accepts coins in denominations of $0.05, $0.10, and $0.25. When sufficient coins have been deposited, the machine enables a drink to be selected and returns the appropriate change. Considering each coin deposit and the depression of the drink button to be inputs, construct a state-transition diagram for the machine. The outputs will be signals to vend a drink and return coins in selected denominations. Assume that once the machine has received enough coins to vend a drink, but the vend button has still not been depressed, that any additional coins will just be returned in kind. How will your machine handle cases such as the sequence of coins 10, 10, 10, 5, 25?

6 ••• Consider the problem of controlling traffic at an intersection such as shown below.

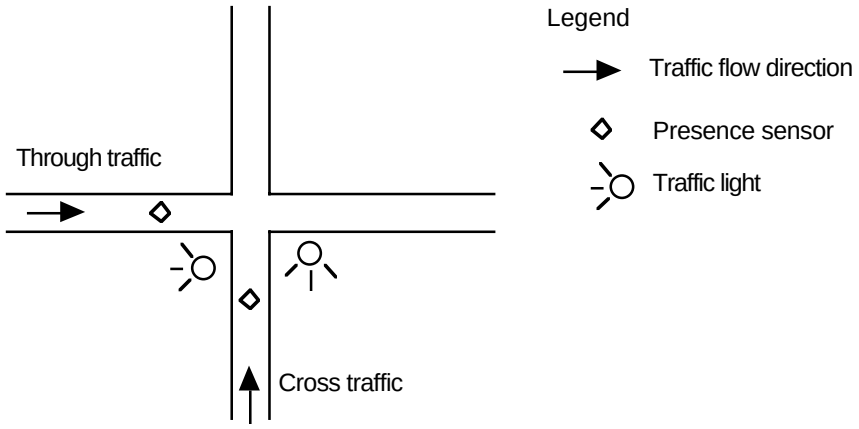


Figure 206: A traffic intersection

Time is divided into equal-length intervals, with sensors sampling the presence of traffic just before the end of an interval. The following priority rules apply:

1. If no traffic is present, through-traffic has the right-of-way.

2. If through-traffic is still present at the end of the first interval during which through-traffic has had the right-of-way, through-traffic is given the right-of-way one additional interval.

3. If cross-traffic is present at the end of the second consecutive interval during which through-traffic has had the right-of-way, then cross-traffic is given the right-of-way for one interval.

4. If cross-traffic is present but through-traffic is absent, cross-traffic maintains the right-of-way until an interval in which through-traffic appears, then through-traffic is given the right-of-way.

Describe the traffic controller as a classifier that indicates which traffic has the right-of-way.

7 ••• A bar code represents bits as alternating light and dark bands. The light bands are of uniform width, while the dark bands have width either equal to, or double, the width of the light bands. Below is an example of a code-word using the bar code. The tick marks on top show the single widths.

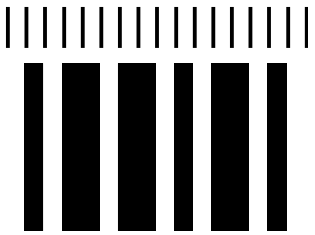


Figure 207: A bar code scheme

Assume that a bar-code reader translates the bands into symbols, L for light, D for dark, one symbol per single width. Thus the symbol sequence for the code-word above would be

```
L D L D D L D D L D L D D L D L
```

A bar pattern represents a binary sequence as follows: a 0 is encoded as LD, while a 1 is encoded as LDD. A finite-state transducer M can translate such a code into binary. The output alphabet for the transducer is {0, 1, _, end}. When started in its initial state, the transducer will "idle" as long as it receives only L's. When it receives its first D, it knows that the code has started. The transducer will give output 0 or 1 as soon it has determined the next bit from the bar pattern. If the bit is not known yet, it will give output _. Thus for the input sequence above, M will produce

```
_ _ 0 _ 1 _ _ 1 _ _ 0 _ 1 _ _ 0
L D L D D L D D L D L D D L D L
```

where we have repeated the input below the output for convenience. The transducer will output the symbol *end* when it subsequently encounters two L's in a row, at which point it will return to its initial state.

a. Give the state diagram for transducer M, assuming that only sequences of the indicated form can occur as input.

b. Certain input sequences should not occur, e.g. L D D D. Give a state-transition diagram for an acceptor A that accepts only the sequences corresponding to a valid bar code.

8 •• A gasoline pump dispenses gas based on credit card and other input from the customer. The general sequence of events, for a single customer is:

Customer swipes credit card through the slot.

Customer enters PIN (personal identification number) on keypad, with appropriate provisions for canceling if an error is made.

Customer selects grade of gasoline.

Customer removes nozzle.

Customer lifts pump lever.

Customer squeezes or releases lever on nozzle any number of times.

Customer depresses pump lever and replaces nozzle.

Customer indicates whether or not a receipt is wanted.

Sketch a state diagram for modeling such as system as a finite-state machine.

Inter-convertibility of Transducers and Classifiers (Advanced)

We can describe a mathematical relationship between classifiers and transducers, so that most of the theory developed for one will be applicable to the other. One possible connection is, given an input sequence x, record the outputs corresponding to the states through which a classifier goes in processing x. Those outputs could be the outputs of an appropriately-defined transducer. However, classifiers are a little more general in this sense, since they give output even for the empty sequence λ, whereas the output for a transducer with input λ is always just λ. Let us work in terms of the following equivalence:

> A transducer T started in state q_0 is equivalent to a classifier C started in state q_0 if, for any non-empty sequence x, the sequence of outputs emitted by T is the same as the sequence of outputs of the states through which C passes.

With this definition in mind, the following would be a classifier equivalent to the edge-detector transducer presented earlier.

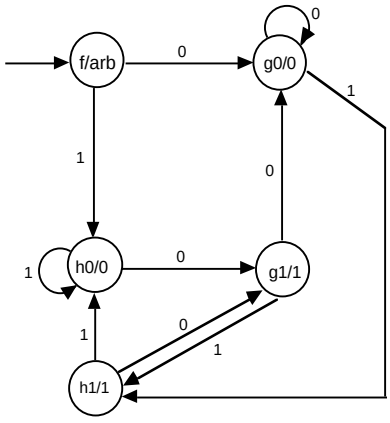


Figure 208: A classifier formally equivalent to the edge-detector transducer

To see how we constructed this classifier, observe that the output emitted by a transducer in going from a state q to a state q', given an input symbol σ, should be the same as the output attached to state q' in the classifier. However, we can't be sure that all transitions into a state q' of a transducer produce the same output. For example, there are two transitions to state *g* in the edge-detector that produce 0 and one that produces 1, and similarly for state h. This makes it impossible to attach a fixed input to either g or h. Therefore we need to "split" the states g and h into two, a version with 0 output and a version with 1 output. Call these resulting states g0, g1, h0, h1. Now we can construct an output-consistent classifier from the transducer. We don't need to split f, since it has a very transient character. Its output can be assigned arbitrarily without spoiling the equivalence of the two machines.

The procedure for converting a classifier to a transducer is simpler. When the classifier goes from state q to q', we assign to the output transition the state output value c(q'). The following diagram shows a transducer equivalent to the classifier that reports 0, 1, or more edges.

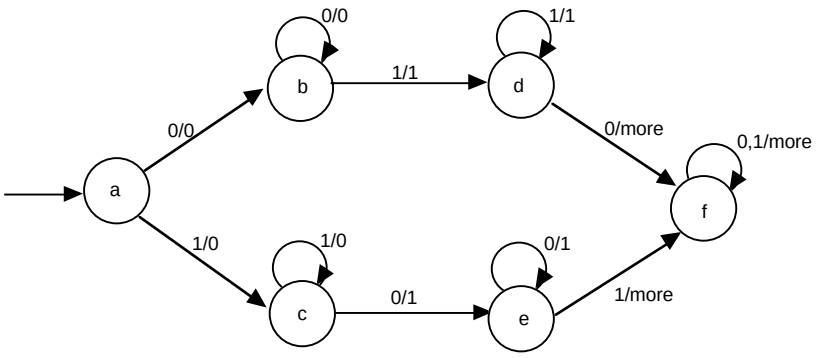


Figure 209: A transducer formally equivalent to the edge-counting classifier

Exercises

1 •• Whichever model, transducer or classifier, you chose for the wrist-watch problem in the previous exercises, do a formal conversion to the other model.

Give a state-transition graph or other equivalent representation for the following machines.

2 •• **MB2 (multipy-by-two)** This machine is a transducer with binary inputs and outputs, both **least-significant bit first**, producing a numeral that is twice the input. That is, if the input is $...x_2x_1x_0$ where x_0 is input first, then the output will be ... $x_2x_1x_00$ where 0 is output first, then x_0, etc. For example:

input	output	input decimal	output decimal
0	0	0	0
01	10	1	2
011	110	3	6
01011	10110	11	22
101011	010110	43	*incomplete*
0101011	1010110	43	86

first bit input ^

Notice that the full output does not occur until a step later than the input. Thus we need to input a 0 if we wish to see the full product. All this machine does is to reproduce the input delayed one step, after invariably producing a 0 on the first step. Thus this machine could also be called a **unit delay machine**.

Answer: Since this machine "buffers" one bit at all times, we can anticipate that two states are sufficient: r0 "remembers" that the last input was 0 and r1 remembers that the last input was 1. The output always reflects the state before the transition, i.e. outputs on arcs from r0 are 0 and outputs on arcs from r1 are 1. The input always takes the machine to the state that remembers the input appropriately.

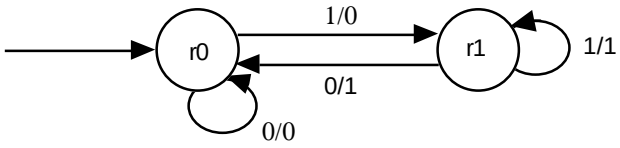


Figure 210: A multiply-by-2 machine

3 •• **MB2^n (multiply-by-2^n, where n is a fixed natural number)** (This is a separate problem for each n.) This machine is a transducer with binary inputs and outputs, both **least-significant bit first**, producing a numeral that is 2^n times as large the input. That is, if the input is ...$x_2x_1x_0$ where x_0 is input first, then the output will be ... $x_2x_1x_00$ where 0 is output first, then x_0, etc.

4 •• **Add1** This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is 1 + the input.

Answer: The states of this machine will represent the value that is "carried" to the next bit position. Initially 1 is "carried". The carry is "propagated" as long as the input bits are 1. When an input bit of 0 is encountered, the carry is "absorbed" and 1 is output. After that point, the input is just replicated.

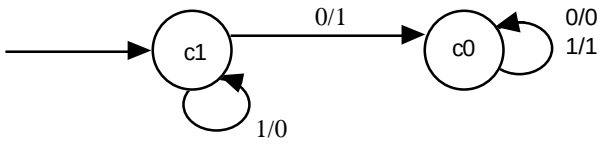


Figure 211: An add-1 machine

5 •• **W2 (Within 2)** This is an acceptor with input alphabet {0, 1}. It accepts those strings such that *for every prefix* of the string, the difference between the number of 0's and the number of 1's is always within 2. For example, 100110101 would be accepted but 111000 would not.

6 ••• **Add3** This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is 3 + the input.

7 ••• **Add-n, where n is a fixed natural number** (This is a separate problem for each n.) This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is n + the input.

8 •• **Binary adder** This is a transducer with binary inputs occurring in pairs. That is, the input alphabet is all pairs over {0, 1}: {(0, 0), (0, 1), (1, 0), (1, 1)}. The inputs are to be interpreted as bits of two binary numerals, least-significant bit first as in the previous problem. The output is a numeral representing the sum of the inputs, also least-significant bit first. As before, we need to input a final (0, 0) if we wish to see the final answer.

		decimal value of	
input	output	input	output
(0, 0)	0	0, 0	0
(0, 1)	1	0, 1	1
(0, 0)(1, 1)	10	1, 1	2
(0, 0)(1, 1)(0, 0)	100	2, 2	4
(0, 0)(1, 1)(1, 1)	110	3, 3	6
first input pair ^			

Answer: Apparently only the value of the "carry" needs to be remembered from one state to the next. Since only two values of carry are possible, this tells us that two states will be adequate.

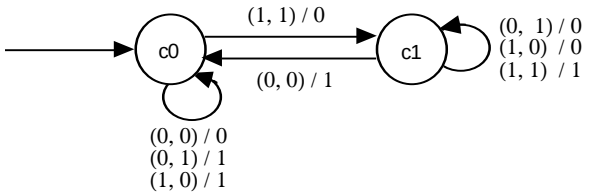


Figure 212: Serial binary adder, least-significant bits first

9 ••• **MB3 (multiply-by-three)** Similar to MB2, except the input is multiplied by 3. For example

		decimal value of	
input	output	input	output
0	0	0	0
01	11	1	3
010	110	2	6
001011	100001	11	33

Note that two final 0's might be necessary to get the full output. Why?

10 •••• **MBN (multiply-by-n, where n is a fixed natural number)** (This is a separate problem for each n.) This machine is a transducer with binary input and outputs, both **least-significant bit first**, producing a numeral that is n times the input.

11 ••• **Binary maximum** This is similar to the adder, but the inputs occur **most-significant digit first** and both inputs are assumed to be the same length numeral.

		decimal value of	
input	output	input	output
(0, 0)	0	0, 0	0
(0, 1)	1	0, 1	1
(0, 1)(1, 1)	11	1, 3	3
(0, 1)(1, 1)(1, 0)110	3, 6	6	
(1, 1)(1, 0)(0, 1)110	6, 5	6	

^ first input pair

12 •• **Maximum classifier** This is a classifier version of the preceding. There are three possible outputs assigned to a state: {tie, 1, 2}, where 1 indicates that the first input sequence is greater, 2 indicates the second is greater, and 'tie' indicates that the two inputs are equal so far.

input	class
(1, 1)	tie
(1, 1)(0, 1)	2
(1, 1)(0, 1)(1, 1)	2
(1, 0)(0, 1)(1, 1)	1

13 •• **1DB3 (Unary divisible by 3)** This is an acceptor with input alphabet {1}. It accepts exactly those strings having a multiple of three 1's (including λ).

14 ••• **2DB3 (Binary divisible by 3)** This is an acceptor with input alphabet {0, 1}. It accepts exactly those strings that are a numeral representing a multiple of 3 in binary, least-significant digit first. (Hint: Simulate the division algorithm.) Thus the accepted strings include: 0, 11, 110, 1001, 1100, 1111, 10010, ...

15 ••• **Sequential combination locks** (an infinite family of problems): A single string over the alphabet is called the "combination". Any string *containing* this combination is accepted by the automaton ("opens the lock"). For example, for the combination 01101, the acceptor is:

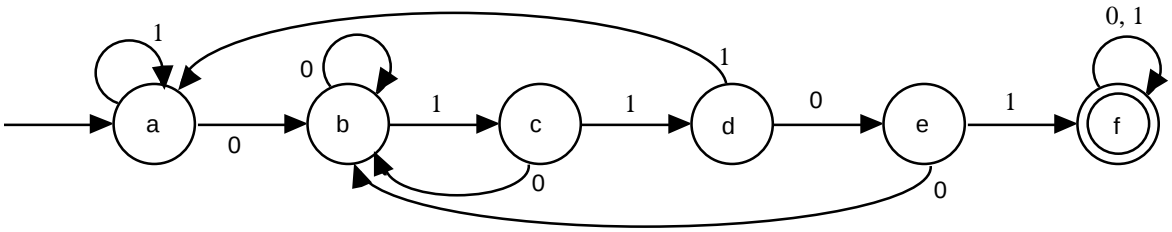


Figure 213: A combination lock state diagram

> The tricky thing about such problems is the construction of the backward arcs; they do not necessarily go back to the initial state if a "wrong" digit is entered, but only back to the state that results from the longest usable suffix of the digits entered so far. The construction can be achieved by the "subset" principle, or by devising an algorithm that will produce the state diagram for any given combination lock problem. This is what is done in a string matching algorithm known as the "Knuth-Morris-Pratt" algorithm.

Construct the state-diagram for the locks with the following different combinations: 1011; 111010; 010010001.

16 ••• Assume three different people have different combinations to the same lock. Each combination enters the user into a different security class. Construct a classifier for the three combinations in the previous problem.

17 ••• The preceding lock problems assume that the lock stays open once the combination has been entered. Rework the example and the problems assuming that the lock shuts itself if more digits are entered after the correct combination, until the combination is again entered.

12.2 Finite-State Grammars and Non-Deterministic Machines

An alternate way to define the language accepted by a finite-state acceptor is through a grammar. In this case, the grammar can be restricted to have a particular form of production. Each production is either of the form:

$$N \rightarrow \sigma M$$

where N and M are auxiliaries and σ is a terminal symbol, or of the form

$$N \rightarrow \lambda$$

recalling that λ is the empty string. The idea is that **auxiliary symbols are identified with states**. The start state is the start symbol. For each transition in an acceptor for the language, of the form

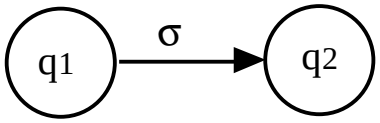


Figure 214: State transition corresponding to a grammar production

there is a corresponding production of the form

$$q_1 \rightarrow \sigma q_2$$

In addition, if q_2 happens to be an accepting state, there is also a production of the form.

$$q_2 \rightarrow \lambda$$

Example: Grammar from Acceptor

For our acceptor for exactly one edge, we can apply these two rules to get the following grammar for generating all strings with one edge:

The start state is *a*. The auxiliaries are {a, b, c, d, e, f}. The terminals are {0, 1}. The productions are:

a ⟶ 0b	c ⟶ 0e
a ⟶ 1c	c ⟶ 1c
b ⟶ 0b	e ⟶ 0e
b ⟶ 1d	e ⟶ 1f
d ⟶ 0f	e ⟶ λ
d ⟶ 1d	f ⟶ 0f
d ⟶ λ	f ⟶ 1f

To see how the grammar derives the 1-edged string 0011 for example, the derivation tree is:

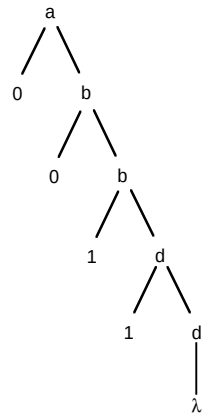


Figure 215: Derivation tree in the previous finite-state grammar, deriving the 1-edged string 0011

While it is easy to see how a finite-state grammar is derived from any finite-state acceptor, the converse is not as obvious. Difficulties arise in productions that have the same lefthand-side with the same terminal symbol being produced on the right, e.g. in a grammar, nothing stops us from using the two productions

a ⟶ 0b
a ⟶ 0c

Yet this would introduce an anomaly in the state-transition diagram, since when given input symbol 0 in state a, the machine would not know to which state to go next:

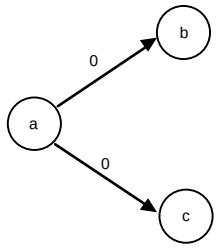


Figure 216: A non-deterministic transition

This automaton would be regarded as non-deterministic, since the next state given input 0 is not determined. Fortunately, there is a way around this problem. In order to show it, we first have to define the notion of "acceptance" by a non-deterministic acceptor.

> A non-deterministic acceptor **accepts** a string if there is some path from a starting state to an accepting state having a sequence of arc labels equal to that string.

We say *a* starting state, rather than *the* starting state, since a non-deterministic acceptor is allowed to have multiple starting states. It is useful to also include λ transitions in non-deterministic acceptors. These are arcs that have λ as their label. Since λ designates the empty string, these arcs can be used in a path but do not contributed any symbols to the sequence.

Example: Non-deterministic to Deterministic Conversion

Recall that a string in the language generated by a grammar consists only of terminal symbols. Suppose the productions of a grammar are (with start symbol a, and terminal alphabet {0, 1}):

$a \rightarrow 0d$	$b \rightarrow 1c$
$a \rightarrow 0b$	$b \rightarrow 1$
$a \rightarrow 1$	$d \rightarrow 0d$
$c \rightarrow 0b$	$d \rightarrow 1$

The language defined by this grammar is the set of all strings ending in 1 that either have exactly one 1 or that consist of alternating 01. The corresponding (non-deterministic) automaton is:

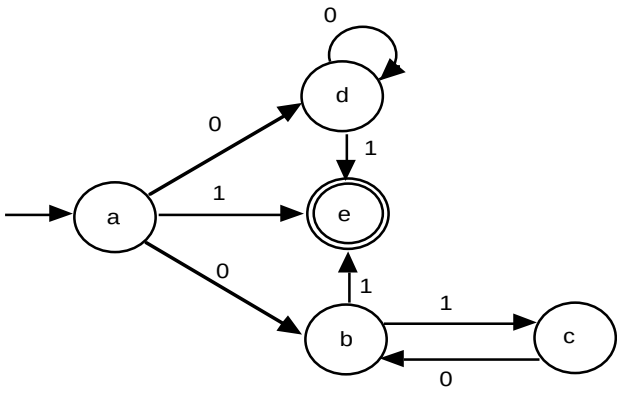


Figure 217: A non-deterministic automaton that accepts the set of all strings ending in 1that have exactly one 1 or consist of an alternating 01's.

There are two instances of non-determinism identifiable in this diagram: the two 0-transitions leaving a and the two 1-transitions leaving b. Nonetheless, we can derive from this diagram a corresponding deterministic finite-automaton. The derivation results in the deterministic automaton shown below.

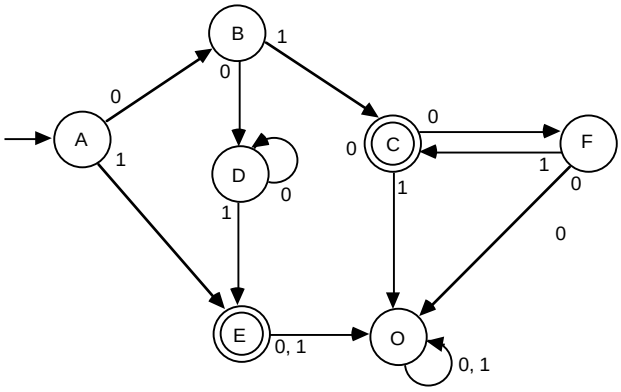


Figure 218: A deterministic automaton that accepts the set of all strings ending in 1 that have exactly one 1 or consist of an alternating 01's.

We can derive a deterministic automaton D from the non-deterministic one N by using *subsets* of the states of N as states of D. In this particular example, the subset associations are as follows:

A ~ {a}
B ~ {b, d}
C ~ {c, e}
D ~ {d}
E ~ {e}
F ~ {b}
O ~ {}

General method for deriving a deterministic acceptor D from a non-deterministic one N:

The **state set** of D is the set of all *subsets* of N.

The **initial state** of D is the set of all initial states of N, together with states reachable from initial states in N using only λ transitions.

There is a **transition** from a set S to a set T of D with label σ (where σ is a single input symbol).

T = {q' | there is a q in S with a sequence of transitions from q to q'
corresponding to a one symbol string σ}

The reason we say *sequence* is due to the possibility of λ transitions; these do not add any new symbols to the string. Note that λ is not regarded as an input symbol.

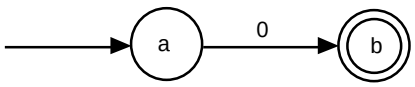
The **accepting states** of the derived acceptor are those that contain at least one accepting state of the original acceptor.

In essence, what this method does is "compile" a breadth-first search of the non-deterministic state graph into a deterministic finite-state system. The reason this works is that the set of all subsets of a finite set is finite.

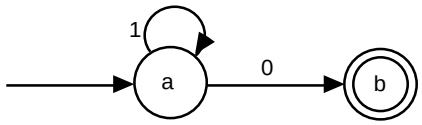
Exercises

Construct deterministic acceptors corresponding to the following non-deterministic acceptors, where the alphabet is {0, 1}.

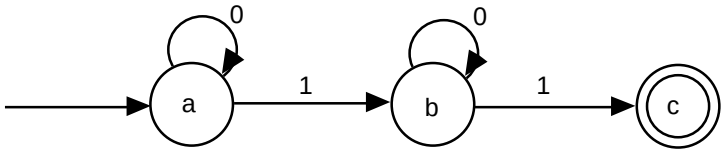
1 •



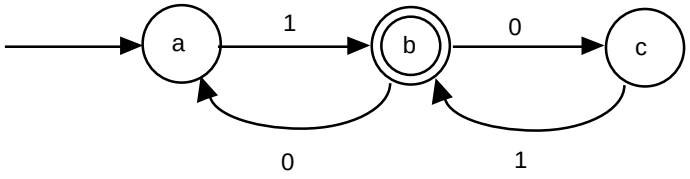
2 •



3 ••



4 ••



12.3 Meaning of Regular Expressions

From the chapter on grammars, you already have a good idea of what regular expressions mean already. While a grammar for regular expression was given in that chapter, for purposes of giving a meaning to regular expressions, it is more convenient to use a grammar that is intentionally ambiguous, expressing constructs in pairs rather than in sequences:

$R \rightarrow$ 'λ'
$R \rightarrow \varnothing$
$R \rightarrow \sigma$, for each letter σ in A
$R \rightarrow R\ R$ // juxtaposition
$R \rightarrow (\ R \mid R\ )$ // alternation
$R \rightarrow (R)^*$ // iteration

To resolve the ambiguity in the grammar, we simply "overlay" on the grammar some conventions about precedence. The standard precedence rules are:

* binds more tightly than either juxtaposition or |

juxtaposition binds more tightly than |

We now wish to use this ambiguous grammar to assign a meaning to regular expressions. With each expression E, the meaning of E is a language, i.e. set of strings, over the alphabet A. We define this meaning recursively, according to the structure of the grammar:

Basis:

- L(λ) is {λ}, the set consisting of one string, the empty string λ.
- L($\varnothing$) is $\varnothing$, the empty set.
- For each **letter** σ in A, and L(σ) is { σ }, the set consisting of one string of one letter, σ.

Induction rules:

- L(RS) = L(R)L(S), where by the latter we mean the set of all strings of the form of the concatenation rs, where r ∈ L(R) and s ∈ L(S).
- L(R | S) = L(R) ∪ L(S).
- L(R*) = L(R)*

To clarify the first bullet, for any two languages L and M, the "set concatenation" LM is defined to be {rs | r ∈ L and s ∈ M}. That is, the "concatenation" of two sets of strings is the set of all possible concatenations, one string taken from the first set and another taken from the second. For example,

{0}{1} is defined to be {01}.

{0, 01}{1, 10} is defined to be {01, 010, 011, 0110}.

{01}{0, 00, 000, ...} is defined to be {010, 0100, 01000, ...}.

To explain the third bullet, we need to define the * operator on an arbitrary language. If L is a language, the L* is defined to be (using the definition of concatenation above)

{λ} ∪ L ∪ LL ∪ LLL ∪ ...

That is, L* consists of all strings formed by concatenating zero or more strings, each of which is in L.

Regular Expression Examples over alphabet {a, b, c}

Expression	**Set denoted**
a | b | c	The set of 1-symbol strings {"a", "b", "c"}
λ | (a | b | c) | (a | b | c)(a | b | c)	The set of strings with two or fewer symbols
a*	The set of strings using only symbol *a*
a*b*c*	The set of strings in which no *a* follows a *b* and no *a* or *b* follows a *c*
(a | b)*	The set of strings using only *a* and *b*.
a* | b*	The set of strings using only *a* or only *b*
(a | b | c)(a | b | c)(a | b | c)*	The set of strings with at least two symbols.
((b | c)* ab (b | c)*)*	The set of strings in which each *a* is immediately followed by a *b*.
(b | c)* | ((b | c)* a (b | c) (b | c)*)* (λ | a)	The set of strings with no two consecutive *a*'s.

Regular expressions are finite symbol strings, but the sets they denote can be finite or infinite. Infinite sets arise only by virtue of the * operator (also sometimes called the Kleene-star operator).

Identities for Regular Expressions

One good way of becoming more familiar with regular expressions is to consider some identities, that is equalities between the sets described by the expressions. Here are some examples. The reader is invited to discover more.

For any regular expressions R and S:

$R \mid S = S \mid R$

$R \mid \varnothing = R$ $\qquad$ $\varnothing \mid R = R$

$R\lambda = R$ $\qquad$ $\lambda R = R$

$R \varnothing = \varnothing$ $\qquad$ $\varnothing R = \varnothing$

$\lambda^* = \lambda$

$\varnothing^* = \lambda$

$R^* = \lambda \mid RR^*$

$(R \mid \lambda)^* = R^*$

$(R^*)^* = R^*$

Exercises

1 •• Determine whether or not the following are valid regular expression identities:

$\lambda\varnothing = \lambda$

$R (S \mid T) = RS \mid RT$

$R^* = \lambda \mid R^*R$

$RS = SR$

$(R \mid S)^* = R^* \mid S^*$

$R^*R = RR^*$

$(R^* \mid S^*)^* = (R \mid S)^*$

$(R^*S^*) = (R \mid S)^*$

For any n, $R^* = \lambda \mid R \mid RR \mid RRR \mid .... \mid R^{n-1} \mid R^nR^*$, where R^n is an abbreviation for RR....R. (n times)

2 ••• Equations involving languages with a language as an unknown can sometimes be solved using regular operators. For example,

$S = RS \mid T$

can be solved for unknown S by the solution $S = R^*T$. Justify this solution.

3 ••• Suppose that we have grammars that respectively generate sets L and M as languages. Show how to use these grammars to form a grammar that generates each of the following languages

$L \cup MLM$ L^*

Regular Languages

The regular operators (|, *, and concatenation) are applicable to any languages. However, a special name is given to languages that can be constructed using only these operators and languages consisting of a single string, and the empty set.

Definition: A language (set of strings over a given alphabet) is called *regular* if it is a set of strings denoted by some regular expression. (A regular language is also called a *regular set*.)

Let us informally explore the relation between regular languages and finite-state acceptors. The general idea is that the regular languages exactly characterize sets of paths from the initial state to *some* accepting state. We illustrate this by giving an acceptor for each of the above examples.

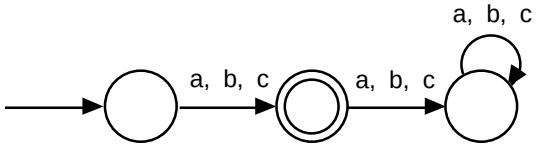


Figure 219: Acceptor for a | b | c

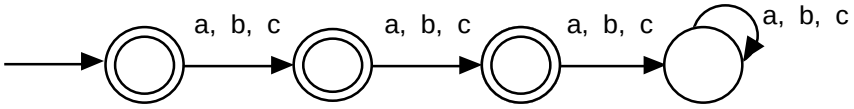


Figure 220: Acceptor for λ | (a | b | c) | (a | b | c)(a | b | c)

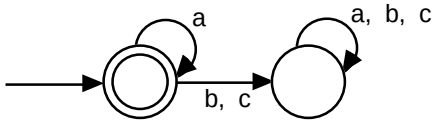


Figure 221: Acceptor for a*

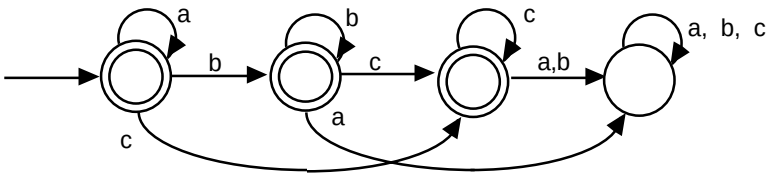


Figure 222: Acceptor for a*b*c*

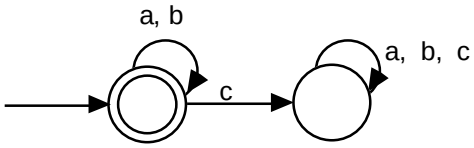


Figure 223: Acceptor for (a | b)*

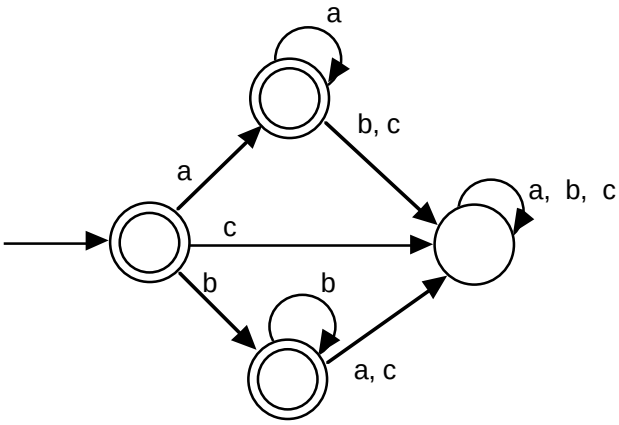


Figure 224: *Acceptor for* a* | b*

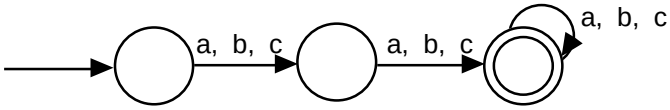


Figure 225: Acceptor for (a | b | c)(a | b | c)(a | b | c)*

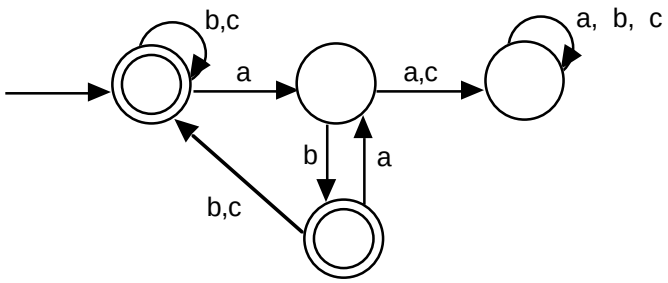


Figure 226: Acceptor for ((b | c)* ab (b | c))*

As we can see, the connection between regular expressions and finite-state acceptors is rather close and natural. The following result makes precise the nature of this relationship.

> **Kleene's Theorem** (Kleene, 1956) A language is regular iff it is accepted by some finite-state acceptor.

The "if" part of Kleene's theorem can be shown by an algorithm similar to Floyd's algorithm. The "only if" part uses the non-deterministic to deterministic transformation.

The Language Accepted by a Finite-State Acceptor is Regular

The proof relies on the following constructive method:

> Augment the graph of the acceptor with a single distinguished starting node and accepting node, connected via λ-transitions to the original initial state and accepting states in the manner shown below. The reason for this step is to isolate the properties of being initial and accepting so that we can more easily apply the transformations in the second step.

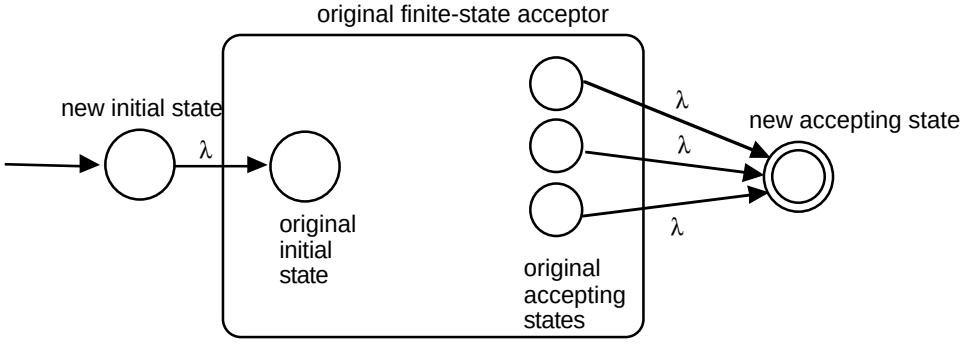


Figure 227: Modifying acceptor in preparation for deriving the regular expression

One at a time, eliminate the nodes in the original acceptor, preserving the set of paths through each node by recording an appropriate regular expression between each pair of other nodes. When this process is complete, the regular expression connecting the initial state to the accepting state is the regular expression for the language accepted by the original finite-state machine.

To make the proof complete, we have to describe the node elimination step. Suppose that prior to the elimination, the situation is as shown below.

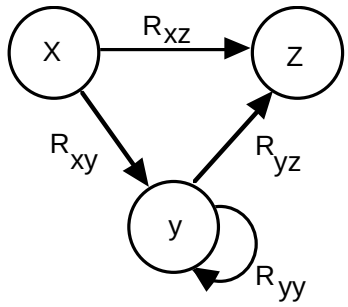


Figure 228: A situation in the graph before elimination of node y

Here x and z represent arbitrary nodes and y represents the node being eliminated. A variable of the form R_{ij} represents the regular expression for paths from i to j using nodes previously eliminated. Since we are eliminating y, we replace the previous expression R_{xz} with a new expression

$$R_{xz} \mid R_{xy} R_{yy}^{*} R_{yz}$$

The rationale here is that Rxz represents the paths that were there before, and $R_{xy} R_{yy}^{*} R_{yz}$ represents the paths that went through the eliminated node y.

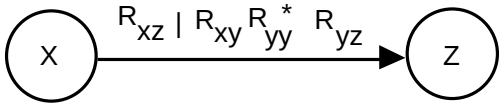


Figure 229: The replacement situation after eliminating node y

The catch here is that we must perform this updating for every pair of nodes x, z, including the case where x and z are the same. In other words, if there are m nodes left, then m^2 regular expression updates must be done. Eliminating each of n nodes then requires $O(n^3)$ steps. The entire elimination process is very similar to the Floyd and Warshall algorithms discussed in the chapter on Complexity. The only difference is that here we are dealing with the domain of regular expressions, whereas those algorithms dealt with the domains of non-negative real numbers and bits respectively.

Prior to the start of the process, we can perform the following simplification:

> Any states from which no accepting state is reachable can be eliminated, along with arcs connecting to or from them.

Example: Regular Expression Derivation

Derive a regular expression for the following finite-state acceptor:

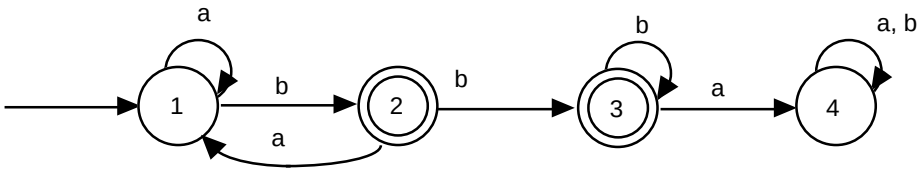


Figure 230: A finite-state acceptor from which a regular expression is to be derived

First we simplify by removing node 4, from which no accepting state is reachable. Then we augment the graph with two new nodes, 0 and 5, connected by λ-transitions. Notice that for some pairs of nodes there is no connection. This is equivalent to the corresponding regular expression being ∅. Whenever ∅ is juxtaposed with another regular expression, the result is equivalent to ∅. Similarly, whenever λ is juxtaposed with another regular expression R, the result is equivalent to R itself.

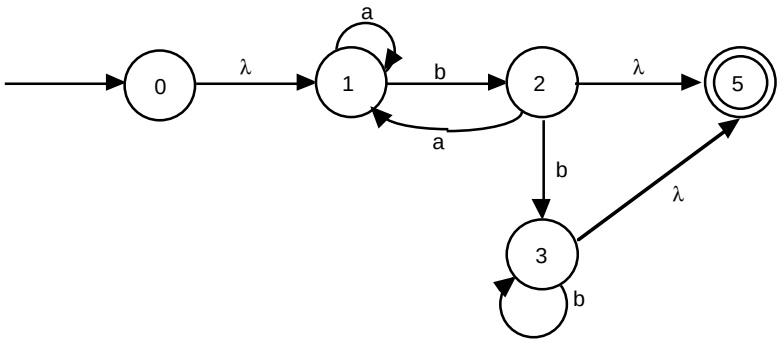


Figure 231: The first step in deriving a regular expression. Nodes 0 and 5 are added.

Now eliminate one of the nodes 1 through 3, say node 1. Here we will use the identity that states λa*b = a*b.

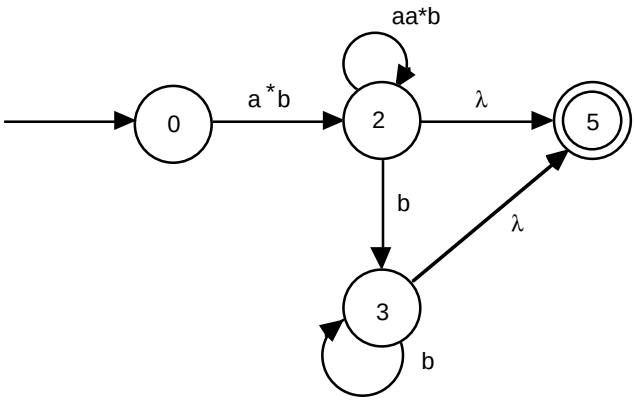


Figure 232: After removal of node 1

Next eliminate node 2.

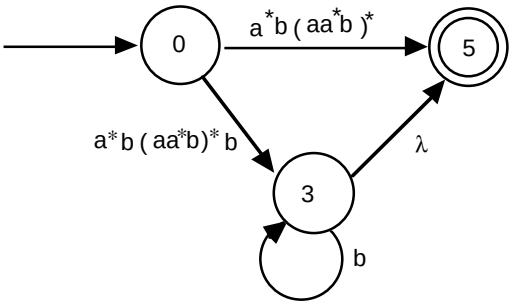


Figure 233: After removal of node 2

Finally eliminate node 3.

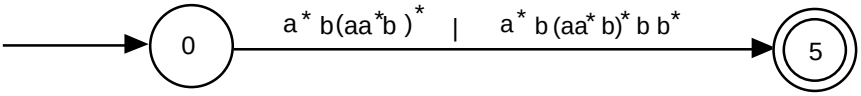


Figure 234: After removal of node 1

The derived regular expression is

$$a^*b(aa^*b)^* \mid a^*b(aa^*b)^*bb^*$$

Every Regular Language is Accepted by a Finite-State Acceptor

We already know that we can construct a deterministic finite-state acceptor equivalent to any non-deterministic one. Hence it is adequate to show how to derive a non-deterministic finite-state acceptor from a regular expression. The paths from initial node to accepting node in the acceptor will correspond in an obvious way to the strings represented by the regular expression.

Since regular expressions are defined inductively, it is very natural that this proof proceed along the same lines as the definition. We expect a basis, corresponding to the base cases λ, $\varnothing$, and σ (for σ each in A). We then assume that an acceptor is constructable for regular expressions R and S and demonstrate an acceptor for the cases RS, R | S, and R^*. The only thing slightly tricky is connecting the acceptors in the inductive cases. It might be necessary to introduce additional states in order to properly isolate the paths in the constituent acceptors. Toward this end, we stipulate that

(i) the acceptors constructed shall always have a single initial state and single accepting state.

(ii) no arc is directed from some state into the initial state

Call these **property P**.

Basis: The acceptors for λ, $\varnothing$, and σ (for σ each in A) are as shown below:

Figure 235: Acceptor for $\varnothing$ with property P

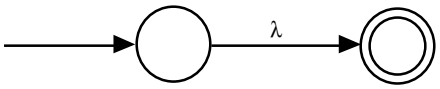


Figure 236: Acceptor for λ (the empty sequence) with property P

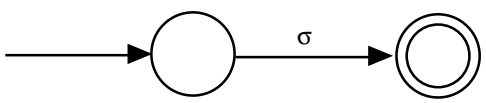


Figure 237: Acceptor for σ (where $\sigma \in$ A) with property P

Induction Step: Assume that acceptors for R and S, with property P above, have been constructed, with their single initial and accepting states as indicated on the left and right, respectively.

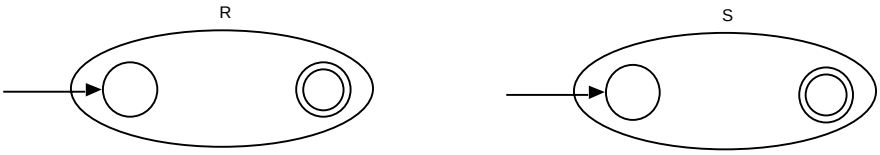


Figure 238: Acceptors assumed to exist for R and S respectively, having property P

Then for each of the cases above, we construct new acceptors that accept the same language and which have property P, as now shown:

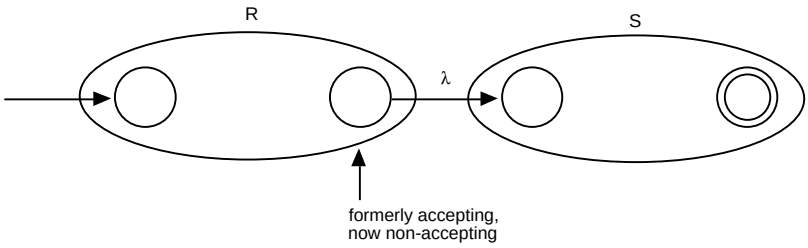


Figure 239: Acceptor for RS, having property P

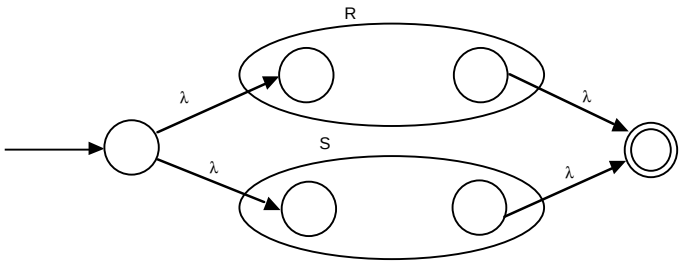


Figure 240: Acceptor for R | S, having property P

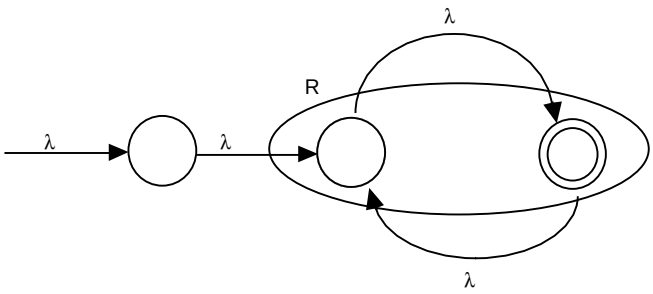


Figure 241: Acceptor for R*, having property P

Regular expressions in UNIX®

Program *egrep* is one of several UNIX® tools that use some form of regular expression for pattern matching. Other such tools are *ed*, *ex*, *sed*, *awk*, and *archie*. The notations appropriate for each tool may differ slightly. Possible usage:

egrep *regular-expression filename*

searches the file line-by-line for lines containing strings matching the regular-expression and prints out those lines. The scan starts anew with each line. In the following description, 'character' means excluding the newline character:

A single character not otherwise endowed with special meaning matches that character. For example, 'x' matches the character x.

The character '.' matches any character.

A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression.

Effectively a regular expression used for searching is preceded and followed by an implied .*, meaning that any sequence of characters before or after the string of interest can exist on the line. To exclude such sequences, use ^ and $:

The character ^ matches the beginning of a line.

The character $ matches the end of a line.

A regular expression followed by a + (plus) matches a sequence of 1 or more matches of the regular expression.

A regular expression followed by a ? (question mark) matches a sequence of 0 or 1 matches of the regular expression.

A \ followed by a single character other than newline matches that character. This is used to escape from the special meaning given to some characters.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in a-z0-9, which means all characters in the range a-z and 0-9. A literal - in such a context must be placed after \ so that it can't be mistaken as a range indicator.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

Care should be taken when using the characters $ * [ ] ^ | ( ) and \ in the expression as they are also meaningful to the various shells. **It is safest to enclose the entire expression argument in single quotes**.

Examples: UNIX Regular Expressions	
Description of lines to be selected	**Regular Expression**
containing the letters `qu` in combination	`qu`
beginning with `qu`	`^qu`
ending with `az`	`az$`
beginning with `qu` and ending with `az`	`^qu.*az$`
containing the letters `qu` or `uq`	`uq|qu`
containing two or more `a`'s in a row	`a.*a`
containing four or more `i`'s	`i.*i.*i.*i`
containing five or more `a`'s and `i`'s	`[ai].*[ai].*[ai].*[ai].*[ai]`
containing `ai` at least twice	`(ai).*(ai)`
containing `uq` or `qu` at least twice	`(uq|qu).*(uq|qu)`

Exercises

Construct deterministic finite-state acceptors for the following regular expressions:

1 • 0*1*

2 • (0*1*)*

3 •• (01 | 011)*

4 •• (0* | (01)*)*

5 •• (0 | 1)*(10110)(0 | 1)*

6 ••• The regular operators are concatenation, union (|), and the * operator. Because any combination of regular languages using these operators is itself a regular language, we say that the regular languages are *closed under* the regular operators. Although intersection and complementation (relative to the set of all strings, Σ^*) are not included among the regular languages, it turns out that the regular languages are closed under these operators as well. Show that this is true, by using the connection between regular languages and finite-state acceptors.

7 ••• Devise a method for determining whether or not two regular expressions denote the same language.

8 •••• Construct a program that inputs a regular expression and outputs a program that accepts the language denoted by that regular expression.

9 ••• Give a UNIX regular expression for lines containing floating-point numerals.

12.4 Synthesizing Finite-State Machines from Logical Elements

We now wish to extend techniques for the implementation of functions on finite domains in terms of logical elements to implementing finite-state machines. One reason that this is important is that digital computers are constructed out of collections of finite-state machines interconnected together.As already stated, the input sequences for finite-state machines are elements of an infinite set Σ^*, where Σ is the input alphabet. Because the output of the propositional functions we studied earlier were simply a *combination* of the input values, those functions are called **combinational**, to distinguish them from the more general functions on Σ^*, which are called **sequential**.

We will show how the implementation of machines can be decomposed into combinational functions and memory elements, as suggested by the equation

Sequential Function = Combinational Functions + Memory

Recall the earlier structural diagrams for transducer and classifiers, shown as "feedback" systems. Note that these two diagrams share a common essence, namely the next-state portion. Initially, we will focus on how just this portion is implemented. The rest of the machine is relatively simple to add.

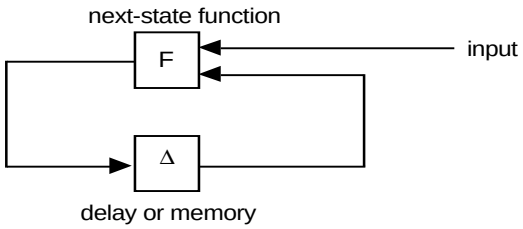


Figure 242: The essence of finite-state machine structure

Implementation using Logic Elements

Before we can "implement" such a diagram, we must be clearer on what items correspond to *changes* of input, output, and state. The combinational logical elements, such as AND-gates and OR-gates, as discussed earlier are abstractions of physical devices. In those devices, the logical values of 0 and 1 are interpretations of physical states. The output of a device is a function of its inputs, *with some qualification.* No device can change state instantaneously. When the input values are first presented, the device's output might be in a different state from that indicated by the function. There is some *delay time* or *switching time* associated with the device that must elapse before the output stabilizes to the value prescribed by the function. Thus, each device has an inherent *sequential* behavior, even if we choose to think of it as a combinational device.

Example Consider a 2-input AND-gate. Suppose that a device implements this gate due to our being able to give logical values to two voltages, say LO and HI, which correspond to 0 and 1 respectively. Then, observed over time, we might see the following behavior of the gate in response to changing inputs. The arrows in the diagram indicate a causal relationship between the input changes and output changes. Note that there is always some delay associated with these changes.

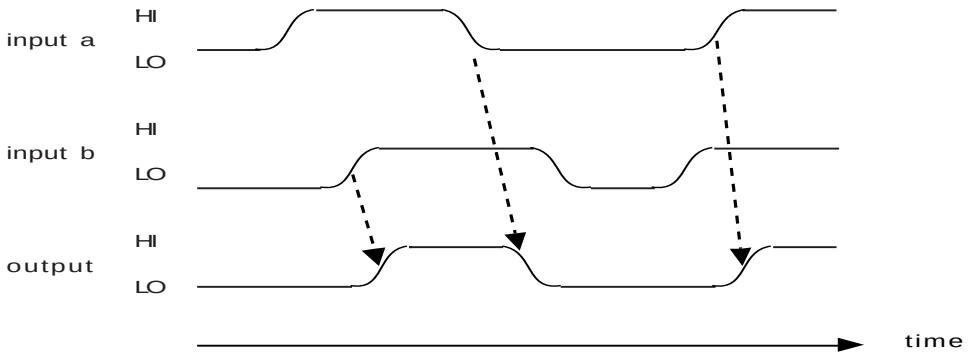


Figure 243: Sequential behavior of an AND-gate

Modeling the sequential behavior of a device can be complex. Computer designers deal with an abstraction of the behavior in which the outputs can only change at specific instants. This simplifies reasoning about behaviors. The abstract view of the AND-gate shown above can be obtained by straightening all of the changes of the inputs and outputs, to make it appear as if they were instantaneous.

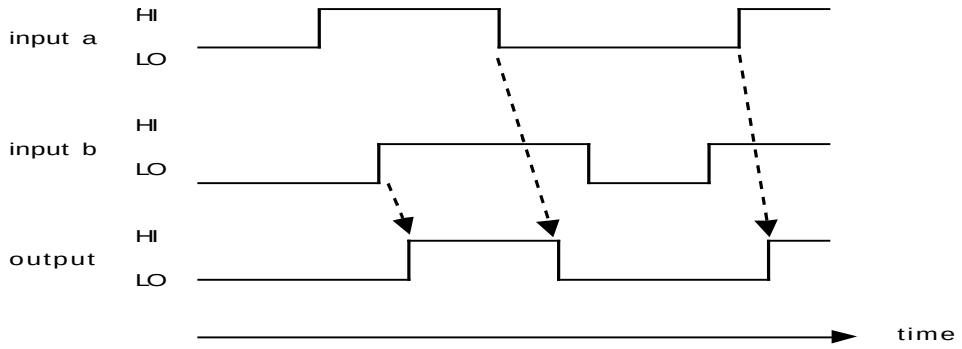


Figure 244: An abstraction of the sequential behavior of an AND-gate

Quantization and Clocks

In order to implement a sequential machine with logic gates, it is necessary to select a scheme for quantizing the values of the signal. As suggested by the preceding diagram, the signal can change *continuously*. On the other hand, the finite-state machine abstraction requires a series of discrete input and output values. For example, as we look at input *a* in the preceding diagram, do we say that the corresponding sequence is 0101 based on just the input changes? If that were the case, what would be the input corresponding to sequence 00110011? In other words, how do we know that a value that stays high for some time is a single 1 or a series of 1's? The most common means of resolving this issue is to use a system-wide clock as a timing standard. The clock "ticks" at regular intervals, and the value of a signal can be *sampled* when this tick occurs.

The effect of using a clock is to superimpose a series of tick marks atop the signals and agree that the discrete valued signals correspond to the values at the tick marks. Obviously this means that the discrete interpretation of the signals depends on the clock interval. For example, one quantization of the above signals is shown as follows:

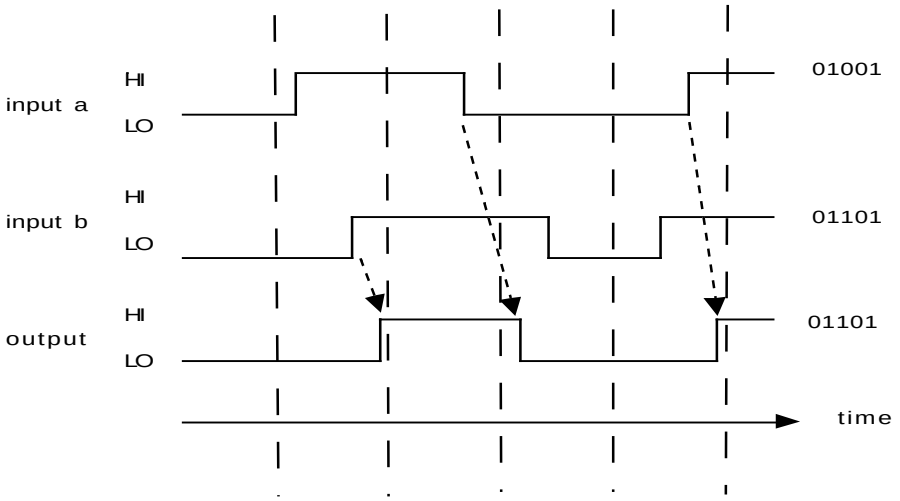


Figure 245: AND-gate behavior with one possible clock quantization

Corresponding to the five ticks, the first input sequence would be 01001, the second would be 01101, and the output sequence would be 01101. Notice that the output is not quite the AND function of the two inputs, as we might have expected. This is due to the fact that the second output change was about to take place when the clock ticked and the previous output value carried over. Generally we avoid this kind of phenomenon by designing such that the changes take place between ticks and at each tick the signals are, for the moment, stable.

The next figure shows the same signals with a slightly wider clock interval superimposed. In this instance, no changes straddle the clock ticks, and the input output sequences appear to be what is predicted by the definition of the AND function.

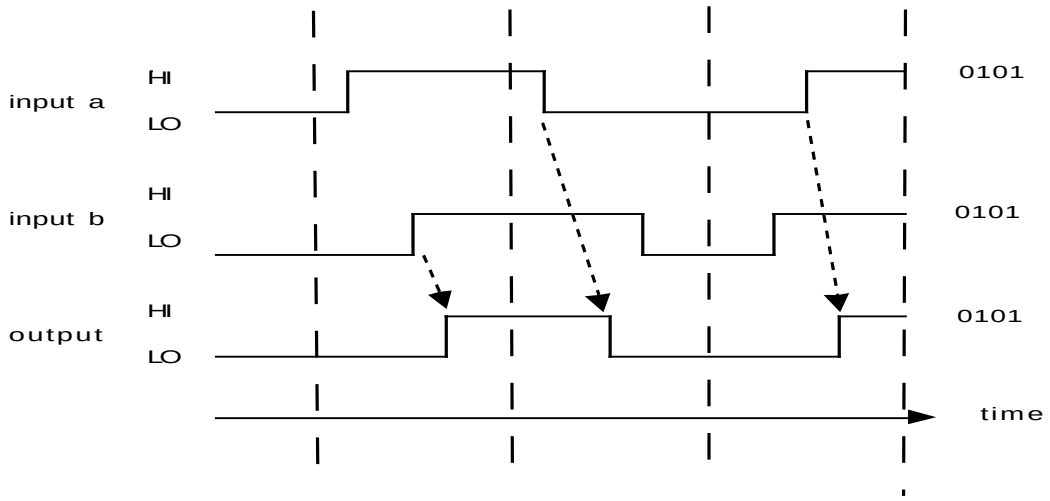


Figure 246: AND-gate behavior with wider quantization

Flip-Flops and Clocks

As stated above, in order to maintain the effect of instantaneous changes when there really is no such thing, the inputs of gates are only *sampled* at specific instants. By using the value of the sample, rather than the true signal, we can approach the effect desired. In order to hold the value of the sample from one instant to the next, a memory device known as a **D flip-flop** is used.

The D flip-flop has two different kinds of inputs: a signal input and a clock input. Whenever the clock "ticks", as represented, say, by the *rising* edge of a square wave, the signal input is sampled and held until the next tick. In other words, the flip-flop "remembers" the input value until the next tick; then it takes on the value at that time.

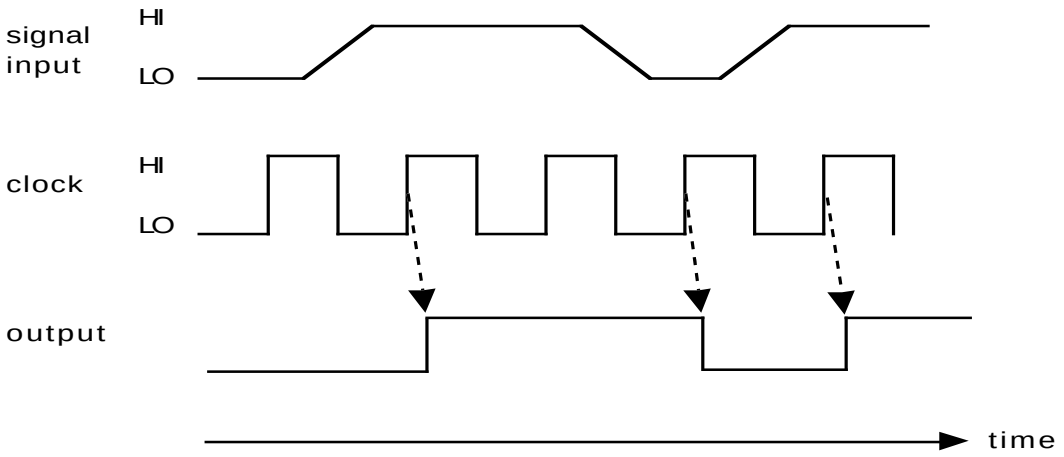


Figure 247: D flip-flop behavior:
The output of the flip-flop changes only in response to the rising edge of the clock, and will reflect the value of the signal input at that time.

Note that the signal input can change in between clock ticks, but it should not be changing near the same time. If it does, a phenomenon known as "meta-stability" can result, which can upset the abstraction as presented. We shall say more about this later.

Clock-based design is called ***synchronous design***. This is not the only form of design, but it is certainly the most prevalent, with at least 99% of computer design being based on this model. We will indicate more about the reasons for this later, but for now, synchronous design is the mode on which we will concentrate.

In synchronous design, the inputs to a device will themselves be outputs of flip-flops, and will change after the clock ticks. For example, the following diagram shows an AND-gate in the context of D flip-flops controlled by a common clock. The inputs to a and b are not shown. However, we assume that they change between clock ticks and thus the outputs of a and b will change right after the clock tick, as will the output c.

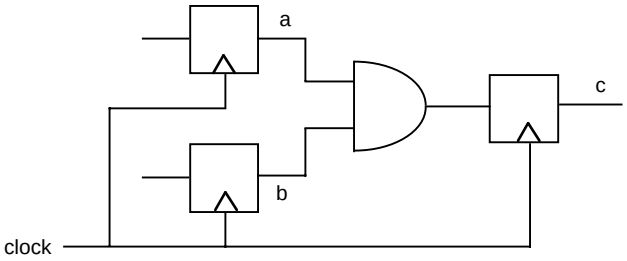


Figure 248: An AND-gate in a synchronous system.
Inputs to the flip-flops with outputs a and b are not shown.

The next diagram shows a sample behavior of the AND-gate in a synchronous system. This should be compared with the abstracted AND-gate presented earlier, to verify that synchronous design implements the abstract behavior.

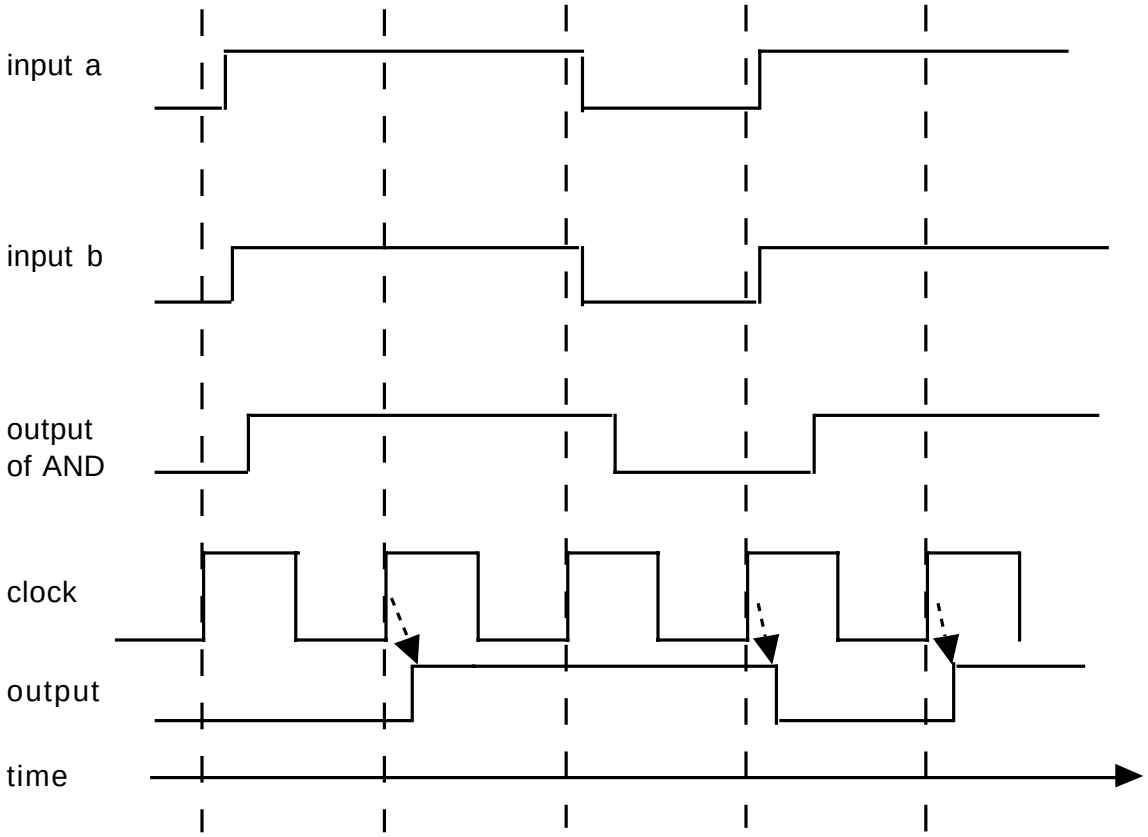


Figure 249: Example behavior of the AND-gate in a synchronous system. Note that the output only changes right after the rising edge of the clock.

The assumption in synchronous design is that the inputs to a device are held constant during each clock interval. During the interval itself, the device has an opportunity to change to the value represented by its logical function. In fact, the length of the interval is chosen in such a way that the device can achieve this value by the end of the interval. At the end of the interval, the output of the device will thus have stabilized. It can then be used as the input to some other device.

Closing the Loop

The previous example of an AND-gate in the context of three flip-flops can be thought of as a simple sequential machine. The state of the machine is held in the output flip-flop c. Thus, the current state always represents the logical AND of the inputs at the previous clock tick. In general, the state is a function of not just the inputs, but also the previous state. This can be accomplished by using the output of the state flip-flop to drive the next

state. An example obtained by modifying the previous example is shown below. Here we connect the output c to the place where the input a was.

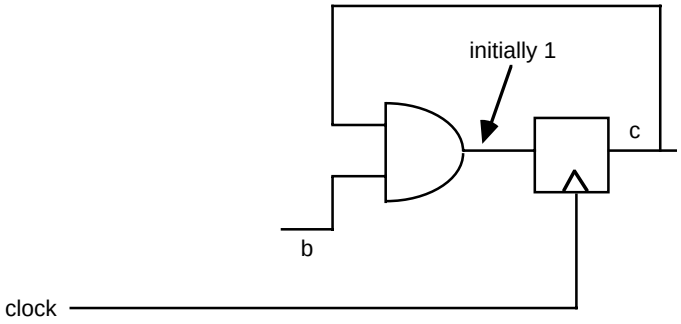


Figure 250: A sequential machine that remembers if b were ever 0

Suppose that we observe this machine from a point at which the output flip-flop c is 1. At the next clock tick, if b is 1, then the flip-flop will stay at 1. However, if b is 0, then the flip-flop will be switched to 0. Once flip-flop c is at 0, it will stay there forever, because no input value *anded* with 0 will ever give 1. The following diagram shows a possible behavior.

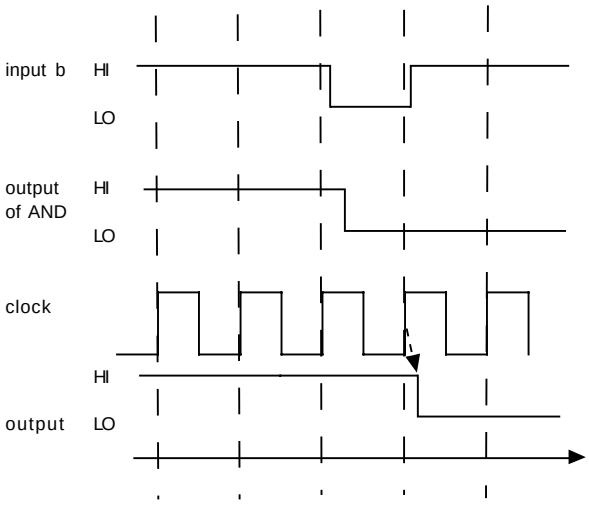


Figure 251: Example behavior of the previous sequential machine

The timing diagram above shows only one possible behavior. To capture all possible behaviors, we need to use the state-transition diagram, as shown below. Again, the state in this case is the output of flip-flop c.

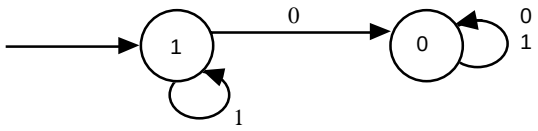


Figure 252: The state diagram for a machine that remembers if it ever saw 0

The state-transition structure of this machine is the same as that of a transducer that adds 1 to a binary representation, least-significant-bit-first:

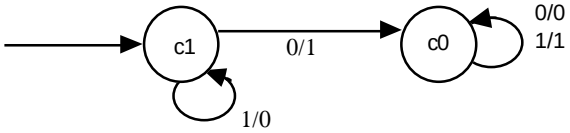


Figure 253: The add-1 transducer

Before giving the general method for synthesizing the logic from a state-transition behavior, we give a couple more examples of structure vs. function.

Sequential Binary Adder Example

This is the essence of the sequential binary adder that adds a pair of numerals together, least-significant bit first. Its state remembers the carry. We present both the full transducer and the abstracted state-transition behavior.

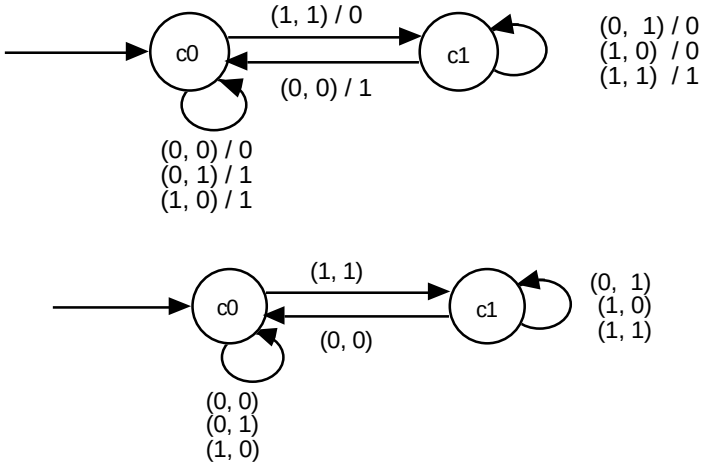


Figure 254: Transducer and state-transition behavior for the binary adder

The machine has 2 inputs that are used in parallel, one for each bit of the two addends. Assuming that we represent the carry by the 1 or 0 value of the output flip-flop, the structure of the adder can be realized as follows:

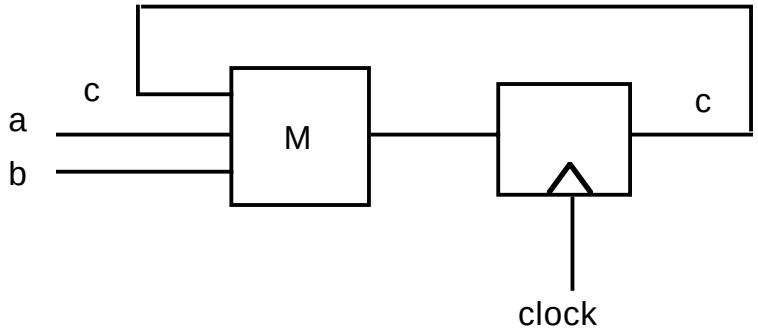


Figure 255: State-transition implementation of the binary adder

The box marked M is the majority combination function, as given by the following table:

a	**b**	**c**	**F**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

As we can see, the output of function M is 1 iff at least two out of three inputs are 1. These combinations could be described by giving the minterm form, or the simplified form:

$$F(a, b, c) = ab + ac + bc$$

Were we to implement this M using AND- and OR- gates, the result would appear as:

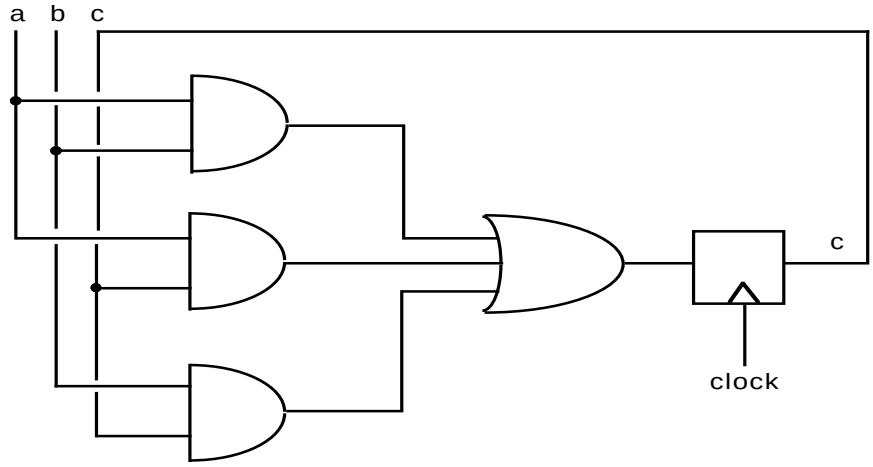


Figure 256: The binary adder state-transition behavior implemented using combinational gates and a flip-flop

Combination Lock Example

This example, a combination lock with combination 01101, was given earlier:

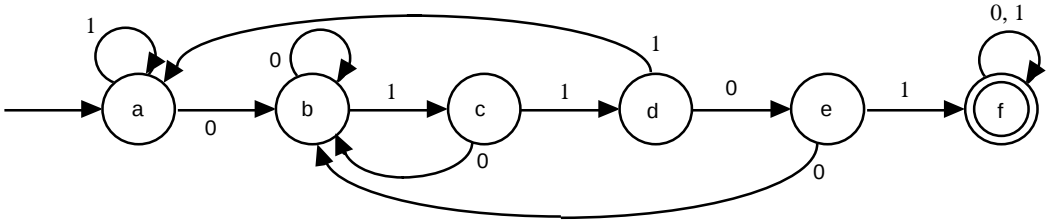


Figure 257: Combination lock state transitions

Suppose we encode the states by using three flip-flops, u, v, and w, as follows:

	flip-flops		
state name	**u**	**v**	**w**
a	0	0	0
b	0	0	1
c	0	1	0
d	0	1	1
e	1	0	0
f	1	1	1

From the tabular form for the state-transition function:

	next state as a function of input	
current state	**0**	**1**
a	b	a
b	b	c
c	b	d
d	e	a
e	b	f
f	f	f

we *transcribe* the table by substituting (e.g. by using a text editor) flip-flop values for each state. This is the same process we used in implementing combinational logic functions.

	next uvw as a function of input	
current uvw	**0**	**1**
000	001	000
001	001	010
010	001	011
011	100	000
100	001	111
111	111	111

For each flip-flop, we derive the next-state in terms of the current one simply by separating this table:

	next u as a function of input	
current uvw	**0**	**1**
000	0	0
001	0	0
010	0	0
011	1	0
100	0	1
111	1	1

Letting x represent the input, from this table, we can see that

next u = u'vwx' + uv'w'x + uvwx' + uvwx

(using the minterm form),but a simpler version derived from considering "don't cares" is:

next u = vwx' + ux

	next v as a function of input	
current uvw	**0**	**1**
000	0	0
001	0	1
010	0	1
011	0	0
100	0	1
111	1	1

From this table, we can derive:

next v = u'v'wx + u'vw'x + uv'w'x + uvw

	next w as a function of input	
current uvw	**0**	**1**
000	1	0
001	1	0
010	1	1
011	0	0
100	1	1
111	1	1

From this table, we can derive the simplified form:

next w =v'x' + vw' + u

Putting these together, we can realize the combination lock as shown on the next page.

12.5 Procedure for Implementing a State-Transition Function

To implement a state-transition function for a finite-state machine in terms of combinational logic and flip-flops:

1. Choose encodings for the input alphabet Σ and the state set Q.

2. Transcribe the table for the state-transition function F: $Q \times \Sigma \rightarrow Q$ into propositional logic functions using the selected encodings.

3. Implement the transcribed F functions from logical elements

4. The functions thus implemented are used as inputs to a bank of D flip-flops, one per bit in the state encoding.

Inclusion of Output Functions

In order to synthesize a finite-state machine having output, we need to augment the state-transition implementation with an output function implementation. Fortunately, the output function is simply a combinational function of the state (in the case of a classifier or acceptor) or of the state and input (in the case of a transducer).

Example: Inclusion of Output for the Combination Lock Example

We see that the lock accepts only when in state f. Equating acceptance to an output of 1, we see that the lock produces a 1 output only when uvw = 1. Therefore, we need only add an AND-gate with inputs from all three flip-flops to get the acceptor output. The complete lock is shown below.

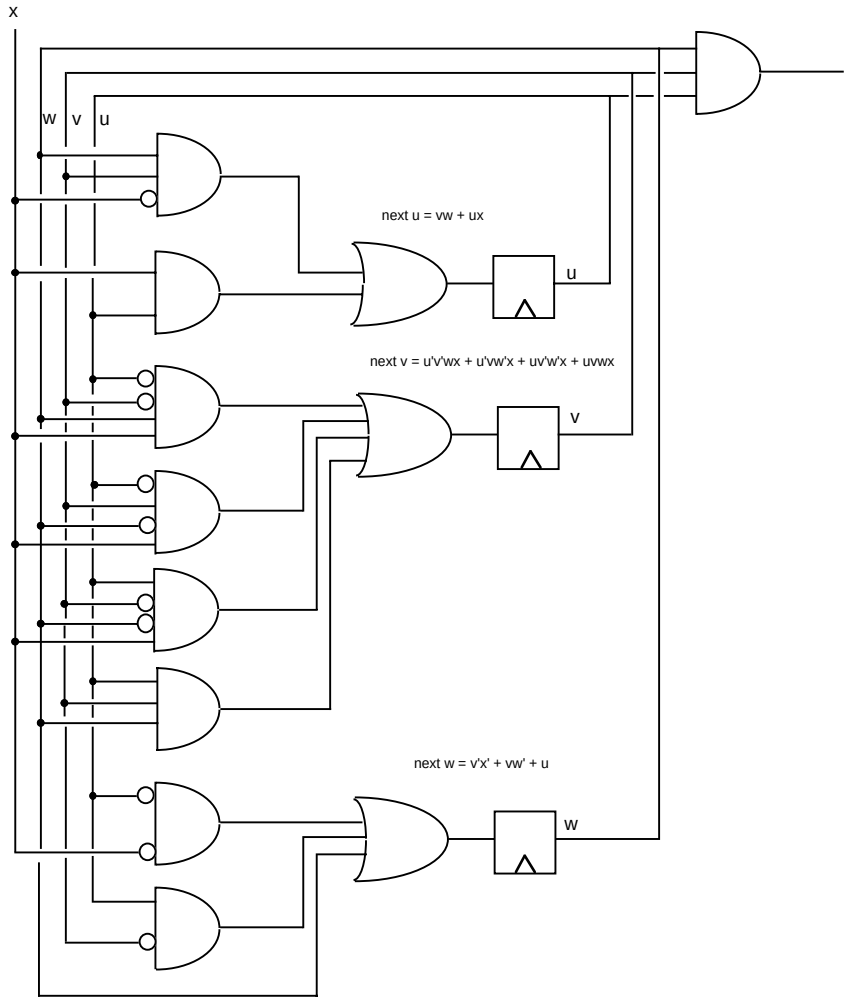


Figure 258: Implementation of a combination lock using flip-flops and gates

Example: Binary Adder with Output

The binary adder is an example of a transducer. The output value is 1 when the one or three of the two inputs and the carry are 1. When none or two of those values are 1, the output is 0. This functionality can be represented as a 3-input exclusive-OR gate, as shown in the figure, but this gate can also be further implemented using AND-, OR-, and NOT- gates as always. Typically the output would be used as input to a system in which this machine is embedded.

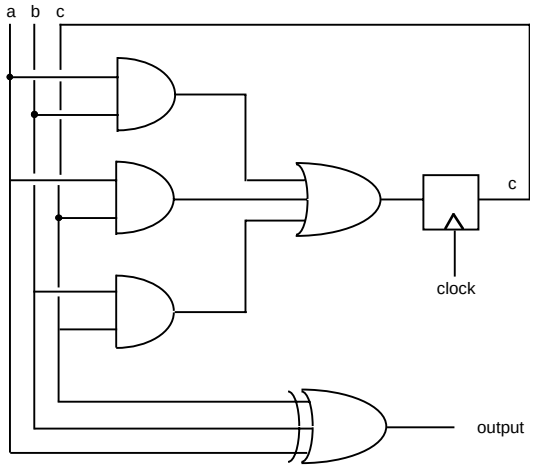


Figure 259: Implementation of a sequential binary adder

12.6 Inside Flip-Flops

In keeping with our desire to show a relatively complete vertical picture of computer structure, we briefly go into the construction of flip-flops themselves. Flip-flops can be constructed from combinational logic, assuming that such logic has a delay between input changes and output changes, as all physical devices do. Flip-flops can be constructed from a component that realizes the memory aspect, coupled with additional logic to handle clocking. The memory aspect alone is often referred to as a **latch** because it holds or "latches" the last value that was appropriately signaled to it.

To a first approximation, a latch can be constructed from two NOR gates, as shown below.

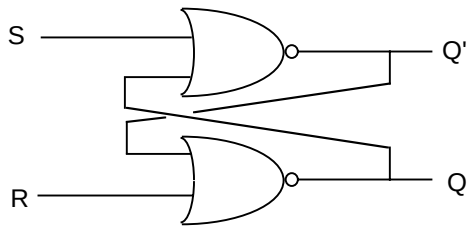


Figure 260: An electronic latch from opposing NOR-gates

In order for the latch to function properly, the inputs have to be controlled with a specific discipline; at this level, there is no clock to help us out. Let the state of the latch be represented by SRQQ'. Normally Q and Q' will be complementary, but there will be times at which they are not. Consider the state SRQQ' = 0010. Here we say the latch is "set". In the similar state 0001, the latch is "reset". The function of the inputs S and R is to put the latch into one of these two states. Specifically, when S is raised to 1, the latch should change to the set state, or stay in the set state if it was already there. Similarly, when R is raised to 1, the latch should change to the reset state. When the input that was raised is lowered again, the latch is supposed to stay in its current state.

We must first verify that the set and reset states are stable, i.e. not tending to change on their own. In 0010, the inputs to the top NOR gate are 01, making the output 0. This agrees with the value of Q' in 0010. Likewise, the inputs to the bottom NOR gate are 00, making the output 1. This agrees with the value of Q in 0010. Therefore 0010 is stable. Similarly, we can see that 0001 is also stable.

Now consider what happens if the latch is in 0010 (set) and R is raised. We then have state 0110. The upper NOR gate's output does not tend to change at this point. However, the lower NOR gate's output is driven toward 0, i.e. Q changes from 1 to 0. Following this, the upper NOR gate's output is driven toward 1, so Q' changes from 0 to 1. Now the latch is in state 0101. We can see this is stable. If R is now lowered, we have state 0001, which was already observed to be stable. In summary, raising R for sufficiently long, then lowering it, results in the reset state. Also, if the latch were in state 0001 when R is raised, then no change would take place and the latch would stay in state 0001 when R is lowered.

Similarly, we can see that raising S momentarily changes the latch to state 0010. So S and R are identified with the functions of setting and resetting the latch, respectively. Thus the latch is called a set-reset or SR latch. The following state diagram summarizes the behavior we have discussed, with stable states being circled and transient states left uncircled.

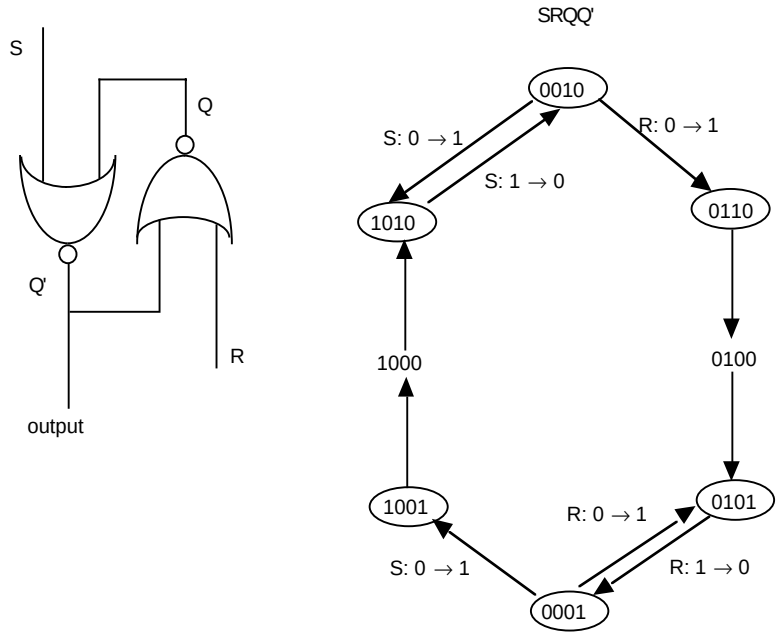


Figure 261: State-transition behavior of the electronic latch. States not outlined (1000, 0100) are unstable and tend to make autonomous transitions toward stable states as shown.

In describing the latch behavior, we dealt with the cases where only one of R or S is changing at a time. This constitutes a constraint under which the latch is assumed to operate. If this condition is not maintained, then the latch will not necessarily behave in a predictable fashion. We invite you to explore this in the exercises.

Next we show how a latch becomes a flip-flop by adding the clock element. To a first approximation, a flip-flop is a latch with some added gates so that one of S or R is only activated when the clock is raised. This approximation is shown below. However, we do not yet have a true flip-flop, but only a **clocked latch**, also called a **transparent latch**. The reason for this hedging is that if the input to the unit is changed while the clock is high, the latch will change. In contrast, in our assumptions about a D flip-flop, the flip-flop is supposed to only change depending on the value of the input at the leading edge of the clock, i.e. the flip-flop is supposed to be **edge-triggered**.

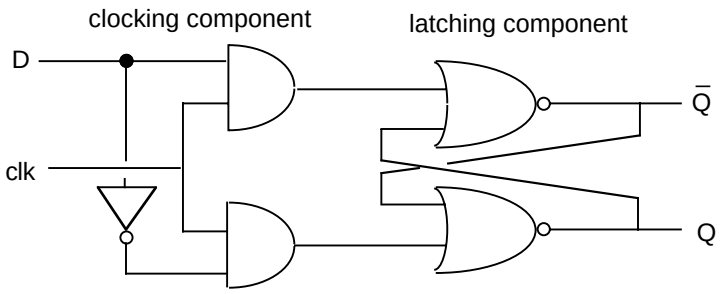


Figure 262: A "transparent" D latch

In order to get edge triggering, we need to latch the input at the time the clock rises, and then desensitize the latch to further changes while the clock is high. This is typically done using a circuit with a more complex clocking component, such as the one below. The assumption made here is that the D input is held constant long enough during the clock high for the flip-flops on the left-hand side to stabilize.

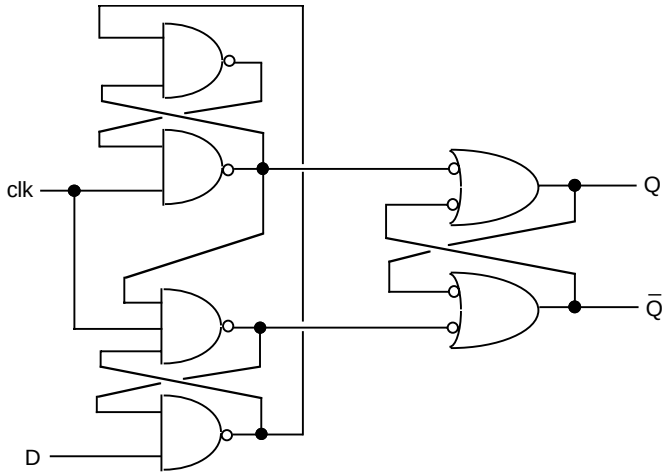


Figure 263: Edge-triggered D flip-flop using NAND gates

Exercises

1 • Explain in your own words why raising and lowering S changes the latch to 0010.

2 •• Explore the state transition behavior for states not shown in the diagram, in particular from the state 1100.

3 ••• By using a state diagram, verify that the edge-triggered D flip-flop is indeed edge-triggered.

12.7 The Progression to Computers

We have already seen hints of the relationship of finite-state machines to computers. For example, the control unit of a Turing machine looks very much like a finite-state transducer. In these notes, we use the "classifier" variety of finite-state machine to act as controllers for computers, introducing a type of design known as register-transfer level (RTL). From there, it is easy to explain the workings of a stored-program computer, which is the primary medium on which programs are commonly run.

Representing states of computers explicitly, as is called for in state diagrams and tables, yields state sets that are too large to be practically manageable. Not only is the number of states too big to fit in the memory of a computer that would be usable as a tool for analyzing such finite-state machines, it is also impossible for a human to understand the workings of a machine based on explicit representation of its states. We therefore turn to methods that combine finite-state machines with data operations of a higher level.

A Larger Sequential Adder

A machine that adds up, modulo 16, a sequence of numbers in the range 0 to 15, each represented in binary. Such a machine is depicted below.

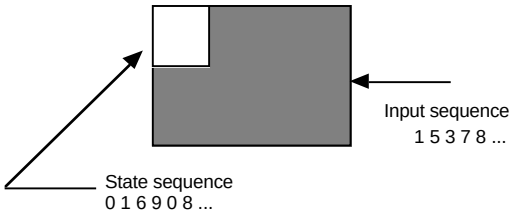


Figure 264: Sequence adding machine, modulo 16

We could show the state transition function for this machine explicitly. It would look like the following large addition table:

next state	input															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
current state	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
	2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
	3	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
	4	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
	5	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
	6	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
	7	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
	8	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
	9	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
	10	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
	11	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
	12	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
	13	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
	14	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	15	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If we chose a much larger modulus than 16 the state table would be correspondingly larger, growing as the square of the modulus. However, the basic principle would remain the same. We can show this principle using a diagram like the finite-state machine diagram:

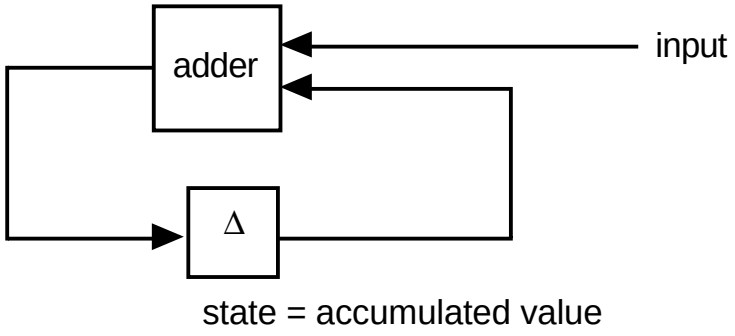


Figure 265: Diagram for the sequence adder.
The adder box adds two numbers together

As before, we can implement the adder in terms of combinational logic and the state in terms of a bank of D flip-flops. The combination logic in this is recognized as the adder module introduced in *Proposition Logic*.

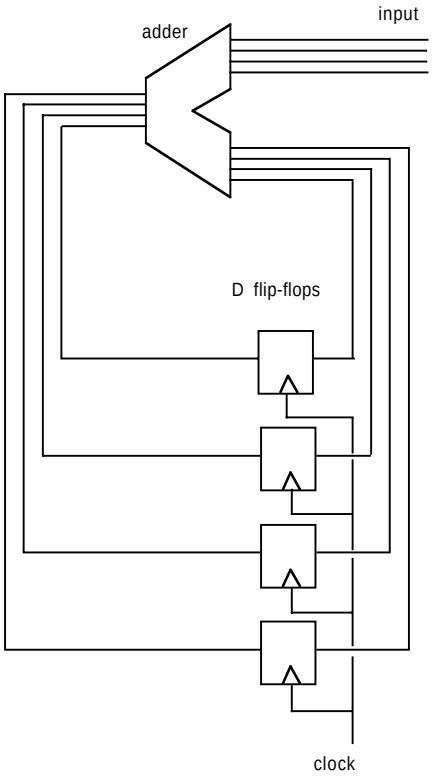


Figure 266: The sequence adder at the next level of detail

Registers

Now return to our diagram of the sequence adder. In computer design, it is common to group flip-flops holding an encoded value together and call them a **register**, as suggested by the following figure.

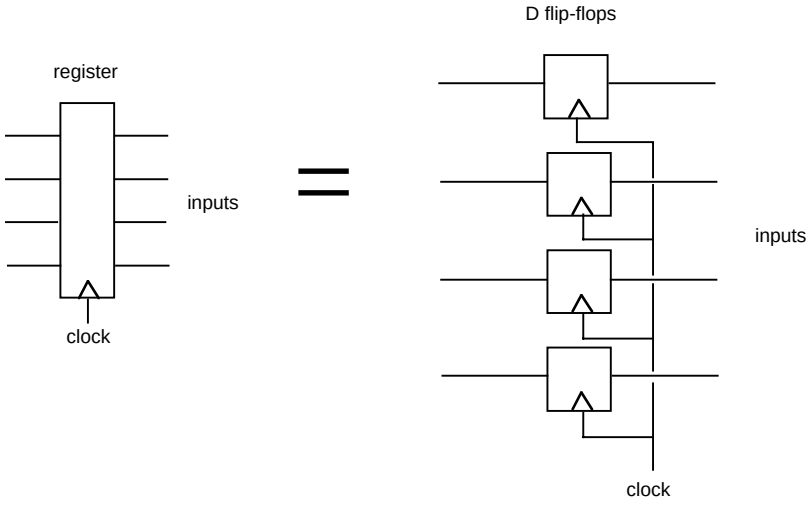


Figure 267: Expansion of a 4-bit register

Every time the clock ticks, the current logical values at the input side are stored into (or "gated into") the flip-flops. This shows a register of minimum functionality. Usually other functions are present. For example, we will often want to selectively gate information into the register. This can be accomplished by controlling whether or not the flip-flops "see" the clock tick. This can be done simply using an AND-gate. The control line is known as a "strobe": when the strobe is 1 and the clock ticks, the external values are gated into the register. When the strobe is 0, the register maintains its previous state.

Note that a register is essentially a (classifier) finite-state machine that just remembers its last data input. For example, suppose that we have a register constructed from two flip-flops. Then the state diagram for this machine is:

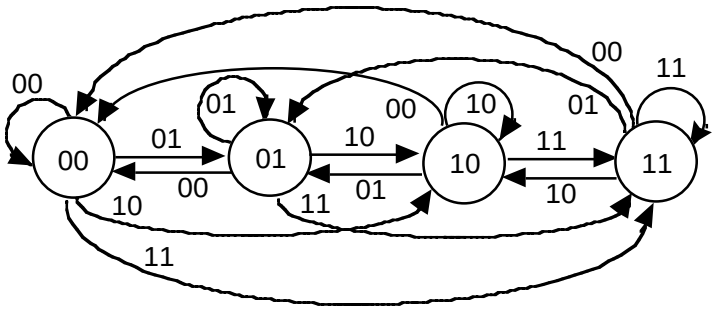


Figure 268: State-transition diagram for a 2-bit register.
Each input takes the machine to a state matching the input.

A typical use of this selective gating is in selective transfer from one register to another. The situation is shown below.

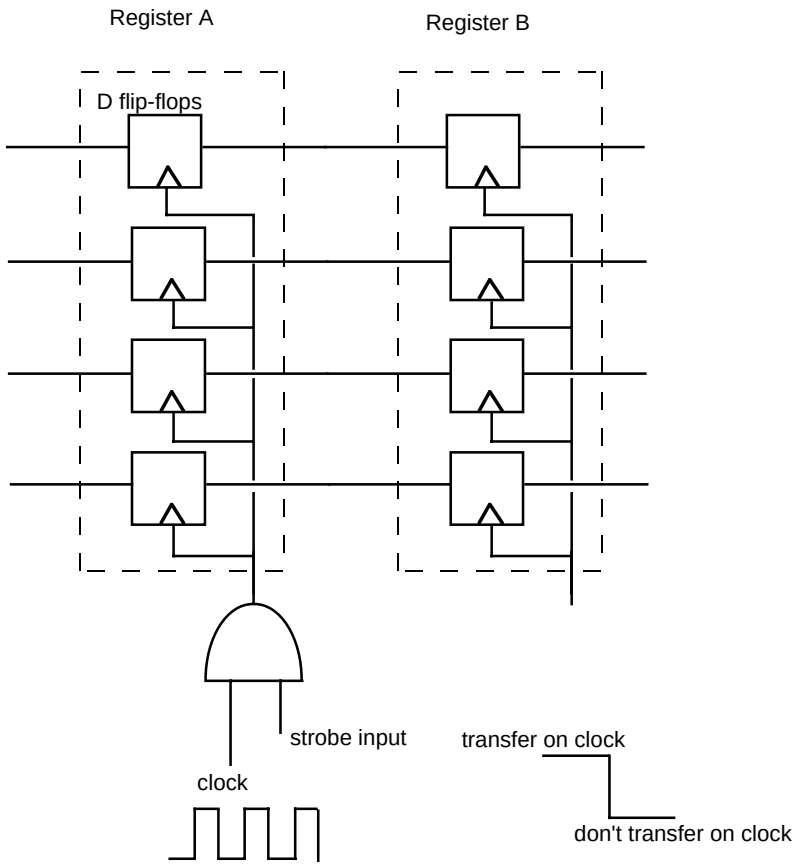


Figure 269: Transferring from one register to another using a strobe

In a similar fashion, the output of any combinational unit can be gated selectively into a register. Typically, the gate with the strobe is considered to be part of the register itself. In this case, the view of the register as a finite state-machine includes the strobe as one of the inputs. If the strobe value is 1, the transitions are as shown in the previous state diagram. If the strobe value is 0, the transitions are to the current state. For ultra-high-speed designs, strobing against the clock this way is not desirable, as it introduces an extra gate delay. It is possible to avoid this defect at the expense of a more complicated register design. We will not go into the details here.

Composition of Finite-State Machines

A computer processor, the kind you can buy, rather than an abstract computer like a Turing machine, is essentially a very large finite-state machine. In order to understand the behavior of such a machine, we must resort to a modular decomposition. We cannot hope to enumerate all the states of even a simple computer, even if we use all the resources in the universe.

There are several fundamental ways to compose finite-state machines. In each case, the overall machine has as its state set a subset of the Cartesian product of the state sets of the individual machines. Consider first the *parallel composition* of two machines, as shown below.

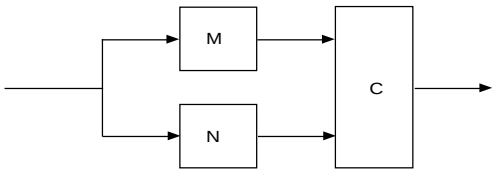


Figure 270: Parallel Composition of machines M and N

The two machines share a common input. They go through their transitions in "lock step" according to the clock. Unit C is combinational logic that combines the outputs produced by the machines but that does not have memory of its own.

To how the structure of this machine relates to the individual machines, let's show something interesting:

The intersection of two regular languages is regular.

Unlike the *union* of two regular languages, his statement does not follow directly from the definition of regular expressions. But we can show it using the parallel composition notion. Let M and N be the machines accepting the two languages in question. We will see how a parallel composition can be made to accept the intersection. What unit C does in this case is to form the logic product of the outputs of the machines, so that the overall machine accepts a string when, and only when, both component machines accept. This is, after all, the definition of intersection.

Example: Parallel Composition

Consider two acceptors used as examples earlier.

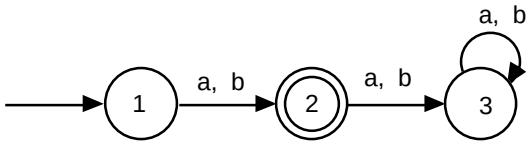


Figure 271: Acceptor for a | b

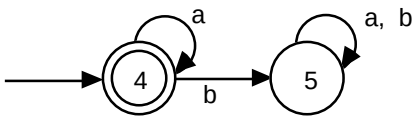


Figure 272: Acceptor for a*

To make an acceptor for the intersection of these two regular languages, we construct a machine that has as its state set the product {1, 2, 3} X {4, 5} = {(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)}. We might not actually use all of these states, as some might not be reachable from the initial state, which is (1, 4) (the pair of initial states of each machine). There is just one accepting state, (2, 4), since it is the only one in which *both* components are accepting.

The following transitions occur in the product machine:

	input	
state	a	b
(1, 4)	(2, 4)	(2, 5)
(2, 4)	(3, 4)	(3, 5)
(2, 5)	(3, 5)	(3, 5)
(3, 4)	(3, 4)	(3, 5)
(3, 5)	(3, 5)	(3, 5)

We see that one state in the product, namely (1, 5), is not reachable. This is because once we leave 1, we can never return. The diagram for the product machine is thus shown in the following figure.

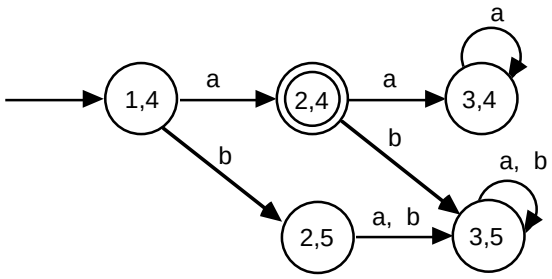


Figure 273: State diagram for the product of the preceding two acceptors

In a similar vein, we may construct other kinds of composition, such as ones in which one machine feeds the other, or in which there is cross-feeding or feedback. These are suggested by the following structural diagrams, but the experimentation with the state constructions is left to the reader.

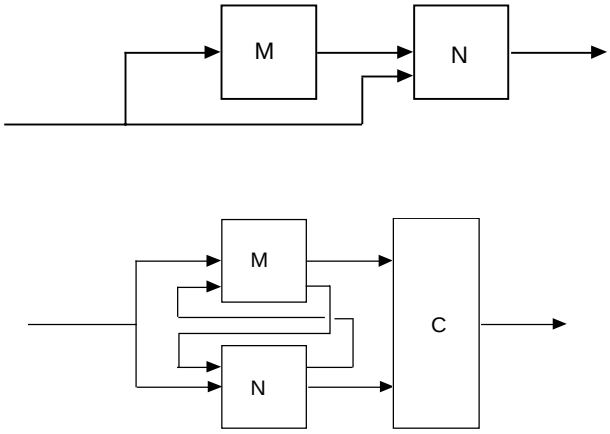


Figure 274: Examples of other machine compositions

Again, we claim that a computer is just one large composition of many smaller machines. A clear example of this will be seen in the next chapter.

Additional Capabilities of Registers

We saw in Part 1 how data can be transferred from one register to another as an array of bits in one clock tick. This is known as a **parallel transfer**. Another way to get data to or from a register is via a **serial transfer**, i.e. one bit at a time. Typically this is done by having the bits gated into one flip-flop and **shifting** the bits from one flip-flop to the next. A register with this capability is called a **shift register**. The following diagram shows how the shift register functionality can be implemented.

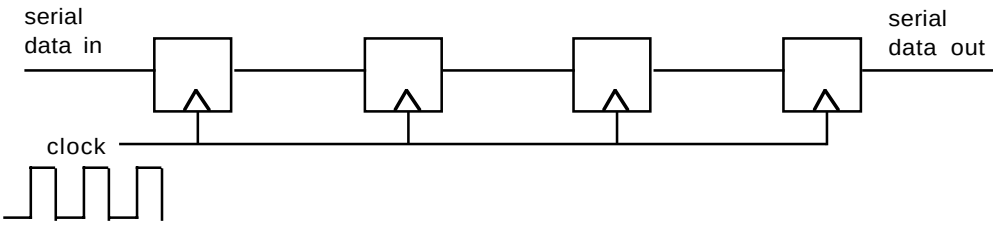


Figure 275: Shift register constructed from D flip-flops

Once the data have been shifted in serially, they can be transferred out in parallel. Also, data could be transferred in parallel and transferred out in serial. Thus the shift register can serve as a serial-parallel converter in both directions.

A shift register is an implementation of finite-state machine, in the same model discussed earlier, i.e. a bank of flip-flops serve as memory and a set of combinational functions compute the next-state function. The functions in this case are trivial: they just copy the

value from one flip-flop to the next. For the 4-flip-flop machine shown above, the transition table would be:

next state		**input**	
	current state	**0**	**1**
	0 0 0 _	0 0 0 0	1 0 0 0
	0 0 1 _	0 0 0 1	1 0 0 1
	0 1 0 _	0 0 1 0	1 0 1 0
	0 1 1 _	0 0 1 1	1 0 1 1
	1 0 0 _	0 1 0 0	1 1 0 0
	1 0 1 _	0 1 0 1	1 1 0 1
	1 1 0 _	0 1 1 0	1 1 1 0
	1 1 1 _	0 1 1 1	1 1 1 1

Here the _ indicates that we have the same next state whether the _ is a 0 or 1. This is because the right end bit gets "shifted off".

In order to combine functionalities of shifting and parallel input to a register, additional combinational logic has to be used so that each flip-flop's input can select either function. This can accomplished by the simple combinational circuit known as a **multiplexor**, as introduced in the Proposition Logic chapter. By using such multiplexors to select between a parallel input and the adjacent flip-flop's output, we can achieve a two-function register. The address line of each multiplexor is tied to a control line (which could be called a "strobe") that specifies the function of the register at a given clock.

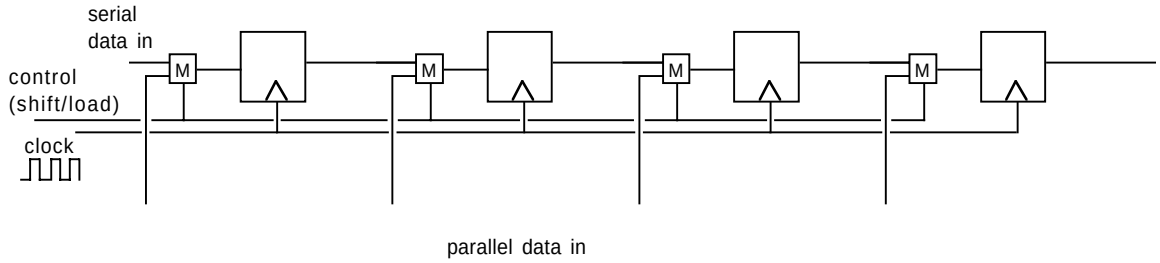


Figure 276: Structure of a shift register with two functions: serial data in and parallel data in

A commercial shift-register is typically constructed from a different type of flip-flop than the D, to simplify the attendant multiplexing logic. Thus the above diagram should be regarded as being for conceptual purposes. Multiplexors (also called **MUX**es) are more often found used in other applications, as will be discussed subsequently.

Buses and Multiplexing

Quite often in computer design we have the need to selectively transfer into a register from one of several other registers. One way to accomplish this is to put a multiplexor with one input per register from which we wish to transfer with the output connected to the register to which we wish to transfer. The address lines can then select one of the input registers. As before, the actual transfer takes place on a clock tick.

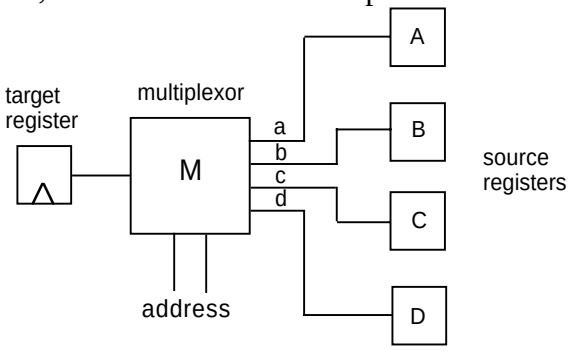


Figure 277: Selective transfer using a multiplexor (one bit shown)

This approach can be both expensive and slow (due to multiple levels of delay in the multiplexor) if the number of source registers is appreciable. An alternate approach is to use a **bus structure** to achieve the multiplexing. To a first approximation, a bus just consists of a single wire per bit. The understanding is that at most one source register will be providing input to the bus at any one time. Hence the bus at that time provides a direct connection to the target register. The problem is how to achieve this effect. We cannot use logic gates to connect the sources to the bus, since by definition these gates always have their output at either 0 or 1. If one gate output is 0 and the other is 1 and they are connected to a common bus, the result will be undefined logically (but the effect is similar to a short-circuit and could cause damage to the components).

A rather miraculous device known as a **three-state buffer** is used to achieve the bus interconnection. As its name suggests, its output can be either 0, 1, or a special third state known as "high impedance" or "unconnected". The effect of the third state is that the output line of the 3-state buffer is effectively not connected to the bus logically. Nonetheless, which of the three states the device is in is controlled electronically by two logical inputs: one input determines whether the output is in the unconnected state or not, and the other input has the value that is transmitted to the output when the device is not in the connected state. Therefore the following function table represents the behavior of the 3-state buffer:

control	data	output
0	0	unconnected
0	1	unconnected
1	0	0
1	1	1

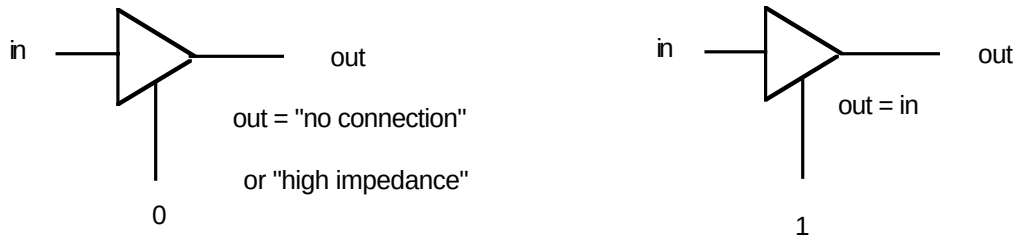


Figure 278: Three-state buffer behavior

The following figure illustrates how multiple source registers can be multiplexed using a bus and one 3-state buffer per source. Notice that the selection is done by one 3-state device being activated at a time, i.e. a "one-hot" encoding, in contrast to the binary encoding we used with a multiplexor. Often we have the control strobe in the form of a one-hot encoding anyway, but if not, we could always use a decoder to achieve this effect. Of course, if there are multiple bits in the register, we must have multiple bus wires and one 3-state buffer per bit per register.

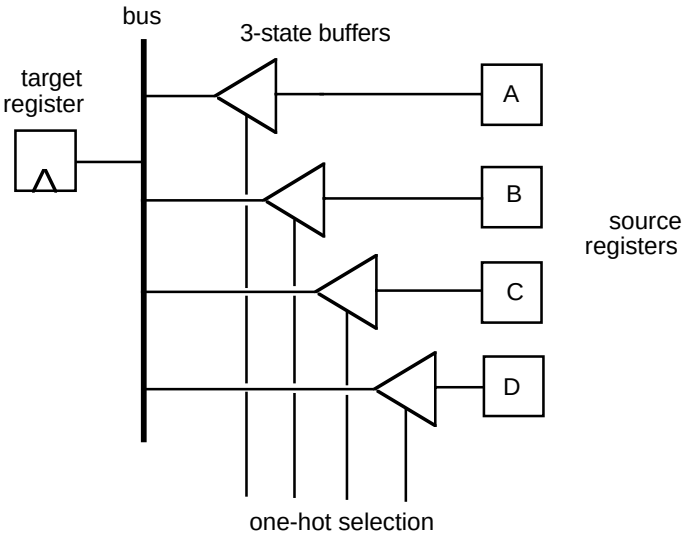


Figure 279: Multiplexing register sources using a bus and 3-state buffers

If there are multiple targets as well as sources, then we can control the inputs to the targets by selectively enabling the clock input to those registers. One contrast to the multiple-source case, however, is that we can "broadcast" the same input to multiple targets in a single clock tick. Put another way, the selection of the target register(s) can be done by a subset encoding rather than a one-hot encoding. This is shown in the next figure.

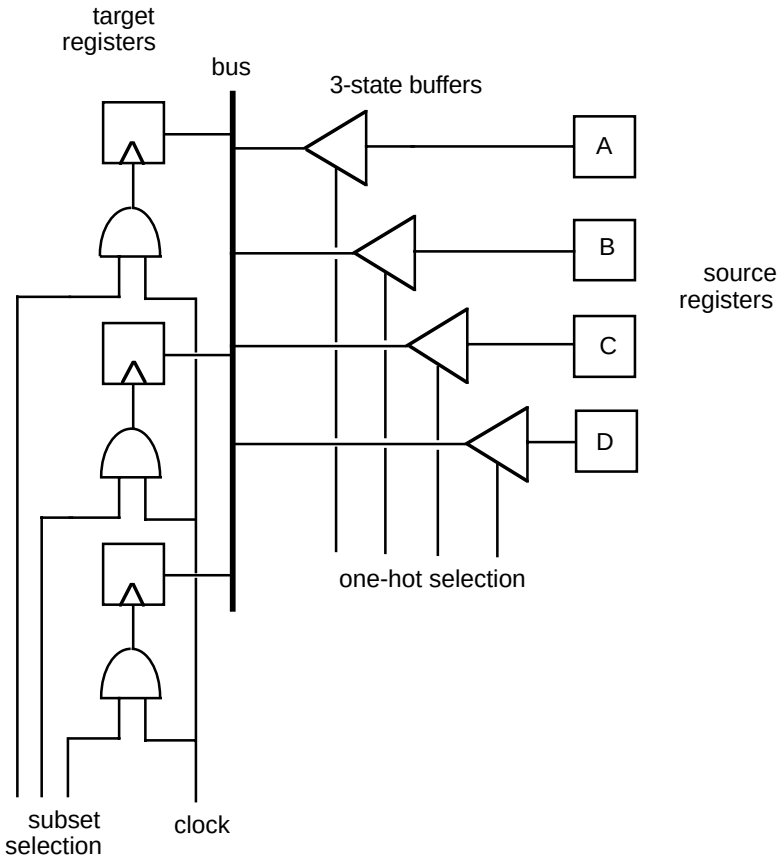


Figure 280: Multiple source and target selections on a bus

The other point to note about a bus is that inputs need not come from a register; they can come from the output of any combinational logic, such as an adder.

At a sufficiently coarse level of abstraction, buses are shown as a single data path, suppressing the details of 3-state devices, selection gates, etc. An example is the diagram of the ISC (Incredibly Simple Computer) in the next chapter.

Exercises

1 •• Show how to construct a 2-function shift-register that has the functions shift-left and shift-right.

2 ••• Show how to construct a 5-function shift-register, with functions shift-left, shift-right, parallel load, clear (set all bits to 0), and no-operation (all bits left as is).

3 ••• While the shift register described is convenient for converting serial data to parallel data, it is too slow to serve as an implementation of the shift instructions found in most computers, where it is desired to shift any number of bits in one clock interval. A combinational device can be designed that accomplishes this type of shift based on using the binary representation of the amount to be shifted

to shift in log N stages, shifting successive powers of two positions at each stage. Sketch the design of such a device based on 2-input multiplexors. In the trade, this is known as a "barrel shifter". Earlier, we referred to the principle on which the barrel shifter is based as **the radix principle**.

12.8 Chapter Review

Define the following terms:

acceptor
bus
classifier
clock
D flip-flop
edge-triggered
feedback system
finite-state machine
flip-flop
Kleene's theorem
latch
multiplexor
non-deterministic finite-state machine
parallel composition
quantization
regular expression
register
sequencer
shift register
stable state
synchronous
three-state buffer
transducer
Turing machine

Demonstrate how to convert a non-deterministic finite-state acceptor to a deterministic one.

Demonstrate how to derive a non-deterministic finite-state acceptor from a regular expression.

Demonstrate how to derive a regular expression from a finite-state machine.

Demonstrate how to synthesize a switching circuit from a finite-state machine specification.

12.9 Further Reading

Frederick C. Hennie, *Finite-State Models for Logical Machines*, Wiley, New York, 1968. [Further examples of finite-state machines and regular expressions. Moderate.]

S.C. Kleene, Representation of events in nerve nets and finite automata, pp 3-41 in Shannon and McCarthy (eds.), *Automata Studies*, Annals of Mathematics Studies, Number 34, Princeton University Press, 1956. [The original presentation of regular expressions and their connection with machines. Moderate.]

G.H. Mealy, *A method for synthesizing sequential circuits*, The Bell System Technical Journal, 34, 5, pp. 1045-1079, September 1955. [Introduces the Mealy model of finite-state machine. Moderate.]

Edward F. Moore, *Gedanken-experiments on sequential machines*, pp 129-153 in Shannon and McCarthy (eds.), *Automata Studies*, Annals of Mathematics Studies, Number 34, Princeton University Press, 1956. [Introduces the Moore model of finite-state machine. Moderate to difficult.]

13. Stored-Program Computers

13.1 Introduction

This chapter concentrates on the low-level usage and structure of stored program computers. We focus on a particular hypothetical machine known as the ISC, describing its programming in assembly language. We show how recursion and switch statements are compiled into machine language, and how memory-mapped overlapped I/O is achieved. We also show the logic implement of the ISC, in terms of registers, buses, and finite-state machine controllers.

13.2 Programmer's Abstraction for a Stored-Program Computer

By stored-program computer, we mean a machine in which the program, as well as the data, are stored in memory, each word of which can be accessed in uniform time. Most of the high-level language programming the reader has done will likely have used this kind of computer implicitly. However, the program that is stored is not high-level language *text*. If it were, then it would be necessary to constantly parse this text, which would slow down execution immensely. Instead one of two other forms of storage is used: An abstract syntax representation of the program could be stored. The identifiers in this representation are pre-translated, and the structure is traversed dynamically as needed during execution. This is the approach used by an ***interpreter*** for the language. A second approach is to use a ***compiler*** for the language. The compiler translates the program into the very low-level language native to the machine, appropriately called ***machine language***. The native machine language acts as a least-common-denominator language for the computer. A machine that had to understand, at a native level, many different languages would be prohibitively complex and slow. Machine language is rarely programmed directly, since that would involve manipulating bits, with which it is easy to err. Instead, an equivalent symbolic form known as ***assembly language*** is employed.

In this chapter, we will build up a stored-program computer using our knowledge of finite-state machine components described earlier. But first, we describe the native language of a simple computer using assembly language. Then we "build-down" from higher-level language constructs to the assembly language to see how various algorithmic concepts get translated.

In the mid-1980's, a major paradigm shift began, from CISCs (Complex Instruction Set Computers) to RISCs (Reduced Instruction Set Computers). RISCs tried to take a "lean and mean" approach, in contrast to their predecessor CISCs, which were becoming bloated with complexity. RISCs focused on features related to speed and simplicity and consciously avoided including the "kitchen sink" in the instruction repertoire. The

machine we use here for illustration is called the ISC, for Incredibly Simple Computer. It is of the RISC philosophy, but simpler than most RISCs for tutorial purposes.

The following is a terse description of the ISC. The unit of addressability of the ISC is one 32-bit word. The ISC has a 32-bit address space. This means that up to 2^{32} different words can be addressed in the memory, in principle, although a given implementation will usually contain far fewer words. Memory words are addressed by a signed integer, and negative addresses are typically used for "memory-mapped I/O", as described later. Instructions in the ISC are all one word long. Both instructions and data are stored in the memory of the computer. The instructions get into the memory by the execution of a special program known as the **loader**, which takes the output of the compiler and loads it into memory. A special part of memory known as the read-only memory (ROM) contains a primitive loader that brings in other programs from a cold-start.

Although the instructions operate on data stored in the memory, ISC instructions do not reference memory locations directly. Instead, the data in memory are brought into **registers** and the instructions specify operation on the registers. The registers also serve to hold addresses designating the locations in memory to and from which data fetching and storage occurs.

Internal to the ISC processor, but accessible by the programmer, are 32 registers, numbered 0-31. All processor state is contained in the registers and the instruction pointer (IP) (equivalent to what is sometimes called "program counter" (PC), unfortunately not a thing that counts programs). The IP contains the address of the next instruction to be executed.

The following kinds of addressing are used within ISC:

> **Register-indirect addressing** is used in all operations involving addressing, including the **jump** operations, **load**, and **store**. In other words, the memory address is contained in a register (put there earlier by the program itself) and the instruction refers to the register that contains the address.
>
> **Immediate values** are used in the **lim** and **aim** operations. The term "immediate" means that the datum comes immediately from the instruction itself, rather than from a register or memory.

In the following table, Ra, Rb, and Rc stand for register indices. The Java language equivalent is given, followed by a brief English description of the action of each instruction. In the cases of the arithmetic instructions (add, sub, mul, div), if the result does not fit into 32 bits, only the lower-order 32 bits are stored.

lim Ra C **reg[Ra] = C**
Load immediate to register *Ra* the signed 24-bit integer (or address) constant C.

aim Ra C **reg[Ra] += C**
Add immediate to register *Ra* the signed 24-bit integer (or address) constant C.

load Ra Rb **reg[Ra] = mem[reg[Rb]]**
Load into *Ra* the contents of the memory location addressed by *Rb*.

store Ra Rb **mem[reg[Ra]] = reg[Rb]**
Store into the memory location addressed by *Ra* the contents of *Rb*.

copy Ra Rb **reg[Ra] = reg[Rb]**
Copy into *Ra* the contents of register *Rb*.

add Ra Rb Rc **reg[Ra] = reg[Rb] + reg[Rc]**
Put into *Ra* the **sum** of the contents of *Rb* and the contents of *Rc*.

sub Ra Rb Rc **reg[Ra] = reg[Rb] - reg[Rc]**
Put into *Ra* the contents of *Rb* **minus** the contents of *Rc*.

mul Ra Rb Rc **reg[Ra] = reg[Rb] * reg[Rc]**
Put into *Ra* the **product** the contents of *Rb* and the contents of *Rc*.

div Ra Rb Rc **reg[Ra] = reg[Rb] / reg[Rc]**
Put into *Ra* the contents of *Rb* **divided** by the contents of *Rc*.

and Ra Rb Rc **reg[Ra] = reg[Rb] & reg[Rc]**
Put into *Ra* the contents of *Rb* **bitwise-and** the contents of *Rc*.

or Ra Rb Rc **reg[Ra] = reg[Rb] | reg[Rc]**
Put into *Ra* the contents of *Rb* **bitwise-or** the contents of *Rc*.

comp Ra Rb **reg[Ra] = ~reg[Rb]**
Put into *Ra* the **bitwise-complement** of the contents of *Rb*.

shr Ra Rb Rc **reg[Ra] = reg[Rb] >> reg[Rc]**
The contents of *Rb* is **shifted right** by the amount specified in register *Rc* and the result is stored in *Ra*. If the value in *Rc* is negative, the value is shifted left by the negative of that amount.

shl Ra Rb Rc **reg[Ra] = reg[Rb] << reg[Rc]**

The value in register *Rb* is **shifted left** by the amount specified in register *Rc* and the result is stored in *Ra*. If the value in *Rc* is negative, the value is shifted right by the negative of that amount.

jeq Ra Rb Rc **Jump** to the address in *Ra* if the values in *Rb* and *Rc* are **equal**. Otherwise continue.

jne Ra Rb Rc **Jump** to the address in *Ra* if the values in *Rb* and *Rc* are **not equal**. Otherwise continue.

jgt Ra Rb Rc **Jump** to the address in *Ra* if the value in *Rb* is **greater than** that in *Rc*. Otherwise continue.

jgte Ra Rb Rc **Jump** to the address in *Ra* if the value in *Rb* is **greater than or equal** that in *Rc*. Otherwise continue.

jlt Ra Rb Rc **Jump** to the address in *Ra* if the value in *Rb* is **less than** that in *Rc*. Otherwise continue.

jlte Ra Rb Rc **Jump** to the address in *Ra* if the value in *Rb* is **less than or equal** that in *Rc*. Otherwise continue.

junc Ra **Jump** to the address in *Ra* **unconditionally**.

jsub Ra Rb **Jump to subroutine** in the address in *Ra*. The value of the *IP* (i.e. what would have been the next instruction) is put into *Rb*. Therefore this can be used for jumping to a subroutine. If the return address is not needed, some register not in use should be specified.

Although it is not critical for the current explanation, the following shows a plausible formatting of the ISC instructions into 32-bit words, showing possible assignment of op-code bits. Each register field uses five bits. It appears that there is wasted space in many of the instructions. There are techniques, such as having instructions of different lengths, for dealing with this. We are using the present form for simplicity.

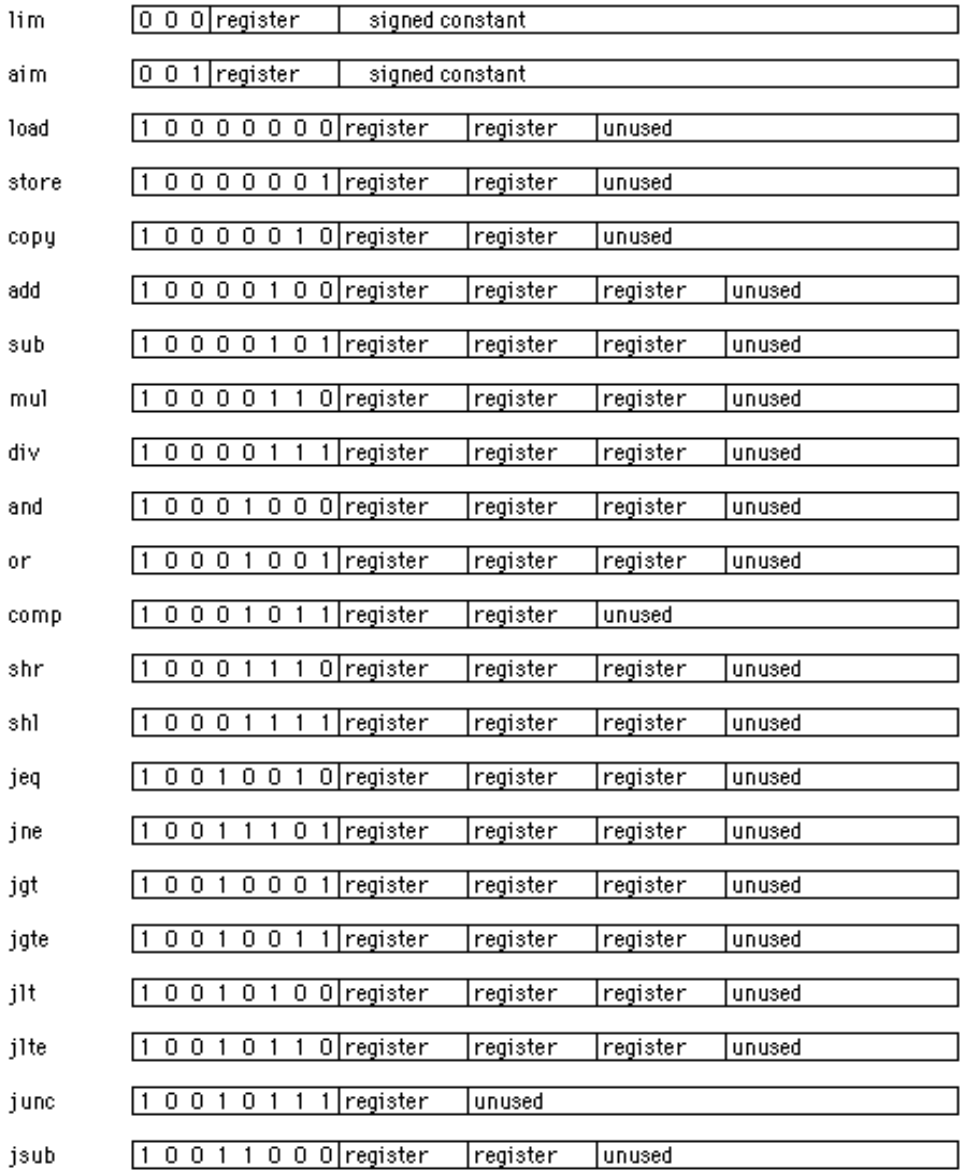


Figure 281: Plausible ISC instruction formatting

13.3 Examples of Machine-Level Programs

Example Add the values in registers 0, 1, and 2 and put the result into register 3:

```
add 3 0  1  // register 3 gets sum of registers 0 and 1
add 3 2  3  // register 3 gets sum of registers 2 and 3
```

Here we use register 3 to hold a temporary value, which is used as an operand in the second instruction.

Example Suppose x is stored in register 0, and y in register 1. Compute the value of (x + y) * (x - y) and put it in register 3. Assume register 4 is available for use, if needed.

```
add 3 0 1   // register 3 gets x + y
sub 4 0 1   // register 4 gets x - y
mul 3 3 4   // register 3 gets (x + y)(x - y)
```

Example Add the contents of memory locations 1000 and 1001 and put the result into 1002. Assume registers 0 and 1 are available.

```
lim 0 1000  // get addresses of operands into registers 0
lim 1 1001  //   and 1
load 0 0    // overlay addresses with operands
load 1 1
add 1 0 1   // put sum  in register 1
lim  0 1002 // re-use register 0 for address of result
store 0 1   // store the value in register 1 into 1002
```

Example Assume that register 0 contains the address of the first location of an array in memory and register 1 contains the number of locations in the array. Add up the locations and leave the result in register 2. Assume that registers 0 and 1 can be changed in the process and that registers 3 through 8 can be used for temporaries. Assume that the program starts in location 0.

```
lim 2 0     // initialize sum
lim 3 0     // comparison value
lim 6 10    // address of instruction following this code
lim 7 4     // address of next instruction
jlte 6 1 3  // jump to location 10 if the count is <= 0
load 5 0    // load register 5 from the next memory location
add 2 5 2   // add the next number to the sum
aim 0 1     // add 1 to the array address
aim 1 -1    // add -1 to the count
junc 7      // go back to location 4 and compare
```

Note that location 10 is the next location following this program fragment. This was determined from our assumption that the first instruction is in location 0 and instructions are one word long each. Similarly, the jump unconditionally back to 4 (the address in register 7) is for the next iteration of the loop.

Exercises

1 •• Show how the following could be evaluated using ISC machine language:

The sum of the squares of four numbers in registers.

The sum of the squares of numbers in an array.

2 • Show how an *xor* (exclusive-OR) instruction could be added to the ISC.

3 •• Show how a *mim* (multiply-immediate) instruction could be added to the ISC.

4 •• Show how a *jim* (jump-immediate) instruction could be added to the ISC.

Assembly Language

A reader who has worked through a simple example such as the above will no doubt immediately realize a need to invent a symbolic notation within which to construct programs. When constructing the preceding example program, at the third instruction, we did not know initially to put the 10 into `lim 6 10`, since we did not know where the next instruction following would be. Instead, we put in a **symbol**, say xx, to be resolved later. Once all the instructions were in place, we counted to find that the value of xx should be 10. This kind of record keeping becomes tedious with even modest size programs. For this reason, a computer program called an **assembler** is usually used to do this work for us. In an assembler, we can use symbolic values that either we equate to actual values or, as in the case of the address 10 above, the assembler will equate automatically for us. The assembler, not the programmer, does the counting of locations. This eliminates many possible errors in counting and is of exceptional benefit if the program needs to be changed. In the latter case, we would have to go back and track down any uses of addresses. We call the assembly language for the ISC **ISCAL** (ISC Assembly Language). The previous program in ISCAL might appear as:

```
      lim 2 0           // initialize sum
      lim 3 0           // comparison value
      lim 6 done_loc    // address of instruction following this code
      lim 7 loop_loc    // address of next instruction
label loop_loc          // implicitly define label 'loop'
      jlte 6 1 3        // jump to location 10 if the count <= 0
      load 5 0          // load register 5 from the next location
      add 2 5 2         // add the next number to the sum
      aim 0 1           // add 1 to the array address
      aim 1 -1          // add -1 to the count
      junc 7            // go back and compare
label done_loc          // implicitly define label 'done'
```

The readability of the code is also considerably improved through the use of **mnemonic labels** in place of absolute addresses. Note that, in contrast to the other instructions, the lines beginning with label are not executable instructions, but rather merely **directives** that define the labels *loop_loc* and *done_loc*. The general term for such directives in the jargon is **pseudo-op**, for "pseudo-operation". The label pseudo-op equates the identifier following the label to the address of the next instruction. This allows us to use that label as an address and load a register with, in preparation for jumping to that instruction.

Other pseudo-ops of immediate interest in ISCAL are:

origin Location — Indicates that the following code is to be loaded into successive locations starting at Location.

define Identifier Value — Causes the assembly-time value of Identifier to be equated to the integer value given.

We can take the idea of symbolic names a step further by allowing symbolic names for *registers* in place of the absolute register names. Let us agree to call the registers by the following names in this example:

0	array_loc
1	count
2	sum
3	zero (for comparing against)
5	value (one of the array elements)
6	done
7	loop

One way to equate the symbolic names to the register numbers is through the use of the **register** pseudo-op. Using this pseudo-op, the code would then appear as:

```
      register array_loc 0
      register count 1
      register sum 2
      register zero 3
      register value 5
      register done 6
      register loop 7
      ...
      lim sum  0                    // initialize sum
      lim zero  0                   // comparison value
      lim done done_loc             // address of instruction following
      lim loop loop_loc             // address of next instruction
label loop_loc
      jlte done count zero          // jump if <= 0
      load value array_loc          // load register next array value
      add sum value sum             // add the next number to the sum
      aim array_loc 1               // add 1 to the array address
      aim count -1                  // add -1 to the count
```

```
      junc loop                    // go back and compare
label done_loc
```

Note that *array_loc* is assumed to be initialized before we get to the executable code, e.g. this takes place somewhere within "..." . In order to use a jump instruction, we would normally expect to see a preceding *lim* instruction that loads an address into a jump target register. Above, both done and loop are used as jump target registers. Note that the *lim* instruction need not be immediately before the jump, although it often is. In the case of loops, for example, the target is sometimes left in its own register that is only loaded once, at the beginning of the loop sequence.

In the code above, the computation, for the most part, did not depend on specific registers being used. To avoid manually assigning register indices to registers when it doesn't matter, the ISC assembler provides another pseudo-op to automatically manage register indices. This is the **use** pseudo-op. When the assembler encounters the use pseudo-op, it attempts to allocate a free register of its choice to the identifier. Registers that have not been identified in register pseudo-ops, or in previous use pseudo-ops, are assumed to be free for this purpose. Furthermore, a register, once used, can be released by naming it in the **release** pseudo-op. Keep in mind that *use* and *release* are not executable instructions. They are interpreted in a purely textual fashion when the assembler input is scanned.

Let's rewrite the preceding code using *use* and *release*. We will assume that *array_loc*, *count*, and *sum* are to be kept as fixed registers, since they must be used to communicate with other code, i.e. they are not arbitrary.

```
register array 0
register count 1
register sum 2

use loop
use zero                        // register to hold zero
use value
use done
    lim sum  0                  // initialize sum
    lim zero 0                  // comparison value
    lim done done_loc           // address of instruction following
    lim loop loop_loc           // address of next instruction
label loop_loc
    jlte done count zero        // jump if <= 0
    load value array            // load register next array value
    add sum value sum           // add the next number to the sum
    aim array 1                 // add 1 to the array address
    aim count -1                // add -1 to the count
    junc loop                   // go back and compare
label done_loc
```

Procedures and Calling Conventions

It is common to have specific calling conventions with respect to registers used for procedure entry and exit. This helps standardize the compilation process. An example might be:

Use register 0 for the return address.
Use register 1 for the returned result.
Use register 2 for the first argument.
Use register 3 for the second argument.
....

up to some convened number of arguments. A procedure having more than this number of arguments would transfer the remaining ones through some sort of memory structure. The registers beyond this number are assumed to be available for internal use within the procedure.

Here is an example of calling a factorial procedure using this convention. There is only one argument.

```
// register definitions

register return 0       // standard return address reg
register result 1       // standard result register
register arg1   2       // first argument register

...

// calling sequence
// get argument in arg1

lim jump_target fac
jsub jump_target return

// use result from result

// procedure definition

label fac     // iterative factorial routine
              // initializes counter 'count' with argument value 'arg'
              // initializes an accumulator with value 1
              // repeats as long as counter greater than 0
              //    multiply accumulator by counter
              //    decrement counter
use zero
      lim zero 0
      lim result 1              // seed result with 1
      lim jump_target test      // set up for loop
label test
      jlte return arg1 zero     // return if arg is 0 or less
      mul  result result arg1   // multiply acc value by counter
      aim arg1 -1               // subtract 1 from the down counter
      junc jump_target          // jump back to the test
```

If such a convention is to be observed, then additional care must be taken when nested procedure calls are present. For example, if a main program calls procedure A, the return address to the point in main is put in register *return*. If A calls B, then the return address to the point in A is put into *return*. Thus, before calling B, A should save the contents of *return* somewhere, e.g. another register or a special location in memory. Following the return from B, and before returning, A should either return to the alternate register or restore *return* to what it was before B was called.

The following code demonstrates return address saving in a procedure that calls `fac` twice: given argument x, it computes `fac(fac(x))`. [Note: "nested refers here to `fac_fac` calling `fac`, not to the nesting `fac(fac(x))`.]

```
label fac_fac                 // calls fac(fac(arg))
use return2                   // return2 avoids clobbering return reg
      copy return2 return     // save original return
      lim jump_target fac     // call fac the first time (original arg)
      jsub jump_target return

      copy arg1 result        // copy result to argument register

      lim jump_target fac
      jsub jump_target return // call fac on the result

      junc return2
```

In the example above, we had no need to save the original argument of `fac_fac`. However, in some cases, we will need to use the original argument again after making the inner call. In this event, the *argument* too must be saved, much in the same manner as the return address.

Recursive Procedures in Machine Language

When a procedure is recursive, the technique described above has to be extended. There is generally no a priori limit on the number of levels of nesting. Thus no fixed number of registers nor special memory locations will suffice to store the return addresses and arguments. In this case, we must use some form of stack. There are two ways in which a stack could be used: The argument and return address could be put on the stack by the caller, or they could be put there by the callee, when and if it makes a nested call. In the following code, we use the latter method: data are not stacked unless a nested call is made. In either case, the stack itself must be set up beforehand. Once we are in the procedure, it is too late, as the procedure assumes the stack is present if needed.

A stack here will be implemented simply as an array in some otherwise unused area of memory. The code below does not check for stack overflow. Adding appropriate code for this is left as an exercise.

```
// set up stack
      lim stack_pointer save_area_loc  // initialize stack pointer
      aim stack_pointer     -1         // always point to top of stack
…

label fac     // recursive factorial routine

      lim result 1               // basis is 1
      jlte return arg zero       // return if count is 0 or less

      aim  stack_pointer +1      // increment stack pointer
      store stack_pointer return // save return address on stack

      aim  stack_pointer +1      // increment stack pointer
      store stack_pointer arg    // save argument on stack

      aim arg -1                 // subtract 1 from argument

      jsub jump_target return    // call recursively

      load arg stack_pointer     // restore original arg
      aim  stack_pointer -1

      load return stack_pointer  // restore original return address
      aim  stack_pointer -1

      mul  result result arg     // multiply by original arg

      junc return                // return to caller

…

label save_area_loc              // first location in save area
```

There are many ways to optimize the code above. But the purpose of the code is to exemplify recursive calling, not to give the best way to compute factorial.

The following diagram shows the stack growth in the case of calling fac with argument 4.

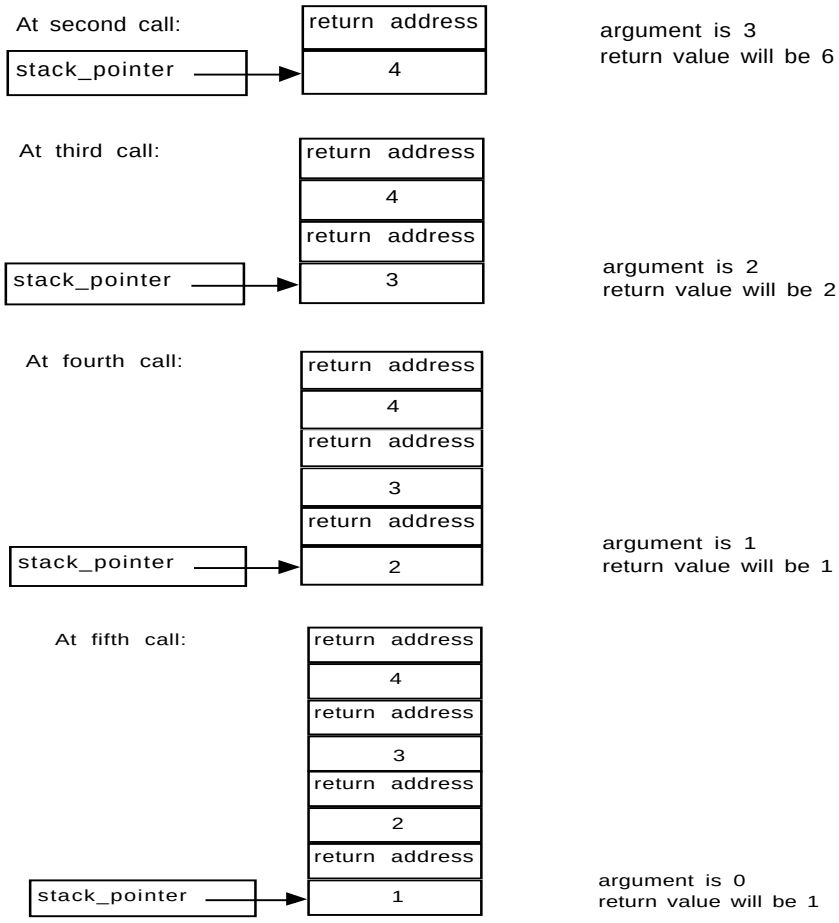


Figure 282: Snapshots of the stack in computing recursive factorial on the ISC

Exercises

1 •• Implement the recursive version of the Fibonacci function in ISCAL. Note: Unlike the case of fac above, the return address values will not always be the same.

2 ••• Implement Ackermann's function in ISCAL.

3 ••• Try to get rid of some of the recursions in Ackermann's function by converting them to iterations. Can you get rid of all recursion? [Ackermann's function is an example of a function that can be proved to be non-primitive-recursive.]

4 ••• Implement Quicksort in ISCAL.

Switch Statement Equivalents

While we are discussing machine language, it would be worthwhile to see how Java switch statements are compiled to take advantage of the linear addressing principle, as discussed earlier. As mentioned, the idea is that switches over a dense set of values are compiled to an array of jumps. Let us illustrate with an example. Consider the Java code

```
int i, x, y, z;
....
switch( i )
  {
  case 0:   x = y + z;  break;
  case 1:   x = y - z;  break;
  case 2:   x = y * z;  break;
  case 3:   x = y / z;  break;
  default:  x = 0;      break;
  }
```

An ISC equivalent of this code is shown below. The structure should be understood carefully, as it exemplifies the structure that could be used for any switch statement. There is an **initial** part where outlying cases, those corresponding to the default, are handled. Then there is a **dispatch** part where a jump address is computed by adding to a base jump address an appropriate multiple (in this case 2) of the integer upon which we are switching. Then there are branches, one for each different case and the default. Finally, there is a final part, to which each branch converges.

```
      use temp
      use zero
      use jump_target
      use converge
      lim converge converge_loc      // set up location for converging

// initial part
      lim zero 0
      lim jump_target default_branch
      jlt jump_target i zero         // handle i < 0
      lim temp 3
      jgt jump_target i temp         // handle i > 3

// dispatch part
      lim jump_target branch_array   // set up jump address
      add jump_target i jump_target
      add jump_target i jump_target  // add twice i
      junc jump_target               // jump to branch_array+2*i

label branch_array                   // dispatching array of jumps
                                     // each 2 locations
      lim jump_target branch_0       // case 0
      junc jump_target

      lim jump_target branch_1       // case 1
      junc jump_target

      lim jump_target branch_2       // case 2
```

```
      junc jump_target

      lim jump_target branch_3       // case 3
      junc jump_target

// one branch for each default and switch case

label default_branch                 // default case
      lim x 0
      junc converge

label branch_0                       // case 0
      add x y z
      junc converge

label branch_1                       // case 1
      sub x y z
      junc converge

label branch_2                       // case 2
      mul x y z
      junc converge

label branch_3                       // case 3
      div x y z
      junc converge

// converge here
label converge_loc                   // statements after switch
```

13.4 Memory-mapped I/O

There is a notable absence of any I/O (input/output) instructions in the ISC. While I/O instructions were included in early machines, modern architectures prefer to move such capabilities outside the processor itself. Part of the motivation for doing so includes:

> I/O devices are typically slower than computational speeds, so there is a hesitancy to provide instructions that would encourage tying up the processor waiting for I/O.
>
> The wide variety of I/O devices makes it difficult to provide for all possibilities in one processor architecture.

Instead of providing specific I/O instructions, modern architectures use the memory addressing mechanism to deal with I/O devices. These devices are identified with various memory locations, hence the term "memory-mapped I/O". When writing to those locations occurs, detection logic on the memory bus will interpret the contents as intended for the I/O device, rather than as an actual memory write. Thus the variety of I/O devices is essentially unlimited and the processor does not have to take such devices into account.

The most straightforward way to memory map I/O would be to assume a sequential or stream-oriented devices and have one location for input and one location for output. Whenever a read from the input location is issued, the next word in the input is read. Similarly, whenever a write to the output location is issued, a word is sent to the output device. As simple as it is, this picture is slightly undesirable, due to the disparity in speeds between typical I/O devices and processors. If the processor tried to read the location and the device was not ready to send anything, there would have to be a long wait for that memory access to return, during which time the processor is essentially idle. By providing a little more sophistication, there are ways to use this otherwise-idle time. A processor can separate the request for input and checking of whether the next word is ready to be transferred. In the intervening interval, other work could be done in principle. We achieve this effect by having two words per device, one for the datum being transferred and one for the status of the device.

Below we describe one possible memory mapping of an input device and an output device. These would be serial devices, such as a keyboard and monitor.

Location -1 (called **input_word**) is the location from which a word of input is read by the program. Location -2 (called **input_status**) controls the reading of data from the input device and serves as a location that can be tested for input status (e.g. normal vs. end-of-file). In order to read a word, input_status is set to 0 by the program. A write of 0 to this location triggers the input read. When the word has been input and is ready to be read by the program, the computer will change input_status to a non-zero value. The value 1 is used for normal input, while -1 is used for end-of-file condition.

The program should only set input_status to 0 if it is currently 1. If input_status is 0, then a read is already in progress and could be lost due to lack of synchronization. It can be assumed that input_status is initially 1, indicating the readiness of the input device. So a possible input sequence will be something like:

```
input_status = 0;              // start first read
end_of_file = 0;

while( ! end_of_file )
{
  ......                       // other processing can go on here
  while( input_status == 0 )   // wait for read complete
    {}
  switch( input_status )
    {
    case 1:
      use input_word;
      input_status = 0;        // start next read
      break;

    case -1:
      end_of_file = 1;         // indicate done
    }
}
```

Below we show a simpler input reader in ISCAL. This reader can be called as a procedure by the programmer to transfer the next input word. It assumes that the first input word has been requested by setting input_status to 0 earlier on.

```
define input_word_loc    -1 // fixed location for input word
define input_status_loc  -2 // fixed location for input status
use input_status            // register to hold input_status

register return 0       // standard return address reg
register result 1       // standard result register
register arg1   2       // first argument register

      lim input_status  input_status_loc  // setup input status reg
      store input_status zero             // request input
....

label input     // input routine, returns result in register 'result'
use input_word  // register to hold input_word_loc
use jump_target
use zero
use temp                              // temporary register
      lim zero 0
      lim input_word input_word_loc // memory-mapped input
      lim jump_target input_loop    // set up to loop back
label input_loop
      load  temp input_status       // get input status
      jeq   jump_target temp zero   // loop if previous input not ready
      jlt   halt temp zero          // quit if -1 (end-of-file)
      load  result input_word       // load from input word
      store input_status zero       // request next input
      junc  return
```

Output in the ISC gets a similar, although not identical, treatment. The routines are not identical because, unlike input, we cannot request output before the program knows the word to be output. Location -3 (called **output_word**) is the location to which a word of input is written by the program. Location -4 (called **output_status**) controls the writing of data to the output device and serves as a location that can be tested for output status. In order to write a word, output_status is set to 0 by the program, which in turn triggers the output write. When the word has been output, the computer will change output_status to a non-zero value.

It is important that output_status be tested to see that it is not already 0 before changing it. Otherwise, an output value can be lost. It can be assumed that output_status is 1 when the machine is started. So the normal output sequence will be something like:

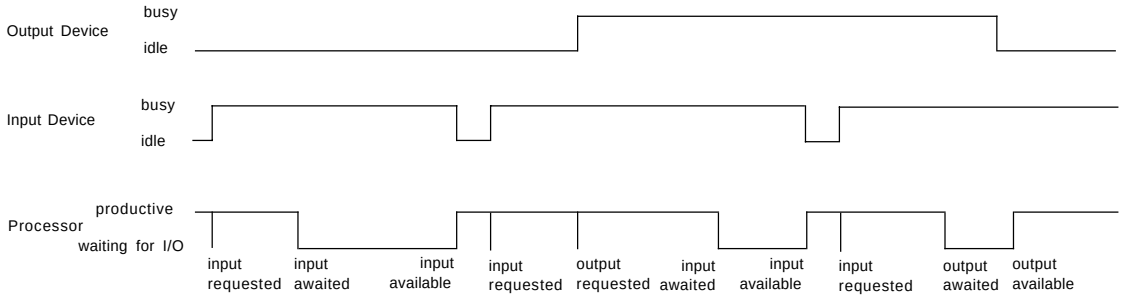
```
      while( more to be written )
        {
        ......                        // other processing can go on here
        while( output_status == 0 )   // wait for write complete
          {}
        output_word = next word to write;
        output_status = 0;
```

```
        }
```

A procedure for output of one word using this scheme in ISCAL is:

```
define output_word_loc   -3 // fixed location for output word
define output_status_loc -4 // fixed location for output status

register return 0       // standard return address reg
register result 1       // standard result register
register arg1   2       // first argument register

use output_status       // register to hold output_status

label output     // output routine, outputs word in register 'arg1'
use output_word  // register to hold output_word_loc
use jump_target
use zero
use temp                                // temporary register
      lim output_word output_word_loc // memory-mapped output
      lim zero 0
      lim jump_target output_loop     // set up loop address
label output_loop
      load  temp output_status        // get output status in temp
      jeq   jump_target temp zero     // jump back if output not ready
      store output_word arg1          // set up for output of result
      store output_status zero        // request output
      junc  return
```

The timing diagram below shows how these two routines could be called in a loop to keep both the input and output device busy, by requesting input in advance and only waiting when the processor cannot proceed without input or output being complete. This is an example of *overlapped I/O*, that is, input-output overlapped with processing.



Overlapped I/O timing

13.5 Finite-State Control Unit of a Computer Processor

Up to this point, we have seen the ISC primarily from the programmer's viewpoint. Next we look at a possible internal structure, particularly the various finite-state machine components that comprise it. Viewed from the processor, the instructions of the stored

program are also a form of "data". The computer reads the instructions as if data from memory and **interprets** them as instructions. In this sense, a computer is an interpreter, just as certain language processors are interpreters.

A typical memory abstraction employed in stored-program computers is as follows: The address of a word to be read or written is put into the MAR (memory address register). If the operation is a *write*, the word to be written is first put into the MDR (memory data register). On command from the sequencer of the computer, the memory is directed to write and transfers the word in the MDR into the address presented by the MAR. If the operation is a *read*, then the word read from the location presented in the MAR is put into the MDR by control external to the processor.

Instructions are normally taken from successive locations in memory. The register IP (instruction pointer) maintains the address of the location for the next instruction. Only when there is a "jump" indicated by an instruction does the IP value deviate from simply going from one location to the next. While the instruction is being interpreted, it is kept in the IR (instruction register). The reason that it cannot be kept in the MDR is because the MDR will be used for other purposes, namely reading or writing data to memory, during the instruction's execution. Unlike the numbered registers used in programming, the registers MAR, MDR, IP, and IR are not directly visible or referenceable by the programmer.

Refer to the ISC diagram below, showing a bus encircling most of the registers in the processor. (In actuality, multiple buses would probably be used, but this version is used for simplicity at this point.) This bus allows virtually any register shown to be gated into any other. At the lower left-hand corner, we see the control sequencer, a finite state machine that is responsible for the overall control of the processor. The control sequencer achieves its effect by selectively enabling the transfer of values from one register to another. The inputs to the sequencer consist of the value in the instruction register and the ALU test bit. Based on these, the sequencer goes through a series of state changes. In each state, certain transfers are enabled.

Every instruction executed undergoes the following **instruction fetch cycle** to obtain the instruction from memory (using Java notation):

```
MAR = IP;      // load the MDR with the address of next instruction
read_memory(); // get the instruction from that address into the MDR
IP++;          // set up for next instruction
IR = MDR;      // move the instruction to the IR
```

The portion of the sequencer for the instruction fetch cycle simply consists of four states. In the first state, the bus is used to gate IP into MAR. If we were to look at a lower level, this would mean that a set of 3-state buffers on the output of the IP is enabled, and a set of AND-gates on the input of the MAR is enabled. In the next state, read_memory is enabled (signaled to the memory controller). In the next state the IP register is incremented (we can build this logic into the register itself, similar to our discussion

regarding shift registers), and in the last state of the cycle, the output of the MDR is enabled onto the bus while the input to the IR is enabled.

The description above is simplified. If we had a fast memory, it would pay to do `IP++` at the same time as `read_memory()`, i.e. ***in parallel***, so that we used one fewer clock time for instruction fetch. More likely, we might have a slow memory that takes multiple clock cycles just to read a word. In this case, we would have additional **wait states** in the sequencer to wait until the memory read is done before going on. This would show up in our state diagram as a loop from the memory access state to itself, conditioned on memory not being finished.

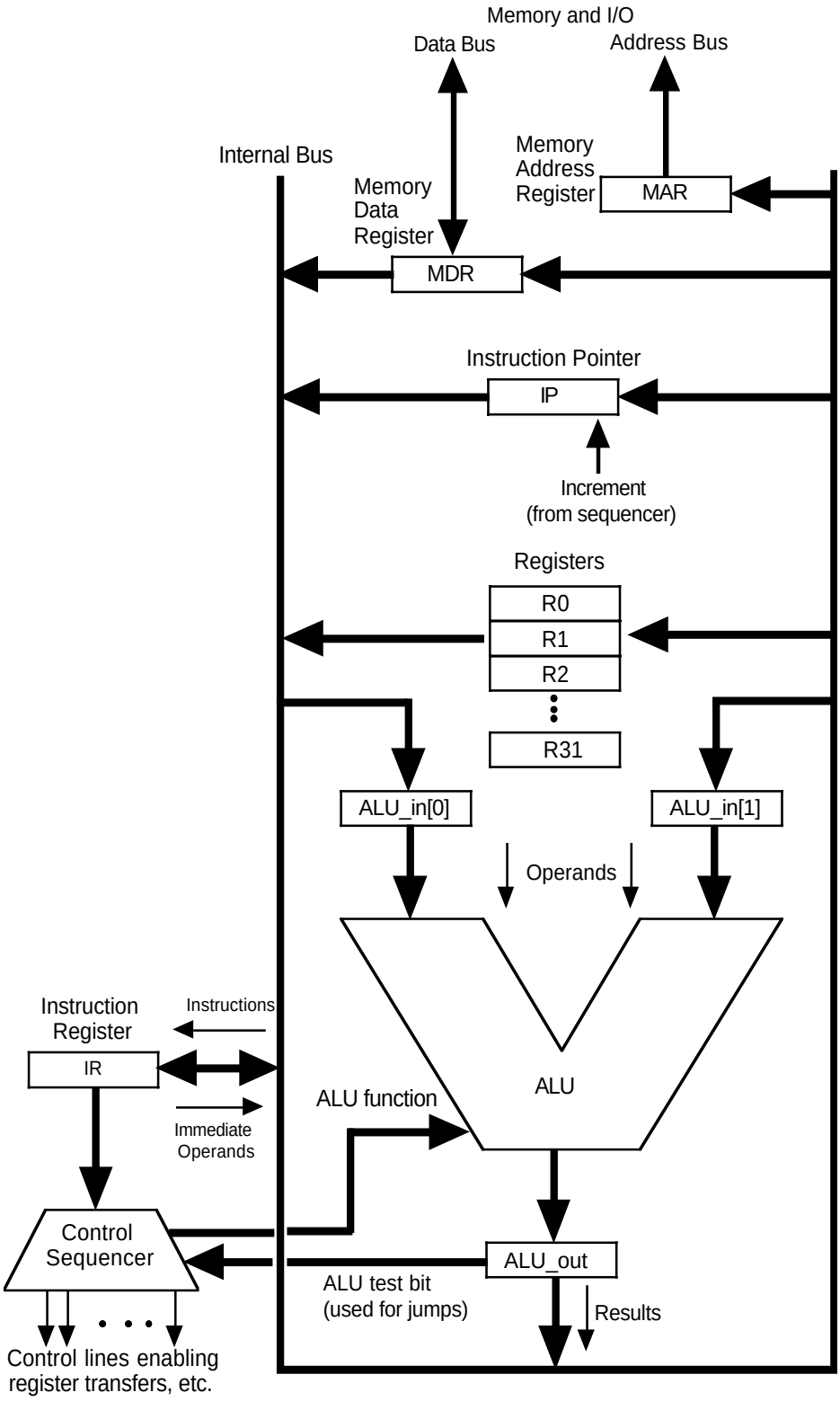


Figure 283: Possible ISC Internal Structure (unoptimized)

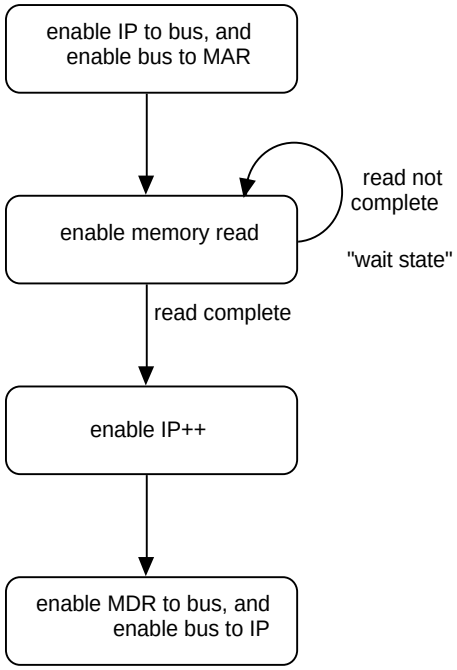


Figure 284: A possible state diagram for the instruction-fetch cycle of the ISC

Once the instruction to be interpreted is in the IR, a different cycle of states is used depending on the bits in the instruction. We give just a couple of examples here:

> If the instruction were **add Ra Rb Rc**, the intention is that we want to add the values in Rb and Rc and put the result in Ra. The sequence would be:

```
ALU_in[0] = Ra;
ALU_in[1] = Rb;
            // add done by the ALU here
Rc = ALU_out;
```

The registers Ra, Rb, and Rc are selected by **decoding** the binary register indices in the instruction and using it to drive 3-state selection (in the case of Ra and Rb) or and-gates (in the case of Rc), as per earlier discussion. We assume here that the combinational addition can be done in one clock interval. If not, additional states would be inserted to allow enough time for the addition to complete.

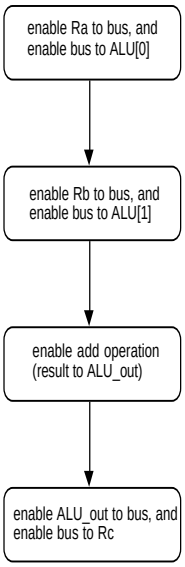


Figure 285: Portion of the ISC state diagram corresponding to the add operation

The ALU is capable of multiple functions: adding, subtracting, multiplying, shifting, AND-ing, shifting, etc. Exactly which function is performed is determined by bits provided by the IR. Most of the instructions involving data processing follow the same pattern as above.

If the instruction were **jeq Ra Rb Rc**, this is an example where the next instruction might be taken from a different location. The sequence would be:

```
ALU_in[0] = Rb;
ALU_in[1] = Rc;
            // comparison is done by the ALU here
if( the result of comparison is equal )
IP = Ra
```

Here Ra contains the address of the next instruction to be used in case Rb and Rc are equal. Otherwise, the current value of IP will just be used.

Overall, then, the behavior of the machine can be depicted as:

```
for( ; ; )
  {
  instruction_fetch();
  switch( IR OpCode bits )
    {
    case add:     add_cycle;        break;
    case sub:     subtract_cycle:   break;
    .
    .
    case jeq:     jeq_cycle;        break;
    .
    .
    }
```

```
}
```

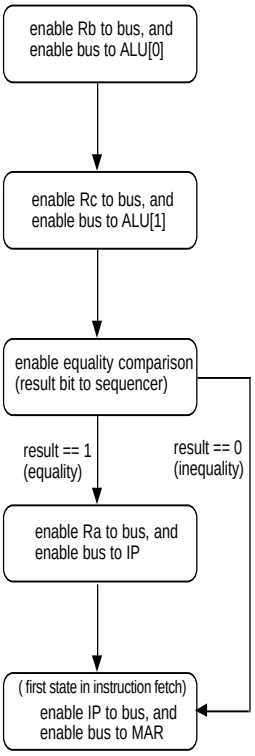


Figure 286: Portion of the ISC state diagram for the jeq instruction

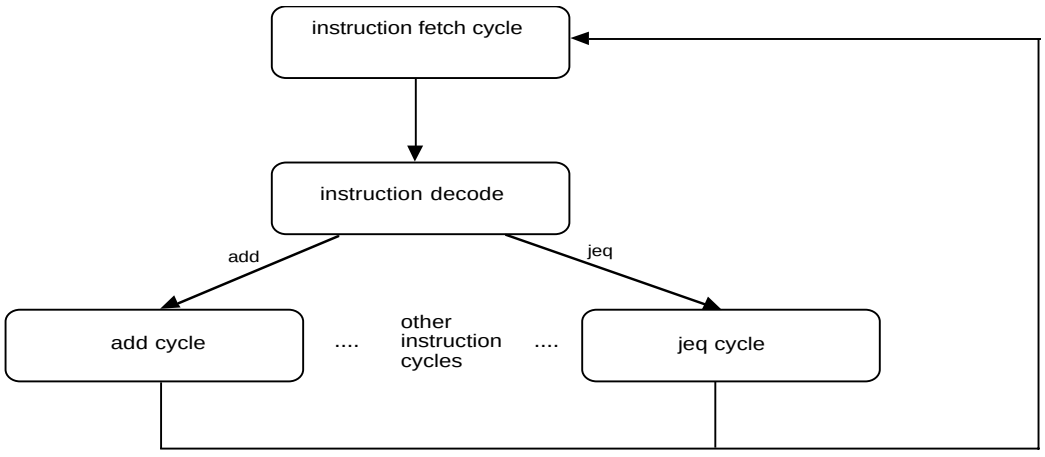


Figure 287: Overall state behavior of the ISC

Exercises

1 •• Based on the above discussion, estimate the number of states in each of the cycles for the various instructions in the ISC instruction set. Obtain an estimate of the number of states in the instruction sequencer. Assume that all memory operations and ALU operations take one clock period.

2 •••• How many flip-flops (in addition to those in the IR) would be sufficient to implement the sequencer? Give a naive estimate based on the preceding question, then a better estimate based on a careful assignment analysis of how functionality in the sequencer can be shared.

3 ••• By using more than one bus, some register transfers that would have been done in sequence can be done concurrently, or "in parallel". For example, in the add cycle, both Rb and Rc need to be transferred. This could, in principle, be done concurrently, but two buses would be required. Go through the instruction set and determine where parallelism is possible. Then optimize the ISC register-transfer structure so as to reduce the number of cycles required by as many instructions as possible.

13.6 Forms of Addressing in Other Machines

As mentioned earlier, the ISC uses register-indirect and immediate addressing only. The following diagrams abstract these two general forms.

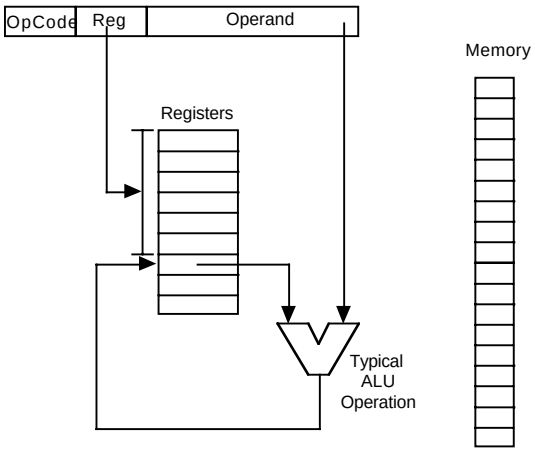


Figure 288: Immediate operand

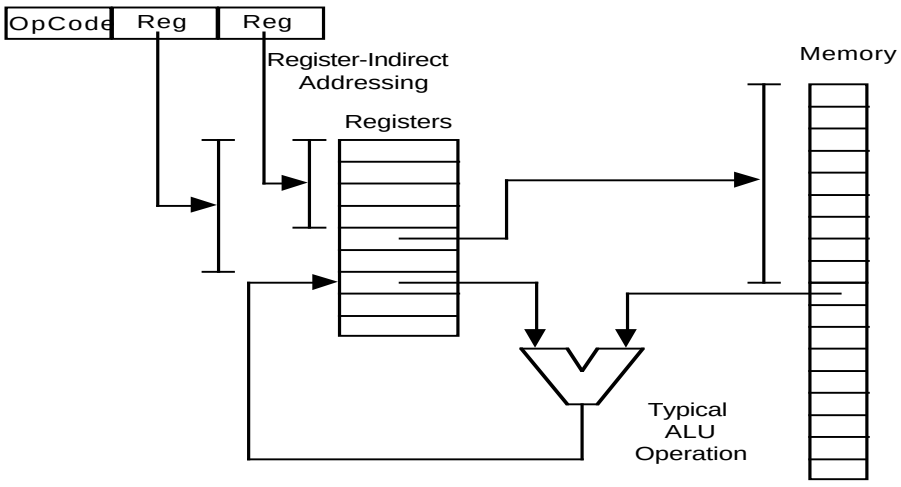


Figure 289: Register indirect addressing

For contrast, other machines might employ some or all of the following types of addressing:

direct addressing – The address of a datum is in the instruction itself. It is not necessary to load a register with the address. The problem with this mode of addressing is that addresses can be very large, making it difficult for a single instruction to directly address all of memory. For example, the ISC's address space is 32-bits, the same size as an instruction. This would leave no space in the instruction for op-code information.

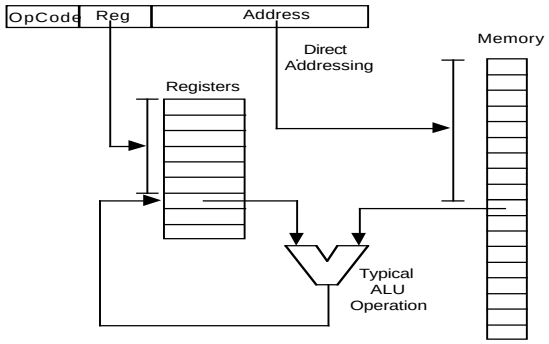


Figure 290: Direct addressing

indirect addressing – The address of the datum is in a word in memory. The instruction contains the address of the latter word. An example of the use of this type of addressing is pointer dereferencing. In a C++ statement

```
x = *p;
```

The address of p would be in the instruction. The contents of p is interpreted as a memory address. The contents of the latter address is stored into a register. The contents of the register would be stored into x by a subsequent instruction (unless the instruction can contain two addresses, one of which would be the address of x).

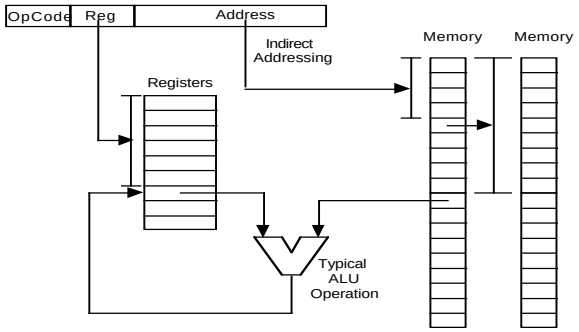


Figure 291: Indirect addressing

indexed addressing – The address of the datum is formed by adding the address in the instruction to the contents of a register, called the **index register**. This sum is called the effective address and is used as the address of the actual datum. In Java or C++, indexed addressing would be useful in indexing arrays. For example, in the statement

```
x = a[i];
```

the instruction could contain the base address, &a[0] and the index register could contain the value of i.

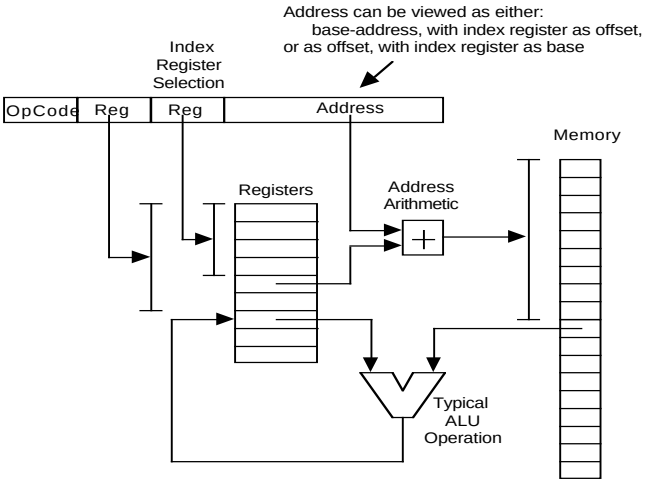


Figure 292: Indexed addressing

based addressing – This is similar to indexed addressing, except that the base address &a[0] is contained in a register. The value of i, called an **offset**, is contained in the word addressed by the instruction.

based indexed addressing – This uses two registers, one containing a base address and one containing an index. The instruction specifies an offset. The effective address is obtained by adding the base address, the index, and the offset. An example of a statement using such addressing would be

```
x = a[i+5];
```

where 5 would be the offset.

There are, of course, other possible combinations of addressing. Machines such as the ISC that do not have all of these forms of addressing must achieve the same effect by a sequence of instructions.

Exercises

1 •• Consider adding a *lix* (load-indexed) instruction to the ISC. This is similar to the load instruction, except that there is an additional register, called the index register. The address of the word in memory to be loaded (called the effective address) is formed by taking the sum of the address register, as in the current load instruction, plus the index register. Show how this instruction could be added to the ISC. Then suggest a possible use for the instruction. To retain symmetry, what other indexed instructions would be worthwhile?

2 •• Explain why a *jix* (jump-indexed) instruction might be useful.

3 •• How could indexing be useful in implementing recursion?

4 •• Give a diagram that abstracts based indexed addressing.

13.7 Processor-Memory Communication

Our diagram of the ISC internal structure omitted details of how the processor and memory interact. We indicated the presence of a data bus for communicating data to and from the memory and an address bus for communicating the address, but other details have been left out. There are numerous reasons for not including the memory in the same physical unit as the processor. For one thing, the processor will fit on a single VLSI chip, whereas a nominal-sized memory will not, at least not with current technology. It is also common for users to add more memory to the initial system configuration, necessitating a more modular approach to memory. Another reason for separation is that memory technology is generally slower than processors. Moderately-priced memory cannot

deliver data at the rate demanded by sophisticated processors. However, the memory industry keeps making memory faster, opening the possibility of an upgrade in speed. This is another reason not to tie down the processor to a particular memory speed.

Let us take a look at the control aspects of processor-memory communication. The processor and memory can be regarded as separate agents. When the processor needs data from the memory, it sends a request to the memory. The memory can respond when it has fulfilled the request. This type of dialog is called **handshaking**. The key components in handshaking, assuming the processor is making a read request, are:

a. Processor asserts address onto address bus.

b. Processor tells memory that it has a read request.

c. Memory performs the read.

d. Memory asserts data onto data bus.

e. Memory tells processor that data is there.

f. Process tells memory that it has the data.

g. Memory tells processor that it is ok to present the next request.

The following timing diagram indicates a simple implementation of handshaking along these lines. The transitions are labeled to correspond to the events above. However, step c is not shown because it is implicitly done by the memory, without communication. The **strobe** signal is under control of the processor to indicate initiation of a read. The ack signal is under control of the memory.

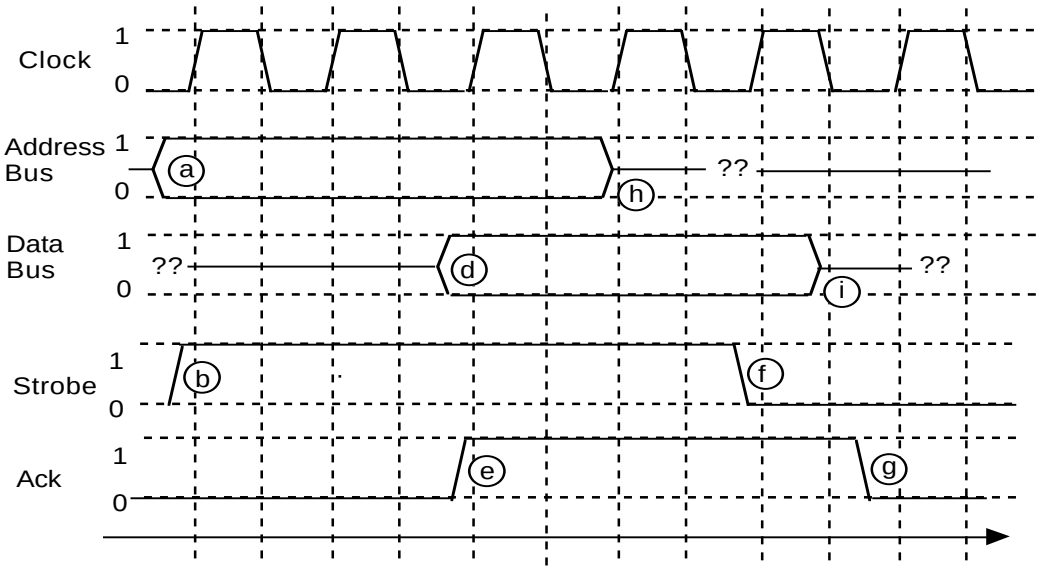


Figure 293: Handshaking sequence for a memory read

The address and data bus lines are shown as indicating both 0 and 1 during some part of the cycle. This means that the values could be either 0 or 1. Since addresses and data consist of several lines in parallel, some lines will typically be each. When the signal is shown mid-way, it means that it is not important what the value is at that point.

Events shown as *h* and *i* in the diagram are of less importance. Event h indicates that once the memory has read the data (indicated by event *e*), the address lines no longer need to be held.

The advantage of the handshaking principle is that it is effective no matter how long it takes for the memory to respond: The period between events *b* and *e* can just be lengthened accordingly. Meanwhile, if the processor cannot otherwise progress without the memory action having been completed, it can stay in a **wait state**, as shown in earlier diagrams. This form of communication is called **semi-asynchronous**. It is not truly asynchronous, since the changes in signals are still supposed to occur between clock signal changes.

The sequence for a memory write is similar. Since reads and writes typically share the same buses to save on hardware, it is necessary to have another signal so that the processor can indicate the type of operation. This is called the **read/write strobe**, and is indicated as **R/W**, with a value of 1 indicating read and a value of 0 indicating a write.

The following table and diagram shows the timing of a write sequence.

a. Processor asserts address onto address bus.

b. Processor asserts data onto data bus.

c. Processor tells memory that it has a request.

d. Memory performs the write.

e. Memory tells processor that write is performed.

f. Processor acknowledges previous signal from memory.

g. Memory tells processor that it is ok to present the next request.

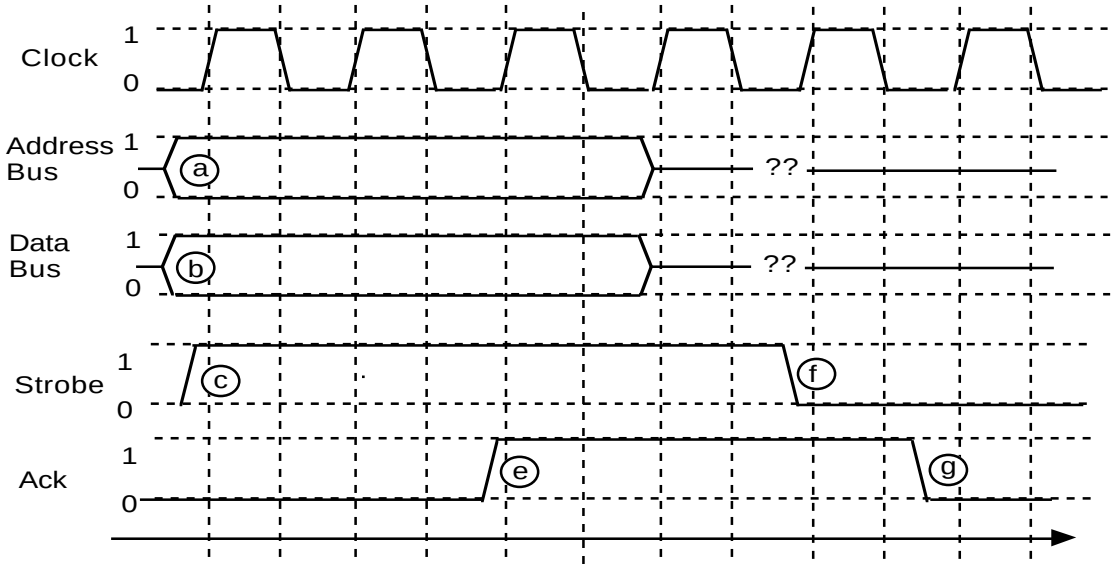


Figure 294: Handshaking sequence for a memory write

Again, it is the responsibility of the RW strobe to convey the type of request to the memory and thereby determine which of the above patterns applies. The handshaking principle is usable whenever it is necessary to communicate between independent sub-systems, not just between processor and memory. The general setup for such communication is shown by the following diagram, where *function strobe* is, for example, the RW line. The sub-system initiating the communication is called the *master* and the sub-system responding is called the *slave*.

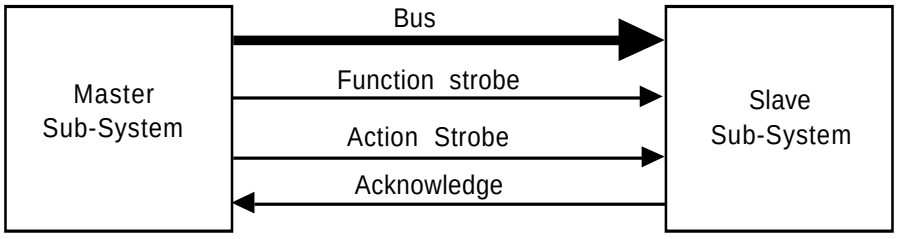


Figure 295: Set-up for communication using handshaking

13.8 Interrupts

When a processor wishes to initiate communication with the outside world, it can use the approach taken here for input/output: writing to certain special memory locations is interpreted by the processor's environment as a directive to carry out some action, such as

start an i/o device. It is also necessary to provide a way for the environment to get the attention of the processor. Without such a method, the processor would have to explicitly "poll" the environment to know whether certain events have taken place, e.g. the completion of an i/o operation. The problems with exclusive reliance on polling are the following:

- It is often unclear where the best place is in the program to insert polling instructions. If polling is done too often, time can be wasted. If it is done too seldom, then critical events can be left waiting the processor's attention too long.

- An end-user's program cannot be burdened with the insertion of polling code.

- If the program is errant, then polling might not take place at all.

The concept of "interrupt" is introduced to solve such problems. An interrupt is similar to a procedure call in that there is a return to the point where the program was interrupted (i.e. to where the procedure was called). However, an interrupt is different in that the call is not done explicitly by the interrupted code but rather by some external condition.

The fact that the call is not explicit in the code raises the issue of where the procedure servicing the interrupt is to reside, so that the processor can go there and execute instructions. Typically, there are preset agreed upon locations for this purpose. These locations are aggregated in a small array known as the **interrupt vector**. Typically a special register indicates where this vector is in memory. The interrupt vector is indexed by an integer that indicates the cause of the interrupt. For example, if there are four different classes of devices that can interrupt, there might be four locations in the interrupt vector. The locations within the interrupt vector are address of routines called **interrupt service routines**.

The sequence of actions that take place at an interrupt is:

The cause of interrupt is translated by the hardware into an index, used to access the interrupt vector.

The current value of the instruction pointer (IP register) is saved in a **special interrupt save location**. This provides the return address later on.

The IP register is loaded with the address specified at the indexed position within the interrupt vector.

Execution at this point is within the interrupt service routine.

At the end of the interrupt service routine, a **return-from-interrupt** instruction causes the IP to be reloaded with the address in the interrupt save location.

Execution is now back within the program that was interrupted in the first place.

The addition of interrupts has thus necessitated the introduction of one new instruction, return-from-interrupt, to the repertoire of the processor. It also requires a new processor register to point to the base of the interrupt vector. Finally, there needs to be a way to get to the interrupt save location. One scheme for doing this might be to interleave the save locations with the addresses in the interrupt vector. In this way, no additional register is needed to point to the interrupt save location. Furthermore, we have one such location per interrupt vector index. This is useful, since it should be possible for a higher priority interrupt to interrupt the service routine of a lower priority interrupt. Finally, we don't want to allow the converse, i.e. a lower priority interrupt to interrupt a higher priority one. To achieve this, there would typically be an **interrupt mask register** in the processor that indicates which class of interrupts is enabled. The interrupt mask register contents is changed automatically by the processor when an interrupt occurs and when a return-from-interrupt instruction is executed.

Interrupts vs. Traps

Communication with the environment is not the only need for an interrupt mechanism. There are also needs internal to the processor, which correspond to events that we don't want to have to test repeatedly but which nonetheless occur. Examples include checking for arithmetic overflow within registers and for memory protection violations. The latter are designed to keep an errant program from over-writing itself. Sometimes these internal causes are distinguished from interrupts by calling them "**traps**". Traps are also used for debugging and for communicating with the operating system. It is unreasonable to simply allow a user program to jump to the operating system code; the latter must have special privileges that the user program does not. The only way to provide the transfer from an unprivileged domain to a privileged one is through a trap, which causes a change in a set of mask registers that deal with privileges.

13.9 Direct Memory Access I/O

While interrupts assist in the ability for a processor to communicate with input/output devices at high speed, it is often too slow to have an interrupt deal with every word transferred to or from a device. Some devices demand such great attention that it would slow down the executing program significantly to be interrupted so often. To avoid such slow down, special secondary processors are often introduced to handle the flow of data to and from high-speed devices. These are variously known as **DMA** (direct memory access) **channels** (or simply "channels") or **peripheral processors**. A channel competes with the processor for memory access. It transfers an entire array of locations in one

single interaction from the processor, maintaining its own pointer to a word in memory that is next to be transferred. Rather than interrupting the processor at every word transfer, it only interrupts the processor on special events, such as the completion of an array transfer.

13.10 Pipelining within the Processor

In order to gain an additional factor in execution speed, modern processors are designed for "pipelined" execution. This means that multiple, rather than a single, instructions are being executed concurrently, albeit at different stages within their execution cycles. For example, instruction n can be executing an add instruction while instruction n+1 is fetching some data from memory. In order for pipelining to work, there must be additional internal registers and control provided that make the net result for pipelined execution be the same as for sequential execution. To give a detailed exposition of pipelining is beyond the scope of the present text. The reader may wish to consult a more advanced text or the tutorial article [Keller 1975].

13.11 Virtual Memory

Virtual memory is a scheme that simplifies programming by allowing there to be more accessible words than there is physical memory space. This is accomplished by "swapping" some of the memory contents to a secondary storage device, such as a disk. The hardware manages the record-keeping of what is on disk vs. in main memory. This is accomplished by translating addresses from the program to physical addresses through a "page table". Memory is divided up into blocks of words known as pages, which contain sets of contiguous storage locations. When the processor wants to access a word, it uses the higher-order so many bits to access the page table first. The page table indicates where the page, either in main memory or secondary storage, and where it is. If the page is in main memory, the processor can get the word by addressing relative to the physical page boundary. If it is on disk, the processor issues an i/o command to bring the page in from disk. This may also entail issuing a command to write the current contents of some physical memory page to disk, to make room for the incoming page. The page idea also alleviates certain aspects of main memory allocation, since physical pages are not required to be contiguous across page boundaries. This is an example of the linear-addressing principle being applied at two levels: once to find the page and a second time to find the word within the page. There is a constant-factor net slow-down in access as a result, but this is generally considered worth it in terms of the greater convenience it provides in programming.

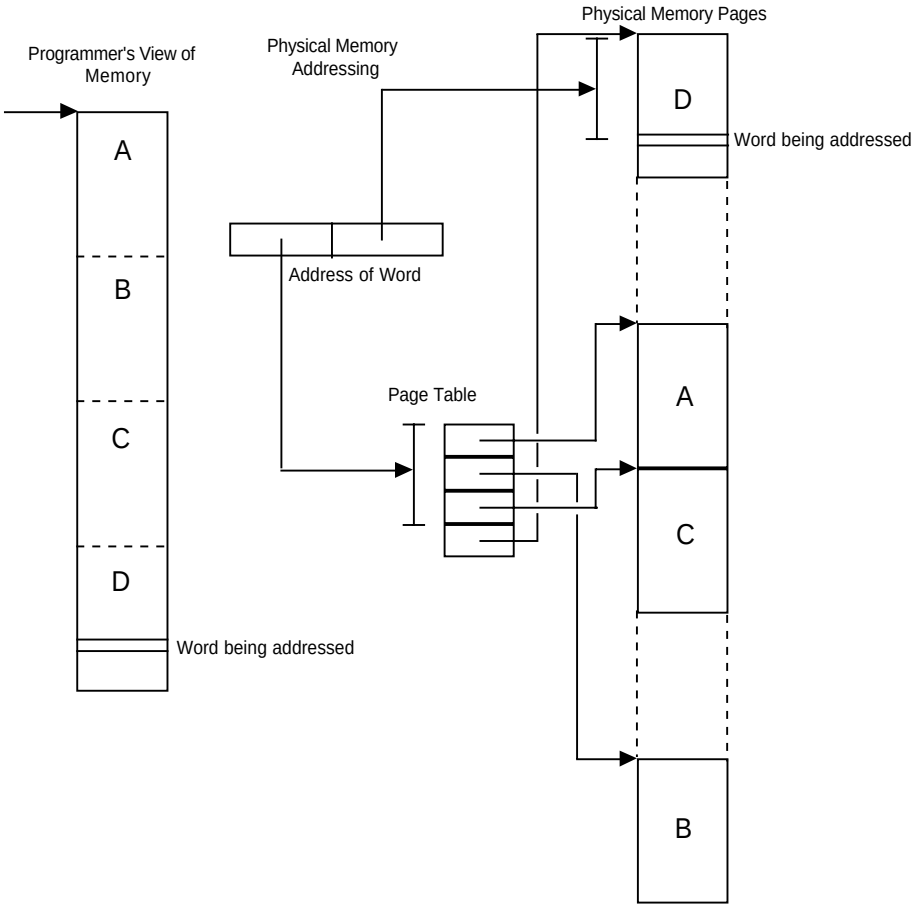


Figure 296: How address translation is done for virtual memory; The physical memory pages could be in main memory or on disk.

13.12 Processor's Relation to the Operating System

Very seldom is a processor accessed directly by the user. At a minimum, a set of software known as the "operating system" provides utility functions that would be too complex to code for the average user. These include:

Loading programs into memory from external storage (e.g. disk).

Communication with devices, interrupt-handling, etc.

A file system for program and data storage.

Virtual memory services, to give the user program the illusion that it has much more memory available than it really does.

Multiple-user coordination, so that the processor resource can be kept in constant use if there is sufficient demand.

The close connection between processors and operating systems demands that operating systems, as well as other software, must be kept in mind when processors are designed. For this reason, it is unreasonable to consider designing a modern processor without a thorough knowledge of the kind of operating system and languages that are anticipated being run on it.

Exercises

1 ••• Modify the design of the ISC to include an interrupt handling facility. Show all additional registers and define the control sequences for when an interrupt occurs.

2 ••• Design a channel processor for the ISC. Show how the ISC would initiate channel operations and how the channel would interact with the interrupt mechanism.

3 ••• Design a paging mechanism for the ISC.

4 ••• A feature of most modern processors is *memory protection.* This can be implemented using a pair of registers in the processor that hold the lower and upper limit of addresses having contents modifiable by the currently-running program. Modify the ISC design to include memory protection registers. Provide an instruction for setting these limit registers under program control.

13.13 Chapter Review

Define the following terms:

assembly language
complex-instruction set computer
direct memory access (DMA)
directives (assembler)
effective address
handshaking
instruction decoding
interpreter
interrupt
linear addressing principle
memory address register
memory data register
recursion
reduced-instruction set computer
stack
strobe
switch statement

trap
virtual memory
wait state

13.14 Further Reading

V. Carl Hamacher, Zvonko G. Vranesic, and Safwat G. Zaky, *Computer Organization*, Third Edition, McGraw-Hill, New York, 1990.

John L. Hennessy and David A. Patterson, *Computer architecture – A quantitative approach*. Morgan Kauffman Publishers, Inc., 1990.

Robert M. Keller. *Look-ahead processors*. ACM Computing Surveys, **7**, 4, 177-195 (December 1975).

14. Parallel Computing

14.1 Introduction

This chapter describes approaches to problems to which multiple computing agents are applied simultaneously.

By "parallel computing", we mean using several computing agents concurrently to achieve a common result. Another term used for this meaning is "concurrency". We will use the terms *parallelism* and *concurrency* synonymously in this book, although some authors differentiate them.

Some of the issues to be addressed are:

> What is the role of parallelism in providing clear decomposition of problems into sub-problems?
>
> How is parallelism specified in computation?
>
> How is parallelism effected in computation?
>
> Is parallel processing worth the extra effort?

14.2 Independent Parallelism

Undoubtedly the simplest form of parallelism entails computing with totally independent tasks, i.e. there is no need for these tasks to communicate. Imagine that there is a large field to be plowed. It takes a certain amount of time to plow the field with one tractor. If two equal tractors are available, along with equally capable personnel to man them, then the field can be plowed in about half the time. The field can be divided in half initially and each tractor given half the field to plow. One tractor doesn't get into another's way if they are plowing disjoint halves of the field. Thus they don't need to communicate. Note however that there is some initial overhead that was not present with the one-tractor model, namely the need to divide the field. This takes some measurements and might not be that trivial. In fact, if the field is relatively small, the time to do the divisions might be more than the time saved by the second tractor. Such overhead is one source of dilution of the effect of parallelism.

Rather than dividing the field only two ways, if N tractors are available, it can be divided N ways, to achieve close to an N-fold gain in speed, if overhead is ignored. However, the larger we make N, the more significant the overhead becomes (and the more tractors we have to buy).

In UNIX® command-line shells, independent parallelism can be achieved by the user within the command line. If c_1, c_2, ..., c_n are commands, then these commands can be executed in parallel in principal by the compound command:

$$c_1 \& c_2 \& .... \& c_n$$

This does not imply that there are *n* processors that do the work in a truly simultaneous fashion. It only says that logically these commands can be done in parallel. It is up to the operating system to allocate processors to the commands. As long as the commands do not have interfering side-effects, it doesn't matter how many processors there are or in what order the commands are selected. If there are interfering side-effects, then the result of the compound command is not guaranteed. This is known as indeterminacy, and will be discussed in a later section.

There is a counterpart to independent parallelism that can be expressed in C++. This uses the *fork* system call. Execution of `fork()` creates a new *process* (program in execution) that is executing the same code as the program that created it. The new process is called a *child*, with the process executing `fork` being called the *parent*. The child gets a complete, but independent, copy of all the data accessible by the parent process.

When a child is created using fork, it comes to life as if it had just completed the call to fork itself. The only way the child can distinguish itself from its parent is by the return value of fork. The child process gets a return value of 0, while the parent gets a non-zero value. Thus a process can tell whether it is the parent or the child by examining the return value of fork. As a consequence, the program can be written so as to have the parent and child do entirely different things within the same program that they share.

The value returned by fork to the parent is known as the *process id* (pid) of the child. This can be used by the parent to control the child in various ways. One of the uses of the pid, for example, is to identify that the child has terminated. The system call wait, when given the pid of the child, will wait for the child to terminate, then return. This provides a mechanism for the parent to make sure that something has been done before continuing. A more liberal mechanism involves giving wait an argument of 0. In this case, the parent waits for termination of any one of its children and returns the pid of the first that terminates.

Below is a simple example that demonstrates the creation of a child process using fork in a UNIX® environment. This is C++ code, rather than Java, but hopefully it is close enough to being recognizable that the idea is conveyed.

```
#include <iostream.h>                    // for <<, cin, cout,
cerr
#include <sys/types.h>                   // for pid_t
#include <unistd.h>                      // for fork()
#include <wait.h>                        // for wait()
```

```
main()
{
pid_t pid, ret_pid;  // pids of forked and finishing child

pid = fork();                           // do it

// fork() creates a child that is a copy of the parent
// fork() returns:
//       0 to the child
//       the pid of the child to the parent

if( pid == 0 )
  {
  // If I am here, then I am the child process.
  cout << "Hello from the child" << endl << endl;
  }
else
  {
  // If I am here, then I am the parent process.
  cout << "Hello from the parent" << endl << endl;

  ret_pid = wait(0);                    // wait for the child
  if( pid == ret_pid )
    cout << "child pid matched" << endl << endl;
  else
    cout << "child pid did not match" << endl << endl;
  }
}
```

The result of executing this program is:

```
Hello from the parent

Hello from the child

child pid matched
```

14.3 Scheduling

In the UNIX® example above, there could well be more processes than there are processors to execute those processes. In this case, the *states* of non-running processes are saved in a pool and executed when a processor becomes available. This happens, for example, when a process terminates. But it also happens when a process waits for i/o. This notion of *multiprocessing* is one of the key ways of keeping a processor busy when processes do i/o requests.

In the absence of any priority discipline, a process is taken from the pool whenever a processor becomes idle. To provide an analogy with the field-plowing problem, work apportionment is simplified in the following way: Instead of dividing the field into one segment per tractor, divide it into many small parcels. When a tractor is done plowing its current parcel, it finds another unplowed parcel and does that. This scheme, sometimes

called *self-scheduling*, has the advantage that the tractors stay busy even if some run at much different rates than others. The opposite of self-scheduling, where the work is divided up in advance, will be called *a priori scheduling*.

14.4 Stream Parallelism

Sets of tasks that are totally independent of each other do not occur as frequently as one might wish. More interesting are sets of tasks that need to communicate with each other in some fashion. We already discussed command-line versions of a form of communication in the chapter on high-level functional programming. There, the compound command

$$c_1 \mid c_2 \mid .... \mid c_n$$

is similar in execution to the command with | replaced by &, as discussed earlier. However, in the new case, the processes are synchronized so that the input of one waits for the output of another, on a character-by-character basis, as communicated through the standard inputs and standard outputs. This form of parallelism is called *stream parallelism*, suggesting data flowing through the commands in a stream-like fashion.

The following C++ program shows how pipes can be used with the C++ iostream interface. Here class filebuf is used to connect file buffers that are used with streams to the system-wide filed descriptors that are produced by the call to function `pipe`.

```
#include <streambuf.h>
#include <iostream.h>          // for <<, cin, cout, cerr
#include <stdio.h>             // for fscanf, fprintf
#include <sys/types.h>         // for pid_t
#include <unistd.h>            // for fork()
#include <wait.h>              // for wait()

main()
{
int pipe_fd[2];                // pipe file descriptors:
                               // pipe_fd[0] is used for the read end
                               // pipe_fd[1] is used for the write end

pid_t pid;                     // process id of forked child

pipe(pipe_fd);                 // make pipe, set file descriptors

pid = fork();                  // fork a child

int chars_read;

if( pid == 0 )
  {
  // Child process does this

  // read pipe into character buffer repeatedly, until end-of-file,
```

```
  // sending what was read to cout

  // note: a copy of the array pipe_fd exists in both processes

  close(pipe_fd[1]);          // close unused write end

  filebuf fb_in(pipe_fd[0]);
  istream in(&fb_in);

  char c;

  while( in.get(c) )
    cout.put(c);

  cout << endl;
  }
else
  {
  close(pipe_fd[0]);           // close unused read end

  filebuf fb_out(pipe_fd[1]);
  ostream out(&fb_out);

  char c;

  while( cin.get(c) )
    out.put(c);

  close(pipe_fd[1]);

  wait(0);                    // wait for child to finish
  }
}
```

It is difficult to present a plowing analogy for stream parallelism – this would be like the field being forced through the plow. A better analogy would be an assembly line in a factory. The partially-assembled objects move from station to station; the parallelism is among the stations themselves.

Exercises

Which of the following familiar parallel enterprises use self-scheduling, which use *a priori* scheduling, and which use stream parallelism? (The answers may be locality-dependent.)

1 • check-out lanes in a supermarket

2 • teller lines in a bank

3 • gas station pumps

4 • tables in a restaurant

5 • carwash

6 •••• Develop a C++ class for pipe streams that hides the details of file descriptors, etc.

14.5 Expression Evaluation

Parallelism can be exhibited in many kinds of expression evaluations. The UNIX® command-line expressions are indeed a form of this. But how about with ordinary arithmetic expressions, such as

(a + b) * ((c - d) / (e + f)))

Here too there is the implicit possibility of parallelism, but at a finer level of granularity. The sum a + b can be done in parallel with the expression (c - d) / (e + f). This expression itself has implicit parallelism. Synchronization is required in the sense that the result of a given sub-expression can't be computed before the principal components have been computed. Interestingly, this form of parallelism shows up if we inspect the dag (directed acyclic graph) representation of the expression. When there are two nodes with neither having a path to the other, the nodes could be done concurrently in principle.

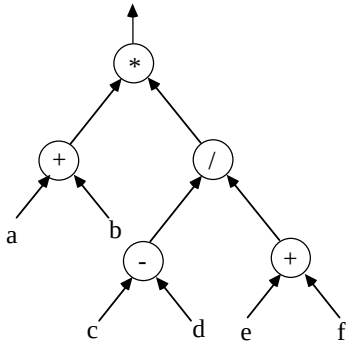


**Figure 297: Parallelism in an expression tree:
The left + node can be done in parallel with any
of the nodes other than the * node.
The - node can be done in parallel with the right + node.**

In the sense that synchronization has to be done from two different sources, this form of parallelism is more complex than stream parallelism. However, stream parallelism has the element of repeated synchronization (for each character) that scalar arithmetic expressions do not. Still, there is a class of languages in which the above expression might represent computation on vectors of values. These afford the use of stream parallelism in handling the vectors.

For scalar arithmetic expressions, the level of granularity is too fine to create processes – the overhead of creation would be too great compared to the gain from doing the operations. Instead, arithmetic expressions that can be done in parallel are usually used to

exploit the "pipelining" capability present in high-performance processors. There are typically several instructions in execution simultaneously, at different stages. A high degree of parallelism translates into lessened constraints among these instructions, allowing more of the instruction-execution capabilities to be in use simultaneously.

Expression-based parallelism also occurs when data structures, such as lists and trees, are involved. One way to exploit a large degree of parallelism is through the application of functions such as *map* on large lists. In mapping a function over list, we essentially are specifying one function application for each element of the list. Each of these applications is independent of the other. The only synchronization needed is in the use vs. formation of the list itself: a list element can't be used before the corresponding application that created it is done.

Recall that the definition of *map* in rex is:

```
map(Fun, []) => [].

map(Fun, [A | X]) => [apply(Fun, A) | map(Fun, X)].
```

The following figure shows how concurrency results from an application of *map* of a function *f* to a list [x_1, x_2, x_3, ...]. The corresponding function in rex that evaluates those function applications in parallel is called *pmap*.

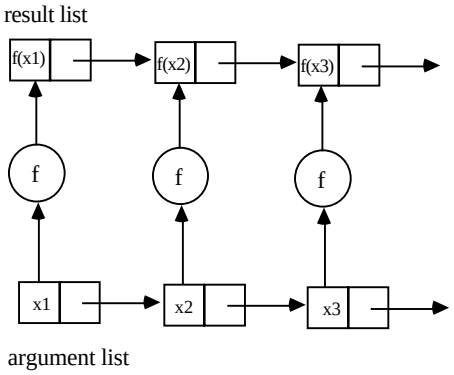


Figure 298: Parallel mapping a function over a list for independent execution. Each copy of f can be simultaneously executing.

Exercises

Which of the following programs can exploit parallelism for improved performance over sequential execution? Informally describe how.

1 •• finding the maximum element in an array

2 •• finding an element in a sorted array

3 •• merge sort

4 •• insertion sort

5 •• Quicksort

6 ••• finding a maximum in a uni-modal array (an array in which the elements are increasing up to a point, then decreasing)

7 •• finding the inner-product of two vectors

8 •• multiplying two matrices

14.6 Data-Parallel Machines

We next turn our attention to the realization of parallelism on actual computers. There are two general classes of machines for this purpose: data-parallel machines and control-parallel machines. Each of these classes can be sub-divided further. Furthermore, each class of machines can simulate the other, although one kind of machine will usually be preferred for a given kind of problem.

Data parallel machines can be broadly classified into the following:

SIMD multiprocessors
Vector Processors
Cellular Automata

SIMD Multiprocessors

"SIMD" stands for "single-instruction stream, multiple data stream". This means that there is one stream of instructions controlling the overall operation of the machine, but multiple data operation units to carry out the instructions on distinct data. The general structure of a SIMD machine can thus be depicted as follows:

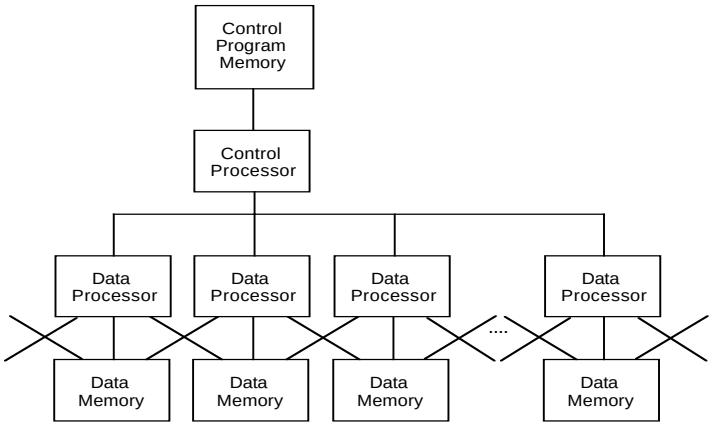


Figure 299: SIMD multiprocessor organization

Notice that in the SIMD organization, each data processor is coupled with its own data memory. However, in order to get data from one memory to another, it is necessary for each processor to have access to the memories of its neighbors, at a minimum. Without this, the machine would be reduced to a collection of almost-independent computers.

Also not clear in the diagram is how branching (jumps) dependent on data values take place in control memory. There must be some way for the control processor to look at selected data to make a branch decision. This can be accomplished by instructions that form some sort of aggregate (such as the maximum) from data values in each data processors' registers.

SIMD Multiprocessors are also sometimes called *array processors*, due to their obvious application to problems in which an array can be distributed across the data memories.

Vector Processors

The name "vector processor" sounds similar to "array processor" discussed above. However, vector processor connotes something different to those in the field: a processor in which the data stored in *registers* are vectors. Typically there are both scalar registers and vector registers, with the instruction code determining whether an addressed register is one or the other. Vector processors differ from array processors in that not all elements of the vector are operated on concurrently. Instead pipelining is used to reduce the cost of a machine that might otherwise have one arithmetic unit for each vector element.

Typically, vector operations are floating-point. Floating point arithmetic can be broken into four to eight separate stages. This means that a degree of concurrency equal to the number of stages can be achieved without additional arithmetic units. If still greater concurrency is desired, additional pipelined arithmetic units can be added, and the vectors apportioned between them.

Cellular Automata

A cellular automaton is a theoretical model that, in some ways, is the ultimate data parallel machine. The machine consists of an infinite array of cells. Each cell contains a state drawn from a finite set, as well as a finite state machine, which is typically the same for every cell. The state transitions of the machine use the cell's own state, as well as the states of selected other cells (known as the cell's neighbors), to determine the cell's next state. All cells compute their next states simultaneously. The entire infinite array therefore operates in locked-step fashion.

In most problems of interest, only a finite number of the cells are in a non-quiescent state. In other words, most of the cells are marking time. The state-transition rules are usually designed this way. The automaton is started with some specified cells being non-quiescent. This is how the input is specified. Then non-quiescent states generally propagate from those initial non-quiescent cells.

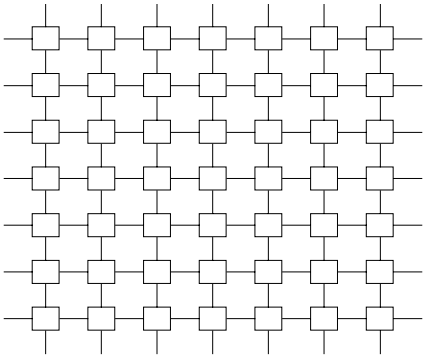


Figure 300: Fragment of a two-dimensional cellular automaton; cells extend forever in all four compass directions

We have shown above a two-dimensional cellular automaton. However, cellular automata can be constructed in any number of dimensions, including just one. It is also possible to consider irregular cellular automata, connected as an arbitrary graph structure, in which there is no clear notion of dimension. Most cellular automata that have been studied are rectangular ones in one, two, or three dimensions. It is also possible to use different sets of neighbors than the one shown. For example, an eight-neighbor automaton is common.

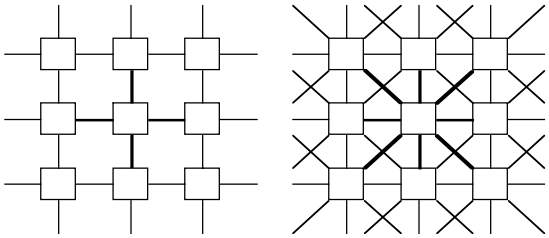


Figure 301: Four- vs. eight-neighbor cellular automata

Cellular automata were studied early by John von Neumann. He showed how Turing machines can be embedded within them, and moreover how they can be made to reproduce themselves. A popular cellular automaton is Conway's "Game of Life". Life is a two-dimensional cellular automaton in which each cell has eight neighbors and only two states (say "living" and "non-living", or simply 1 and 0). The non-living state corresponds to the quiescent state described earlier.

The transition rules for Life are very simple: If three of a cell's neighbors are living, the cell itself becomes living. Also, if a cell is living, then if its number of living neighbors is other than two or three, it becomes non-living.

The following diagram suggests the two kinds of cases in which the cell in the center is living in the next state. Each of these rules is only an example. A complete set of rules would number sixteen.

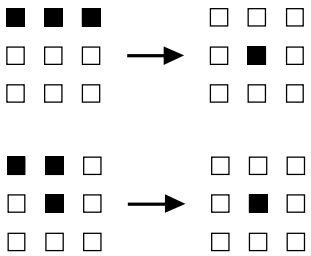


Figure 302: Examples of Life rules

Many interesting phenomena have been observed in the game of life, including patterns of living cells that appear to move or "glide" through the cellular space, as well as patterns of cells that produce these "gliders".

Figure 303: A Life glider pattern

It has been shown that Life can simulate logic circuits (using the presence or absence of a glider to represent a 1 or 0 "flowing" on a wire). It has also been shown that Life can simulate a Turing machine, and therefore is a universal computational model.

Exercises

1 • Enumerate all of the life rules for making the center cell living.

2 •• Write a program to simulate an approximation to Life on a bounded grid. Assume that cells outside the grid are forever non-living.

3 •• In the Fredkin automaton, there are eight neighbors, as with life. The rules are that a cell becomes living if it is non-living and an odd number of neighbors are living. It becomes non-living if it is living and an even number of neighbors are living. Otherwise it stays as is. Construct a program that simulates a Fredkin automaton. Observe the remarkable property that any pattern in such an automaton will reproduce itself in sufficient time.

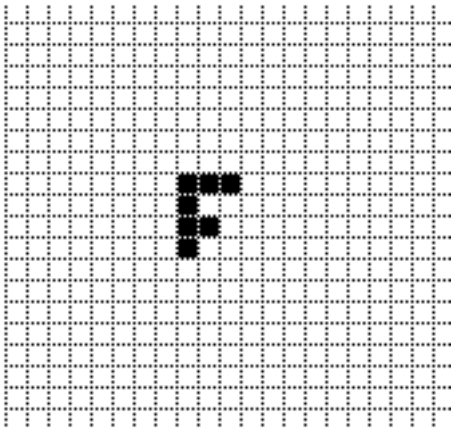


Figure 304: The Fredkin automaton, with an initial pattern

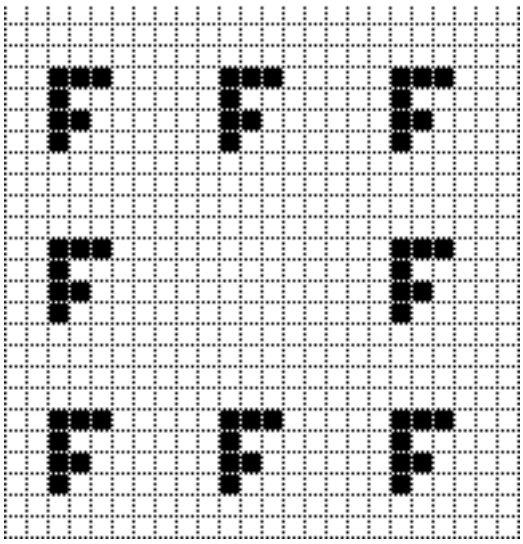


Figure 305: The Fredkin automaton, after 8 transitions from the initial pattern

4 ••• For the Fredkin automaton, make a conjecture about the reproduction time as a function of the number of cells to be reproduced.

5 •• Show that any Turing machine can be simulated by an appropriate cellular automaton.

6 •••• Write a program to simulate Life on an unbounded grid. This program will have to dynamically allocate storage when cells become living that were never living before.

7 ••••• A "garden-of-Eden" pattern for a cellular automaton is one that cannot be the successor of any other state. Find such a pattern for Life.

14.7 Control-Parallel Machines

We saw earlier how data-parallel machines rely on multiple processors conducting similar operations on each of a large set of data. In control-parallel machines, there are multiple instruction streams and no common control. Thus these machines are also often called MIMD ("Multiple-Instruction, Multiple-Data") machines. Within this category, there are two predominant organizations:

Shared memory

Distributed memory

Shared Memory Multiprocessors

In a shared memory machine, all processors conceptually share a common memory. Each processor is executing its own instruction stream. The processors can communicate with one another based on inspection and modification of data common to both of them.

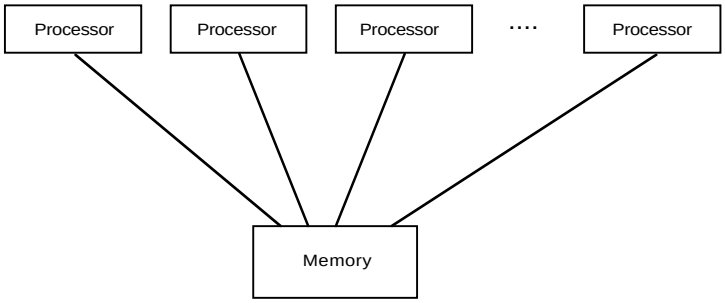


Figure 306: Shared Memory Multiprocessor

As an example of how shared memory permits one process to control another, suppose we want to make one process wait until an event occurs, as determined by another process. We can achieve this in a shared memory machine by agreeing on a common location to contain a flag bit. We agree in advance that a 0 in this bit means the event has not occurred, while a 1 indicates it has. Then the process that potentially waits will test the bit. If it is 0, it loops back and tests again, and so on. If it is 1, it continues. All the signaling processor has to do is set the bit to 1.

The following code fragments show the use of a flag for signaling between two processes.

Flag = 0 initially

Process A:

```
A1:  ....
A2: Flag = 1;

A3: ....
```

Process B:

```
B1: ....
B2: if( Flag == 0 )
      goto B2;
B3: ....
```

The following state diagram shows that the signaling scheme works. The progress of process A corresponds to the vertical dimension, and that of process B to the horizontal. The components of each state are:

(ip of A, ip of B, Flag)

Note that states in which ip of B is B3 cannot be reached as long as Flag is 0. Only when A has set the flag to 1 can such a state (at the lower-left corner) be reached.

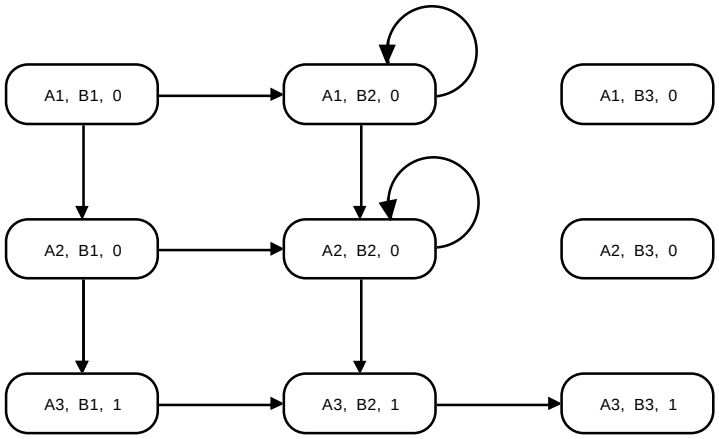


Figure 307: State-transition diagram of multiprocessor flag signaling

The type of signaling described above is called "busy-waiting" because the looping processor is kept busy, but does no real work. To avoid this apparent waste of the processor as a resource, we can interleave the testing of the bit with some useful work. Or we can have the process "go to sleep" and try again later. Going to sleep means that the process gives up the processor to another process, so that more use is made of the resource.

Above we have shown how a shared variable can be used to control the progress of one process in relation to another. While this type of technique could be regarded as a feature of shared memory program, it can also present a liability. When two or more processors share memory, it is possible for the overall result of a computation to fail to be unique. This phenomenon is known as indeterminacy. Consider, for example, the case where two processors are responsible for adding up a set of numbers. One way to do this would be to share a variable used to collect the sum. Depending on how this variable is handled, indeterminacy could result. Suppose, for example, that a process assigns the sum variable to a local variable, adds a quantity to the local variable, then writes back the result to the sum:

```
local = sum;
local += quantity to be added;
sum = local;
```

Suppose two processes follow this protocol, and each has its own local variable. Suppose that sum is initially 0 and one process is going to add 4 and the other 5. Then depending on how the individual instructions are interleaved, the result could be 9 or it could be 4 or 5. This can be demonstrated by a state diagram. The combined programs are

```
A1:  localA = sum;          B1: localB = sum;
A2:  localA += 4;           B2: localA += 5;
A3:  sum = localA;          B3: sum = localB;
A4:                         B4:
```

The state is:

(ip of A, ip of B, localA, localB, sum)

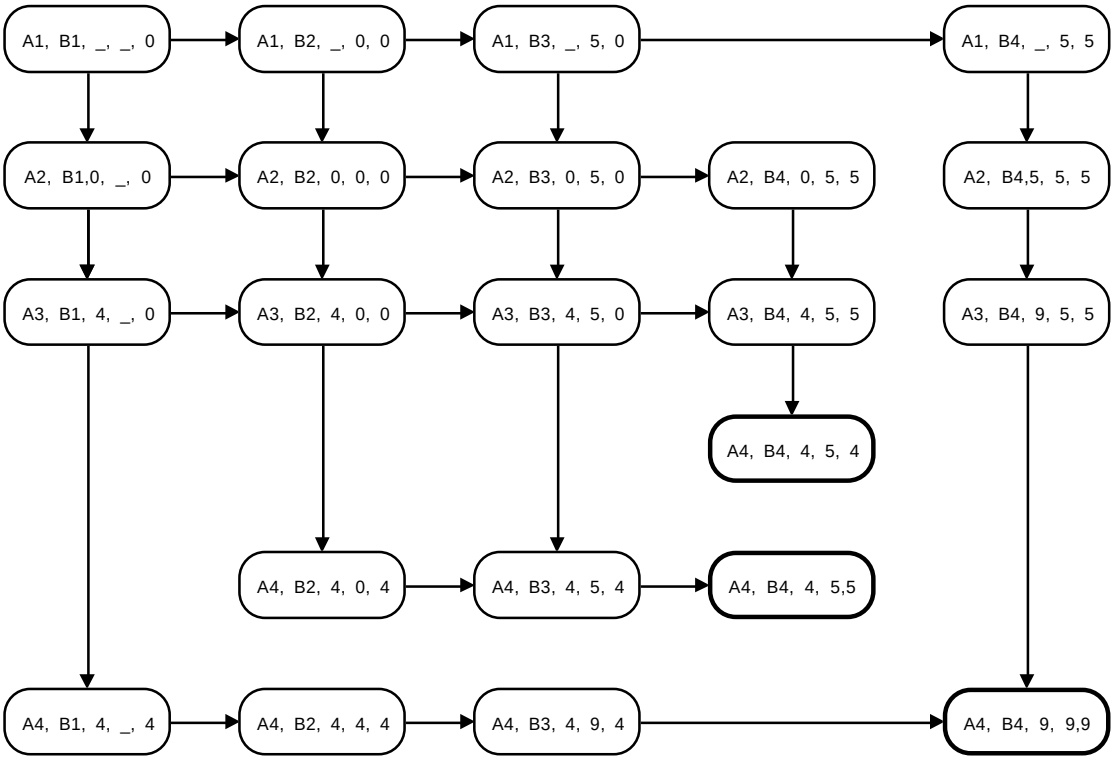


Figure 308: A state diagram exhibiting indeterminacy

Note that in the state diagram there are three states having the ip components A4, B4, each with a different value for sum. One state corresponds to the two processes having done their operations one after the other, and the other two corresponds to one process writing while the other is still computing. Thus we have *indeterminacy* in the final value of sum. This phenomenon is also called a *race condition*, as if processes A and B were racing with each other to get access to the shared variable.

Race conditions and indeterminacy are generally undesirable because they make it difficult to reason about parallel programs and to prove their correctness. Various abstractions and programming techniques can be used to reduce the possibilities for indeterminacies. For example, if we use a purely functional programming model, there are no shared variables and no procedural side-effects. Yet there can still be a substantial amount of parallelism, as seen in previous examples, such as the function *map*.

Semaphores

Computer scientists have worked extensively on the problem of using processor resources efficiently. They have invented various abstractions to not only allow a process to go to sleep, but also to wake it when the bit has been set, and not sooner. One such abstraction is known as a semaphore. In one form, a semaphore is an object that contains a positive integer value. The two methods for this object are called P and V. When P is done on the semaphore, if the integer is positive, it is lowered by one (P is from a Dutch word meaning "to lower") and the process goes on. If it is not positive, however, the process is put to sleep until a state is reached in which lower can be done without making the integer negative. The only way that such a state is reached is by another process executing the V ("to raise") operation. If no process is sleeping for the semaphore, the V operation simply increments the integer value by one. If, on the other hand, at least one process is sleeping, one of those processes is chosen for awakening and the integer value remains the same (i.e. the net effect is as if the value had been raised and then lowered, the latter by the sleeping process that was not able to lower it earlier). The exact order for awakening is not specified. Most often, the process sleeping for the longest time is awakened next, i.e. a queue data structure is used.

The following program fragments show the use of a semaphore for signaling.

Semaphore S's integer value is 0 initially

Process A:	Process B:
`A1: ....`	`B1: ....`
`A2: V(S);`	`B2: P(S)`
`A3: ....`	`B3: ....`

The state-transition diagram is similar to the case using an integer flag. The main difference is that no looping is shown. If the semaphore value is 0, process B simply cannot proceed. The components of each state are:

(ip of A, ip of B, Semaphore value)

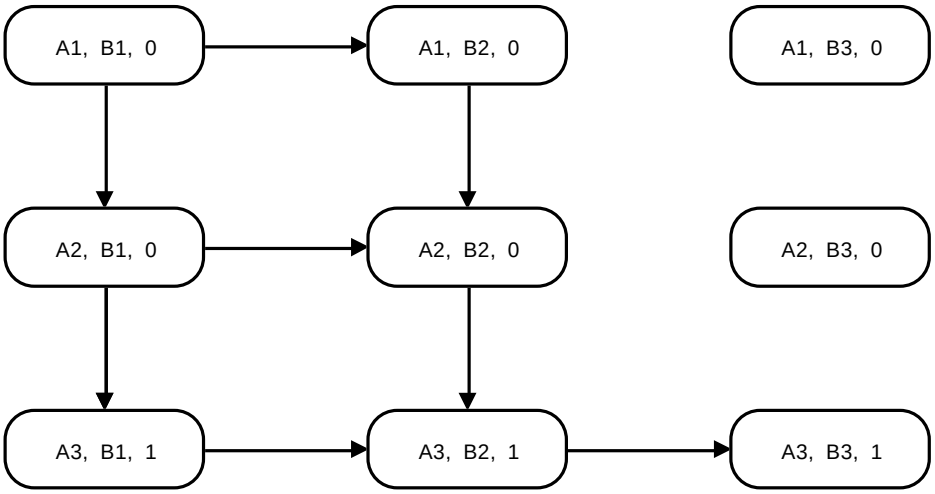


Figure 309: State diagram for signaling using a semaphore

Above we saw a flag and a semaphore being used to *synchronize* one process to another. Another type of control that is often needed is called *mutual exclusion*. In contrast to synchronization, this form is symmetric. Imagine that there is some data to which two processes have access. Either process can access the data, but only one can access it at a time. The segment of code in which a process accesses the data is called a *critical section*. A semaphore, initialized to 1 rather than 0, can be used to achieve the mutual exclusion effect.

Semaphore S's integer value is 1 initially

Process A:

```
A1: ....
A2: P(S);
A3: critical section
A4: V(S)
A5: ....
```

Process B:

```
B1: ....
B2: P(S)
B3: critical section
B4: V(S)
B5: ....
```

A useful extension of the semaphore concept is that of a *message queue* or *mailbox*. In this abstraction, what was a non-negative integer in the case of the semaphore is replaced by a queue of messages. In effect, the semaphore value is like the length of the queue. A process can send a message to another through a common mailbox. For example, we can extend the P operation, which formerly waited for the semaphore to have a positive value, to return the next message on the queue:

P(S, M); sets M to the next message in S

If there is no message in S when this operation is attempted, the process doing P will wait until there is a message. Likewise, we extend the V operation to deposit a message:

V(S, M); puts message M into S

Mailboxes between UNIX® processes can be simulated by an abstraction known as a *pipe*. A pipe is accessed in the same way a file is accessed, except that the pipe is not really a permanent file. Instead, it is just a buffer that allows bytes to be transferred from one process to another in a disciplined way. As with an input stream, if the pipe is currently empty, the reading process will wait until something is in the pipe. In UNIX®, a single table common to all processes is used to hold descriptors for open files. Pipes are also stored in this table. A pipe is created by a system call that returns two file descriptors, one for each end of the pipe. The user of streams in C++ does not typically see the file descriptors. These are created dynamically and held in the state of the stream object. By appropriate low-level coding, it is possible to make a stream object connect to a pipe instead of a file.

Exercises

1 •• Construct a state-transition diagram for the case of a semaphore used to achieve mutual exclusion. Observe that no state is reached in which both processes are in their critical sections.

2 ••• Show that a message queue can be constructed out of ordinary semaphores. (Hint: Semaphores can be used in at least two ways: for synchronization and for mutual exclusion.)

3 ••• Using a linked-list to implement a message queue, give some examples of what can go wrong if the access to the queue is not treated as a critical section.

14.8 Distributed Memory Multiprocessors

A distributed memory multiprocessor provides an alternative to shared memory. In this type of design, processors don't contend for a common memory. Instead, each processor is closely linked with its own memory. Processors must communicate by sending *messages* to one another. The only means for a processor to get something from another processor's memory is for the former to send a message to the latter indicating its desire. It is up to the receiving processor to package a response as a message back to the requesting processor.

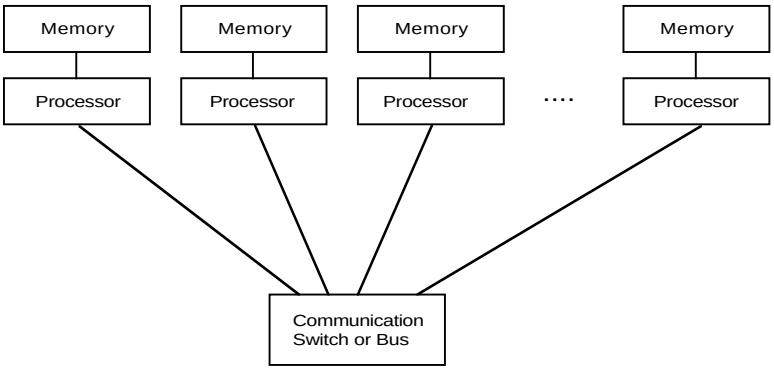


Figure 310: Distributed Memory Multiprocessor

What possible advantage could a distributed memory multiprocessor have? For one thing, there is no contention for a common memory module. This improves performance. On the other hand, having to go through an intermediary to get information is generally slower than in the case of shared memory.

A related issue is known as *scalability*. Assuming that there is enough parallelism inherent in the application, one might wish to put a very large number of processors to work. Doing this exacerbates the problem of memory contention: only one processor can access a single memory module at a time. The problem can be alleviated somewhat by adding more memory modules to the shared memory configuration. The bottleneck then shifts to the processor-memory communication mechanism. If, for example, a bus interconnection is used, there is a limit to the number of memory transactions per unit time, as determined by the bus technology. If the number of processors and memory modules greatly exceeds this limit, there will be little point in having multiple units. To overcome the bus limitation, a large variety of multi-stage interconnect switches have been proposed. A typical example of such a switch is the *butterfly* interconnection, as shown below.

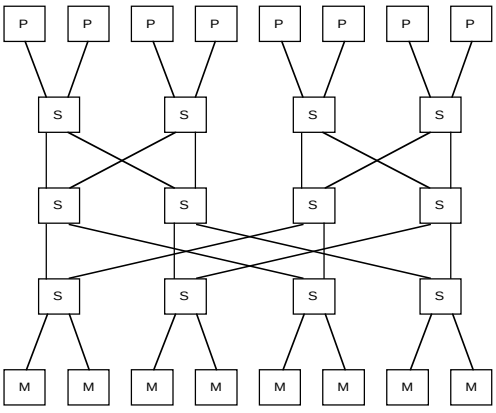


Figure 311: Butterfly interconnection for shared memory

A major problem with such interconnection schemes is known as *cache coherency*. If processors cache data in memory, there is the possibility that one processor could have a datum in its cache while another processor updates the memory version of that datum. In order to manage this possibility, a cached copy would have to be invalidated whenever such an update occurs. Achieving this invalidation requires communication from the memory to the processor, or some kind of vigilance (called "snooping") in the interconnection switch. On the other hand, a distributed memory computer does not incur the cache coherency problem, but rather trades this for a generally longer access time for remote data.

Distributed memory multiprocessors bet on a high degree of *locality*, in the sense that most accesses will be made to local memory, in order to achieve performance. They also mask latency of memory accesses by switching to a different thread when a remote access is needed.

Networking and Client-Server Parallelism

Another type of distributed memory computer is a *computer network*. In this type of system, several computers, each capable of operating stand-alone, are interconnected. The interconnection scheme is typically in the nature of a bus, such as an Ethernet®. These networks were not originally conceived for purposes of parallel computing as much as they were for information sharing. However, they can be used for parallel computing effectively if the degree of locality is very high.

A common paradigm for parallel computing in a network is known as *client-server*. In this model, long-term processes running on selected nodes of the network provide a service for processes on other nodes. The latter, called *clients*, communicate with the server by sending it messages. Many servers and clients can be running concurrently. A given node might support both client and server processes. Furthermore, a server of one function might be a client of others. The idea is similar to object-oriented computing, except that the objects in this case run concurrently with each other, rather than being dormant until the next message is received.

From the programmer's point of view, communication between client and server takes place using data abstractions such as *sockets*. The socket concept permits establishment of a connection between a client and a server by the client knowing the server's address in the network. Knowing the address allows the client to send the server an initial connect message. After connection is established, messages can be exchanged without the need, by the program, to use the address explicitly. Of course, the address is still used implicitly to route the message from one node to another. A common form of socket is called the *stream socket*. In this form, once the connection is established, reading and writing appears to be an ordinary i/o operation, similar to using streams in C++.

14.9 Speedup and Efficiency

The *speedup* of a parallel computation is the ratio of the sequential execution time to the parallel execution time for the same problem. Conventions vary as to whether the same algorithm has to be used in both time computaions. The most fair comparison is probably to compare the speed of the parallel algorithm to that of the best-known sequential algorithm.

An ideal speedup factor would be equal to the number of processors, but rarely is this achieved. The reasons that prevent it from being achieved are: (i) not all problems lend themselves to parallel execution; some are inherently sequential, and (ii) there is overhead involved in creating parallel tasks and communicating between them.

The idea of efficiency attempts to measure overhead; it is defined as the amount of work actually done in a parallel computation divided by the product of the run-time and the number of processors, the latter product being regarded as the *effort*

A thorough study of speedup issues is beyond the scope of this book. Suffice it to say that difficulty in attaining acceptable speedup on a large class of problems has been one of the main factors in the slow acceptance of parallel computation. The other factor is the extra programming effort typically required to achieve parallel execution. The next chapter mentions Amdahl's law, which is one attempt at quantifying an important issue, namely that some algorithms might be inherently sequential.

14.9 Chapter Review

Define the following terms:

cellular automaton
client-server
distributed-memory
efficiency
expression parallelism
fork
map
MIMD
pipeline processing
process
scheduling
shared-memory
semaphore
SIMD
speedup
stream-parallelism
vector processor

14.10 Further Reading

Elwyn Berlekamp, John Conway, and Richard Guy. *Winning ways for your mathematical plays*, vol. 2, Academic Press, 1982. [Chapter 25 discusses the game of Life, including self-reproduction and simulation of switching circuits. Moderate.]

A.W. Burks (ed.), *Essays on cellular automata*, University of Illinois Press, 1970. [A collection of essays by Burks, von Neumann, Thatcher, Holland, and others. Includes an analysis of von Neumann's theory of self-reproducing automata. Moderate.]

Vipin Kumar, et al., *Introduction to parallel computing – Design and Analysis of Algorithms*, Benjamin/Cummings, 1994. [Moderate.]

Michael J. Quinn, *Parallel computing –theory and practice*, McGraw-Hill, 1994. [An introduction, with emphasis on algorithms. Moderate.]

William Poundstone, *The recursive universe*, Contemporary books, Inc., Chicago, Ill., 1985. [A popular discussion of the game of Life and physics. Easy.]

Evan Tick, *Parallel logic programming*, MIT Press, 1991. [Ties together parallelism and logic programming. Moderate.]

Stephen Wolfram, Cellular automata and complexity, Addison-Wesley, 1994. [A collection of papers by Wolfram. Difficult.]

15. Limitations to Computing

15.1 Introduction

We conclude the book with a discussion of what is very difficult or even not possible in various models, or in computation as a whole.

In the course up to this point, we have concentrated on what is "doable", with what models, what speed, etc. Other questions asked in computer science focus on what is not doable, or not doable in less than certain time, etc. Here we briefly survey the kinds of things that are known.

15.2 Algorithm lower bounds

For any given problem, the growth rate of an algorithm to solve the problem will have an absolute minimum. Unlike upper bounds on algorithmic performance, which are demonstrated for specific algorithms, lower bounds are for problems, and span all possible algorithms. Thus establishing them is much more difficult.

An example of a fairly accessible lower-bound argument is for sorting using two-way comparisons between data objects without regard to their internal structure (this excludes radix sort, which relies on a radix representation of the objects). For this problem, we have a lower bound of

$$\Omega (N \log N)$$

to sort N data objects, meaning that N log N is O(the growth rate for *any* algorithm for the problem). This means that sorts such as heapsort and mergesort are "optimal" to within a constant factor.

In order to derive this bound, we take an abstract view of what it means to compute by comparisons. Each time a program makes a comparison, that provides new information about the data. Since each comparison has two outcomes, we can cast the information states about the data as *an information tree*. The root of the tree represents the state where we are completely ignorant of the data since we have not yet made any comparisons. From a general information state, a comparison will take us to one of two other states. At some point, we will have made sufficiently many comparisons to have determined the original order of the data, in other words to determine which *permutation* of the data is needed to put the elements into sorted order. Such an unambiguous state is a *leaf* of the information tree. Thus there will be one leaf for each permutation of the data. The sequence of comparisons made during a given run of the program will correspond to a path from root to leaf. A lower bound on the time to sort is thus the minimum length of all such paths.

We know the following: For N data elements, there are P = N! permutations, hence P leaves. Secondly, in a binary tree with P leaves, *some* path must have length at least log P. With all paths less than log P, we would have fewer than $2^{\log P} = P$ leaves, a contradiction. The worst case sorting time is thus at least log P = log N!. An approximation known as Stirling's formula says

$$\log N! \approx k\, N \log N + \text{lower order constants}$$

for appropriate k, which is what we need for our lower bound.

There are many areas where tight lower bounds are not known, e.g. in the area of NP-complete problems that includes the traveling salesman problem, proposition logic satisfiability, and many others.

15.3 Limitations on Specific Classes of Machines

For example, finite-state machines are limited in the kinds of functions they can compute. Some examples of languages that are not finite-state acceptable are:

matched-parenthesis languages

$\{0^n1^n \mid n \text{ a natural number}\} = \{\lambda, 01, 0011, 000111, ...\}$

$\{1^p \mid p \text{ a prime number}\} = \{11, 111, 11111, 1111111, ...\}$

The first two of these can be represented by a context-free grammar, but context-free grammars have their own limitations, e.g. $\{0^n1^n2^n \mid n \text{ a natural number}\}$ is not generated by any context-free grammar. All of the above are acceptable by Turing machines.

To see that the second language mentioned above is not acceptable by a finite-state acceptor, suppose to the contrary that it is. Let *N* be the number of states of a machine accepting $\{0^n1^n \mid n \text{ a natural number}\}$. Now consider the input 0^N1^N. Since the machine makes 2N > N state transitions in the process of reading this input, some state must recur. (This is called the **pigeon-hole principle**; if one puts *M* pigeons (states occurring in a sequence) into *N* < *M* holes (states in the machine), then some of holes must have more than one pigeon in them.) Suppose that q is such a repeated state. If q occurs both times during the processing of 0^n, then we have a problem: it would then also be the case that 0^p1^n would be accepted for some p < n. Similarly, we also have a contradiction if a repeated state occurs during processing of 1^n. Finally, if q occurs the first time during the processing of 0^n, and the second time during the processing of 1^n, then we get a contradiction in that a sequence where some 1's precede 0's is also acceptable.

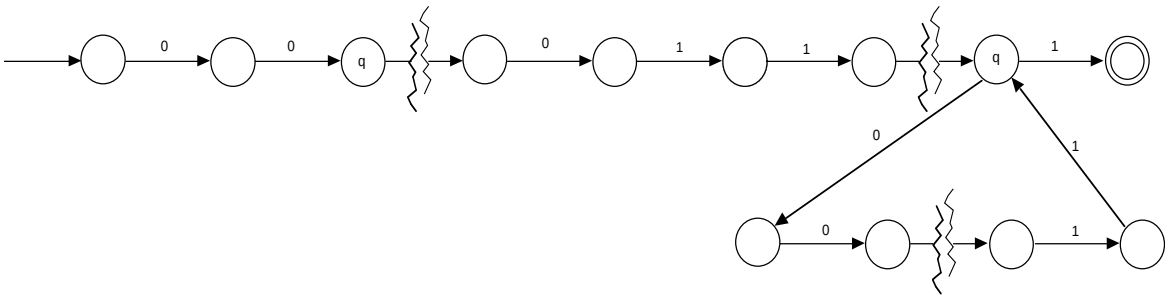


Figure 312: ***Anomalous acceptance arising from repeated states in a long sequence***

15.4 The Halting Problem

Consider any class of reasonably-powerful machines (Turing machines, programs written in C++, partial recursive functions, etc.) It is provable that there are well-defined functions that cannot be computed in the framework. Consider the program-analysis function H where

H(P) = 1 if program P diverges with P as input

H(P) diverges otherwise

[Here "diverges" means "never halts".] H is well defined. Many programs can accept programs as input (compilers, text formatters, etc.) Most of these sorts of programs always halt, and could be modified to halt with answer 1 if desired.

The problem is that there is no program for the particular function H. To see why, suppose P is the program for H. Then H(P) either diverges or it doesn't. But since P is a program for H, the first line of the definition says that if H(P) diverges, then H(P) halts. The second line says that if H(P) does not diverge, then H(P) diverges. Thus we have an absurdity. We are forced to conclude that our assumption was wrong that a program for H exists. This is usually summarized as

The halting problem is unsolvable.

Principle of Diagonalization

The proof that the halting problem is unsolvable relies on the *Principle of Diagonalization*. A classical mathematical use of this principle is to prove that the set of all subsets of the natural numbers is not countable. As promised in the first chapter, we will now discuss this proof. Let 2^{ω} represent the set of all subsets of ω. Suppose that 2^{ω}

were countable, i.e. there is an enumeration of it. Let the sets in the enumeration be $\{S_0, S_1, S_2, ... \}$. Now we construct a new subset of ω, call it S. The definition of S is

$$S = \{ n \in \omega \mid n \notin S_n \}$$

That is, S is the set of all natural numbers n such that n is not a member of S_n in the supposed enumeration. The funny thing about S is that it is obviously a subset of ω. Therefore, it should appear in the enumeration. But where does it appear? It must be S_n for some n. However, then we have a problem: For the index n of S_n, is $n \in S_n$ or not? If we assume $n \in S_n$, then we get from the definition of S (a synonym for S_n) that $n \notin S_n$. If we assume $n \notin S_n$, then we get from the same definition that $n \in S_n$. We are snookered either way. Therefore we must backtrack to where we made an assumption, and that was that 2^{ω} could be enumerated. Thus 2^{ω} is not countable.

The reason this is called diagonalization is that if we create an infinite two-dimensional array, listing the elements of along the top and the supposed enumeration along the side, then put a 1 for the entry in column n, row S_m, if $n \in S_m$ and put a 0 if $n \notin S_m$. Consider the diagonal of this array. Flip the values on the diagonal, i.e. interchange 0 and 1. If this diagonal is flattened out as a if a row, its 0's and 1's correspond to a subset of ω, and it should therefore appear as one of the rows. However it cannot, because it differs from every row in at least one position, according to its construction.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
S_0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…
S_1	1	**0**	1	0	1	0	1	0	1	0	1	0	1	0	1	…
S_2	0	1	**0**	1	0	1	0	1	0	1	0	1	0	1	0	…
S_3	1	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	…
S_4	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	…
S_5	1	0	1	0	0	**0**	1	0	1	0	1	0	1	0	0	…
S_6	0	1	1	0	0	0	**1**	0	0	0	0	0	0	0	0	…
S_7	1	1	1	0	0	0	0	**0**	0	0	0	0	0	0	0	…
S_8	0	0	0	1	0	1	0	1	**0**	1	0	1	0	0	0	…
S_9	1	0	0	1	0	0	0	0	1	**0**	1	0	1	0	1	…
S_{10}	0	1	0	1	0	0	0	1	1	0	**1**	0	1	1	0	…
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	

Figure 313: Diagonalization construction

The row obtained by inverting the diagonal, 11100101110... , cannot appear as any row in the enumeration.

In terms of Turing machines, we can list all of the Turing machines in the same way we listed the subsets of the natural numbers. Across the top, we list the inputs to those Turing machines, which as we have already discussed, can be enumerated. The diagonal of the Turing machine array corresponds to those machines that halt on the tape with their own description. By inverting that diagonal, we have a representation of the function in the halting problem.

Although the definition of H might seem peculiar at first, other desirable, less peculiar functions can be similarly shown to be unsolvable. For example, it turns out that we don't need to rely on P being fed itself as input. We can *fix the input*, for example, to be a totally blank tape in the case of Turing machines, and still get unsolvability. In this kind of argument, we say that we have *reduced* the original halting problem to the blank-tape halting problem: if the latter is solvable, then the former is as well.

Note that throughout such a discussion, the domain of interest is functions that range over a wide set of inputs (e.g. all programs of a given model). We cannot prove unsolvability of the halting question for a particular single program on a particular input. Such a function would be a constant function (with a 0 or 1 answer) and would thus be computable.However, we might not know *which* program to use for it: one that always gives result 1 or one that always gives result 0.

Other Uncomputable Problems

The halting problem may sound very esoteric, something we'd never consider computing. But there are other problems that sound more down-to-earth, but which can be proved unsolvable by either an argument similar to the original halting problem, or by showing that arbitrary Turing machines can be represented in the model:

Busy Beaver Problem

In 1962, T. Rado showed this problem to be uncomputable.

> Fix the alphabet to {1, _} (_ is blank). Let BB(N) be the largest number of 1's that will be printed by any halting N-state Turing machine when started on an all-blank tape. BB is not computable. It can be shown that BB has a faster-growth rate than any computable function. To get some idea of how large BB can be,
>
> BB(5) is at least 4098.
> BB(100) is at least ((((7!)!)!)!)

Domino Problems

In 1961, Hao Wang showed the following to be unsolvable.

> Is there an algorithm for determining whether the infinite half-plane can be completely tiled (such that regions adjacent between dominos have the same color) with an input set of types of 4-sided "dominoes"?

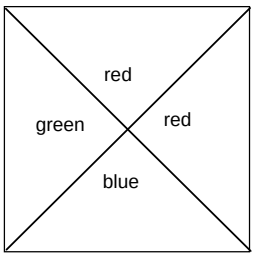


Figure 314: One of a finite set of dominoes to be used to tile an infinite plane

An algorithm for the domino problem would yield an algorithm for telling whether a TM halts. (The dominoes can simulate the possible successive configurations of a tape on the TM.)

For the quarter-plane, the question is "semi-decidable". It can be shown that the quarter-plane is tileable iff every finite "prefix" square is tileable. If some prefix square is untileable, this can be shown by enumerating all possible potential tilings of the square. We can thus determine whether the quarter plane is untileable by an expanding series of enumerations.

Rice's Theorem

The ultimate heartbreak of unsolvability is captured generally in an elegant theorem known as Rice's theorem. It asserts:

> Any property of computable functions that holds for some functions, but not for all, is undecidable for the corresponding programs of those functions. (includes halting, specialized halting, equivalence to a given function, etc.)

Basically, this means that if we are dealing with models that are general enough to represent any computable function, we can forget about developing algorithms that analyze those models for any functional property precisely.

15.5 NP-Complete Problems

Around 1970, computer scientists Edmunds, Cook, and Karp made a series of observations about various computational problems. Some of these problems seemed to be "easy", in the sense that there is a known polynomial-time algorithm for them. Others seemed "hard" in that no polynomial-time algorithm was known (the best known

algorithms were exponential-time). It was observed that there were inter-conversions among the hard algorithms. Indeed, there were conversion algorithms that ran in polynomial time that would convert one hard problem into another. The implication was that if we could find a polynomial algorithm for one such problem, we would immediately have a polynomial time algorithm for several others, according to the known conversions. Eventually the class of hard problems called "NP complete" (NPC) was formulated. This was a set of problems all of that are interconvertible by polynomial time algorithms. The "NP" stands for "non-deterministic polynomial" and alludes to the fact that these problems can be run on a non-deterministic Turing machine in polynomial time. Non-deterministic Turing machines are related to Turing machines in the same way that non-deterministic finite-state machines are related to finite-state machines. Unfortunately, unlike finite-state machines, there is no known general simulation of a polynomial-time non-deterministic Turing machine that runs in polynomial time on a deterministic one. To the present, the question of whether the problems in the class NPC yield to any polynomial algorithm is open. Lacking is either a demonstration of such an algorithm or a proof that no such algorithm exists. Either one of these would resolve the question of the large family of NP complete problems. Meanwhile, simply showing a problem to be a member of this family is an indication that the problem is computationally quite difficult in terms of its complexity. See [Garey and Johnson 1975] or most any algorithms text for further discussion.

15.6 Amdahl's Law

Many computer scientists are interested in possible ways of speeding up computation by doing several things simultaneously, or "in parallel". Gene Amdahl observed that the success of such attempts is limited by the amount of the workload that is inherently sequential. Suppose that a program has a number N basic steps such that a fraction F of those steps cannot be done in parallel with any other steps, while the remainder of the steps can be done in parallel on P processors. Thus the time to do those remaining steps can be sped up by a factor of P.

The overall time to do the work with the parallel processing capability, counting 1 time unit per step, is:

$$F*N + (1-F)*N / P$$

The speedup is the time to do the work on 1 processor (N) divided by this time. In other words, the speedup is

$$\frac{1}{F + (1-F)/P}$$

or

$$\text{speedup} \leq \frac{1}{F + (1-F)/S_{max}}$$

where S_{max} is the maximum speedup if everything were done in parallel.

This puts some limits on speedup that are perhaps surprising. If F were only 10%, then a speedup of at most 10 can be obtained. If F = 50%, then a speedup of at most 2 can be obtained, even if S_{max} were infinity. In other words, if a very fast technique is applicable to only half of the work, then the entire job will never be reduced by more than a factor of 2. Therefore parallel computers are best applicable to tasks that have a low degree of inherent sequentiality.

Amdahl's law is applicable to any speedup method, not just parallel computing. It indicates one of the uses of program profiling: effort to improve a program's performance is most effective if concentrated on the parts that take the most time.

15.7 The Glitch Phenomenon

We saw earlier how it is sometimes convenient to leave the digital abstraction. For example, we used a three-state buffer to achieve multiplexing via a bus. The outputs of such a buffer are 0, 1, and high-impedance, which is neither 0 or 1. When dealing with the interface between synchronous (clocked) systems and asynchronous (unclocked) systems, it should be understood that the digital abstraction is not quite sufficient.

Up to this point, we assumed that the signal to be gated into a latch was never changing during the interval in which the clock is changing. But when interfacing to the outside world, we have no control over when the sampled signal changes. This can cause problems with latch behavior. The following diagram shows such an interface, typically called a "synchronizer".

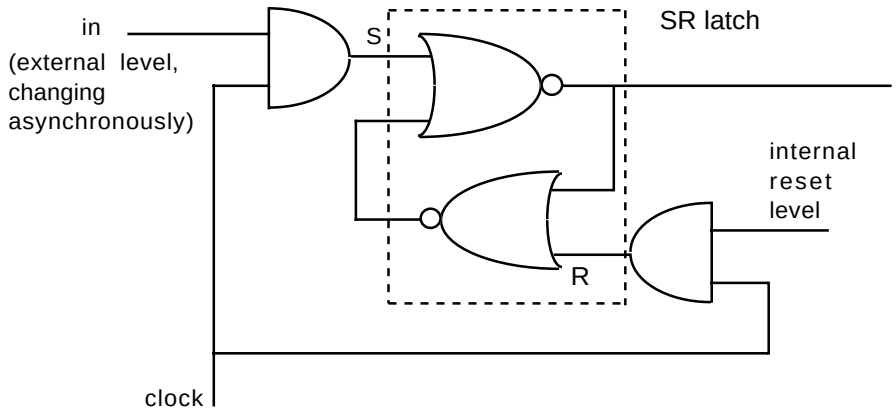


Figure 315: An attempted synchronizer

The set-reset latch formed of NOR gates has an input that is the AND of the clock and an unclocked external signal. The purpose of the circuit is to determine if the external signal has been raised since the latch was last reset. When the external is changing at the same time as the clock, the outputs of the top-left AND gate can be "runts", something in between 0 and 1. A runt can be sufficiently in between 0 and 1 that it causes the latch not to switch to the set state, but rather to "hang" mid way.

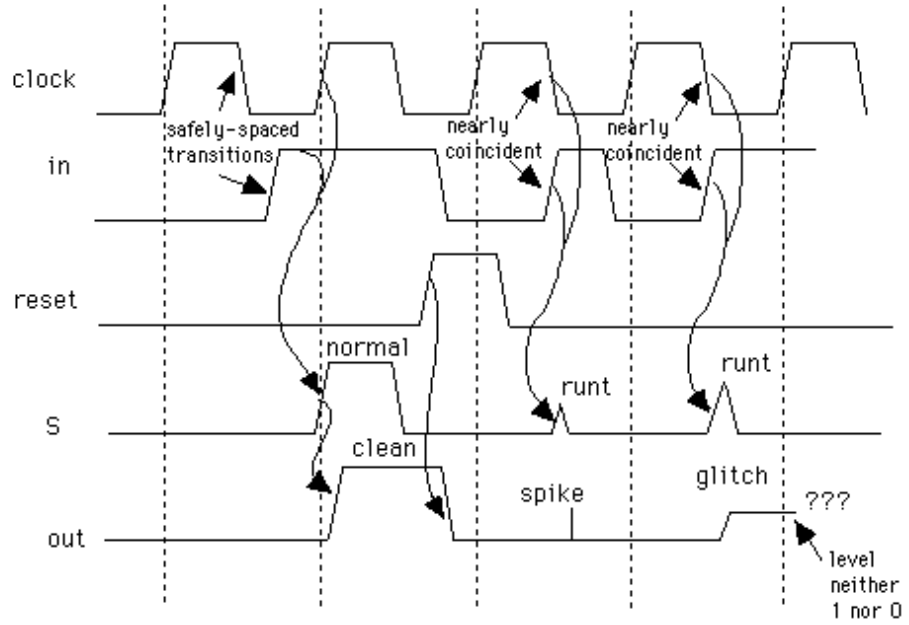


Figure 316: Possible timing of the interplay of the signals in the synchronizer

Metastable Behavior in Physical Terms

Three events of interest are shown. In the first, the external input rises between clock changes, and is detected when the clock rises. In the second, the falling clock ANDed with the rising external input produces a small runt pulse that is not enough to change the state of the latch. A spike (perhaps harmless) results in the output of the latch. In the third event, a similar coincidence creates a runt pulse with enough energy to bring the latch into a mid-way or metastable state, resulting in a glitch: a signal that is neither a digital 0 nor a 1.

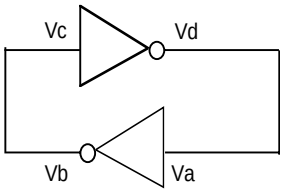


Figure 317: Simplified situation representing the opposing NOR gates in the latch

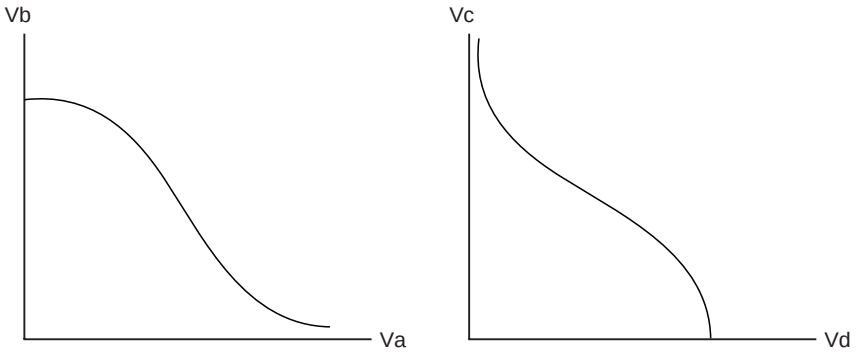


Figure 318: Response curves of the two inverters in isolation (the curves are the same, but one is rotated and reflected

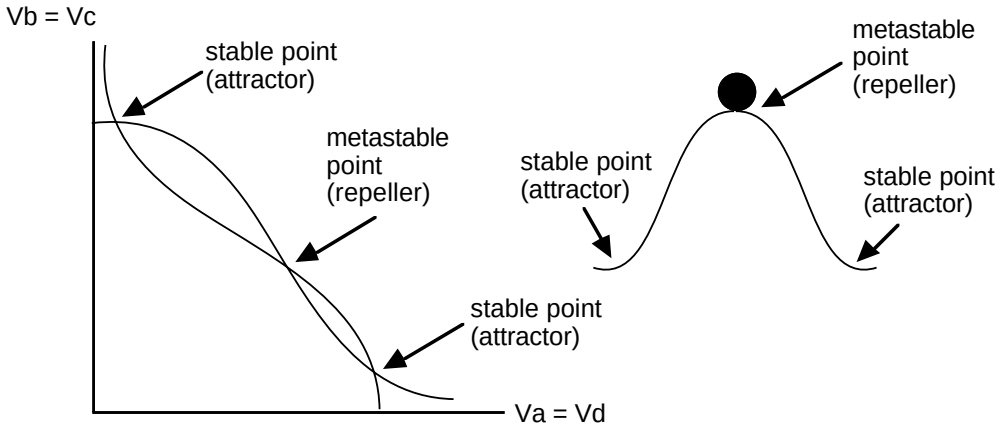


Figure 319: *Left*: Superimposed response curves reflect the interconnection with Vb = Vc and Va = Vd. The stable or equilibrium points are those where the curves intersect. The metastable point is a "repeller" in the sense that points on either side to move away from the repeller and toward one of the two attractors. *Right*: The ball-on-hill analogy. The latch operation is analogous to pushing the ball from one stable point to the other. With insufficient energy, the ball will not make it over the hill and will roll back down. With just the right amount of energy, the ball will sit atop the hill indefinitely (i.e. glitch).

Solution for the glitch problem: A perfectly glitch-free system cannot be built if asynchronous interaction is necessary. By waiting sufficiently long before sampling the output of a synchronizer latch, an adequately-small probability of a glitch can be achieved, e.g. one with expected time-to-failure of several centuries.

15.8 Chapter Review

Define the following terms:

Amdahl's law
diagonalization
glitch
halting problem
lower bound
metastable state
pigeon-hole principle
synchronizer

15.9 Further Reading

Michael R. Garey and David S. Johnson, *Computers and intractability*, W.H. Freeman, San Francisco, 1979.

David Harel, *Algorithmics – The Spirit of Computing*, Addison-Wesley, Reading, Massachusetts, 1987. [Further discussion of unsolvable problems. Easy to moderate.]

J.L. Hennessy and D.A. Patterson. *Computer Architecture - A quantitative approach*, Morgan Kauffman, 1991. [Moderate.]

R. Machlin and Q.F. Stout, *The complex behavior of simple machines*, in S. Forrest (ed.), *Emergent Computation*, MIT Press, 1991. [Recent results on the Busy Beaver problem.]

T. Rado, *On non-computable functions*, Bell Systems Technical Journal, May 1962, pp 877-884. [Introduces the Busy Beaver problem.]

H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967. [Rice's theorem and related arguments. Difficult.]

J.F. Wakerly. *Digital design principles and practices*, Prentice-Hall, 1990. [Easy to moderate.]

Hao Wang, *Proving theorems by pattern recognition*, Bell Systems Technical Journal, 40, pp. 1-42, 1961. [Relationship of a domino problem to the halting problem.]

Index