Reverse Compilation Techniques

by

Cristina Cifuentes

Bc.App.Sc – Computing Honours, QUT (1990)
Bc.AppSc – Computing, QUT (1989)

Submitted to the School of Computing Science
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

at the

QUEENSLAND UNIVERSITY OF TECHNOLOGY

July 1994

Statement of Original Authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed ....

Date ....

QUEENSLAND UNIVERSITY OF TECHNOLOGY
DOCTOR OF PHILOSOPHY THESIS EXAMINATION

CANDIDATE NAME	Cristina Nicole Cifuentes
CENTRE/RESEARCH CONCENTRATION	Programming Languages and Systems
PRINCIPAL SUPERVISOR	Professor K J Gough
ASSOCIATE SUPERVISOR	Professor W J Caelli
THESIS TITLE	Reverse Compilation Techniques

Under the requirements of PhD regulation 9.2, the above candidate was examined orally by the Faculty. The members of the panel set up for this examination recommend that the thesis be accepted by the University and forwarded to the appointed Committee for examination.

Name.................................. Signature....................................
Panel Chairperson (Principal Supervisor)

Name.................................. Signature....................................
Panel Member

Name.................................. Signature....................................
Panel Member

****

Under the requirements of PhD regulation 9.15, it is hereby certified that the thesis of the above-named candidate has been examined. I recommend on behalf of the Examination Committee that the thesis be accepted in fulfilment of the conditions for the award of the degree of Doctor of Philosophy.

Name.................................. Signature....................................
Examination Committee Chairperson

Date..

Reverse Compilation Techniques

by
Cristina Cifuentes

Abstract

Techniques for writing reverse compilers or decompilers are presented in this thesis. These techniques are based on compiler and optimization theory, and are applied to decompilation in a unique way; these techniques have never before been published.

A decompiler is composed of several phases which are grouped into modules dependent on language or machine features. The front-end is a machine dependent module that parses the binary program, analyzes the semantics of the instructions in the program, and generates an intermediate low-level representation of the program, as well as a control flow graph of each subroutine. The universal decompiling machine is a language and machine independent module that analyzes the low-level intermediate code and transforms it into a high-level representation available in any high-level language, and analyzes the structure of the control flow graph(s) and transform them into graphs that make use of high-level control structures. Finally, the back-end is a target language dependent module that generates code for the target language.

Decompilation is a process that involves the use of tools to load the binary program into memory, parse or disassemble such a program, and decompile or analyze the program to generate a high-level language program. This process benefits from compiler and library signatures to recognize particular compilers and library subroutines. Whenever a compiler signature is recognized in the binary program, all compiler start-up and library subroutines are not decompiled; in the former case, the routines are eliminated from the final target program and the entry point to the `main` program is used for the decompiler analysis, in the latter case the subroutines are replaced by their library name.

The presented techniques were implemented in a prototype decompiler for the Intel i80286 architecture running under the DOS operating system, **dcc**, which produces target C programs for source **.exe** or **.com** files. Sample decompiled programs, comparisons against the initial high-level language program, and an analysis of results is presented in Chapter 9.

Chapter 1 gives an introduction to decompilation from a compiler point of view, Chapter 2 gives an overview of the history of decompilation since its appearance in the early 1960s, Chapter 3 presents the relations between the static binary code of the source binary program and the actions performed at run-time to implement the program, Chapter 4 describes the phases of the front-end module, Chapter 5 defines data optimization techniques to analyze the intermediate code and transform it into a higher-representation, Chapter 6 defines control structure transformation techniques to analyze the structure of the control flow graph and transform it into a graph of high-level control structures, Chapter 7 describes the back-end module, Chapter 8 presents the decompilation tool programs, Chapter 9 gives an overview of the implementation of **dcc** and the results obtained, and Chapter 10 gives the conclusions and future work of this research.

Parts of this thesis have been published or have been submitted to international journals. Two papers were presented at the *XIX Conferencia Latinoamericana de Informática* in 1993: "A Methodology for Decompilation"[CG93], and "A Structuring Algorithm for Decompilation"[Cif93]. The former paper presented the phases of the decompiler as described in Chapter 1, Section 1.3, the front-end (Chapter 4), initial work on the control flow analysis phase (Chapter 6), and comments on the work done with **dcc**. The latter paper presented the structuring algorithms used in the control flow analysis phase (Chapter 6). One journal paper, "Decompilation of Binary Programs"[CG94], has been accepted for publication by *Software – Practice & Experience*; this paper gives an overview of the techniques used to build a decompiler (summaries of Chapters 4, 5, 6, and 7), how a signature generator tool can help in the decompilation process (Chapter 8, Section 8.2), and a sample decompiled program by **dcc** (Chapter 9). Two papers are currently under consideration for publication in international journals. "Interprocedural Data Flow Decompilation"[Cif94a] was submitted to the *Journal of Programming Languages* and describes in full the optimizations performed by the data flow analyzer to transform the low-level intermediate code into a high-level representation. "Structuring Decompiled Graphs"[Cif94b] was submitted to *The Computer Journal* and gives the final, improved method of structuring control flow graphs (Chapter 6), and a sample decompiled program by **dcc** (Chapter 9).

The techniques presented in this thesis expand on earlier work described in the literature. Previous work in decompilation did not document on the interprocedural register analysis required to determine register arguments and register return values, the analysis required to eliminate stack-related instructions (i.e. `push` and `pop`), or the structuring of a generic set of control structures. Innovative work done for this research is described in Chapters 5, 6, and 8. Chapter 5, Sections 5.2 and 5.4 illustrate and describe nine different types of optimizations that transform the low-level intermediate code into a high-level representation. These optimizations take into account condition codes, subroutine calls (i.e. interprocedural analysis) and register spilling, eliminating all low-level features of the intermediate instructions (such as condition codes and registers) and introducing the high-level concept of expressions into the intermediate representation. Chapter 6, Sections 6.2 and 6.6 illustrate and describe algorithms to structure different types of loops and conditional, including multi-way branch conditionals (e.g. `case` statements). Previous work in this area has concentrated in the structuring of loops, few papers attempt to structure 2-way conditional branches, no work on multi-way conditional branches is described in the literature. This thesis presents a complete method for structuring all types of structures based on a predetermined, generic set of high-level control structures. A criterion for determining the generic set of control structures is given in Chapter 6, Section 6.4. Chapter 8 describes all tools used to decompile programs, the most important tool is the signature generator (Section 8.2) which is used to determine compiler and library signatures in architectures that have an operating system that do not share libraries, such as the DOS operating system.

Acknowledgments

The feasibility of writing a decompiler for a contemporary machine architecture was raised by Professors John Gough and Bill Caelli in the early 1990s. Since this problem appeared to provide a challenge in the areas of graph and data flow theory, I decided on pursuing a PhD with the aim at determining techniques for the reverse compilation of binary programs. This thesis is the answer to the many questions asked about how to do it; and yes, it is feasible to write a decompiler.

I would like to acknowledge the time and resources provided by a number of people in the computing community. Professor John Gough provided many discussions on data flow analysis, and commented on each draft chapter of this thesis. Sylvia Willie lent me a PC and an office in her lab in the initial stages of this degree. Pete French provided me with an account on a Vax BSD 4.2 machine in England to test a Vax decompiler available on the network. Jeff Ledermann rewrote the disassembler. Michael Van Emmerik wrote the library signature generator program, generated compiler and library signatures for several PC compilers, ported **dcc** to the DOS environment, and wrote the interactive user interface for **dcc**. Jinli Cao translated a Chinese article on decompilation to English while studying at QUT. Geoff Olney proof-read each chapter, pointed out inconsistencies, and suggested the layout of the thesis. I was supported by an Australian Postgraduate Research Award (APRA) scholarship during the duration of this degree.

Jeff Ledermann and Michael Van Emmerik were employed under Australian Research Council ARC grant No. A49130261.

This thesis was written with the LaTeX document preparation system. All figures were produced with the xfig facility for interactive generation of figures under X11.

Cristina Cifuentes
June 1994

Contents

List of Figures

Chapter 1

Introduction to Decompiling

Compiler-writing techniques are well known in the computer community; decompiler-writing techniques are not as well yet known. Interestingly enough, decompiler-writing techniques are based on compiler-writing techniques, as explained in this thesis. This chapter introduces the subject of decompiling by describing the components of a decompiler and the environment in which a decompilation of a binary program is done.

1.1 Decompilers

A decompiler is a program that reads a program written in a machine language – the source language – and translates it into an equivalent program in a high-level language – the target language (see Figure 1-1). A decompiler, or reverse compiler, attempts to reverse the process of a compiler which translates a high-level language program into a binary or executable program.

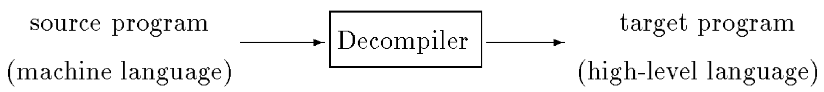


Figure 1-1: A Decompiler

Basic decompiler techniques are used to decompile binary programs from a wide variety of machine languages to a diversity of high-level languages. The structure of decompilers is based on the structure of compilers; similar principles and techniques are used to perform the analysis of programs. The first decompilers appeared in the early 1960s, a decade after their compiler counterparts. As with the first compilers, much of the early work on decompilation dealt with the translation of scientific programs. Chapter 2 describes the history of decompilation.

1.2 Problems

A decompiler writer has to face several theoretical and practical problems when writing a decompiler. Some of these problems can be solved by use of heuristic methods, others cannot be determined completely. Due to these limitations, a decompiler performs automatic program translation of *some* source programs, and semi-automatic program translation of other source programs. This differs from a compiler, which performs an automatic program translation of all source programs. This section looks at some of the problems involved.

1.2.1 Recursive Undecidability

The general theory of computability tries to solve decision problems, that is, problems which inquire on the existence of an algorithm for deciding the truth or falsity of a whole class of statements. If there is a positive solution, an algorithm must be given; otherwise, a proof of non-existence of such an algorithm is needed, in this latter case we say that the problem is unsolvable, undecidable, or non-computable. Unsolvable problems can be partially computable if an algorithm can be given that answers yes whenever the program halts, but otherwise loops forever.

In the mathematical world, an abstract concept has to be described and modelled in terms of mathematical definitions. The abstraction of the algorithm has to be described in terms of what is called a Turing machine. A Turing machine is a computing machine that prints symbols on a linear tape of infinite length in both directions, possesses a finite number of states, and performs actions specified by means of quadruples based upon its current internal configuration and current symbol on the tape. Figure 1-2 shows a representation of a Turing machine.

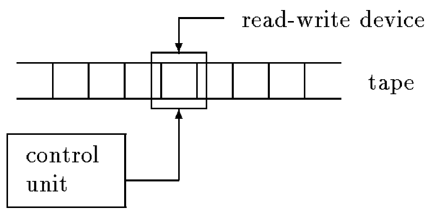


Figure 1-2: Turing Machine Representation

The **halting problem** for a Turing machine Z consists of determining, of a given instantaneous description α, whether or not there exists a computation of Z that begins with α. In other words, we are trying to determine whether or not Z will halt if placed in an initial state. It has been proved that this problem is recursively unsolvable and partially computable[Dav58, GL82].

Given a binary program, the separation of data from code, even in programs that do not allow such practices as self-modifying code, is equivalent to the halting problem, since it is unknown in general whether a particular instruction will be executed or not (e.g. consider the code following a loop). This implies that the problem is partially computable, and therefore an algorithm can be written to separata data from code in some cases, but not all.

1.2.2 The von Neumann Architecture

In von Neumann machines, both data and instructions are represented in the same way in memory. This means that a given byte located in memory is not known to be data or instruction (or both) until that byte is fetched from memory, placed on a register, and used as data or instruction. Even on segmented architectures where data segments hold only data information and code segments hold only instructions, data can still be stored in a code segment in the form of a table (e.g. `case` tables in the Intel architecture), and instructions can still be stored in the form of data and later executed by interpreting such instructions. This latter method was used as part of a Modula-2 compiler for the PC that interprets an intermediate code for an abstract stack machine. The intermediate code was stored as data and the offset for a particular procedure was pointed to by `es:di`[GCC+92].

1.2.3 Self-modifying code

Self-modifying code refers to instructions or preset data that are modified during execution of the program. A memory byte location for an instruction can be modified during program execution to represent another instruction or data. This method has been used throughout the years for different purposes. In the 60s and 70s, computers did not have much memory, and thus it was difficult to run large programs. Computers with a maximum of 32Kb and 64Kb were available at the time. Since space was a constraint, it had to be utilized in the best way. One way to achieve this was by saving bytes in the executable program, by reusing data locations as instructions or vice versa. In this way, a memory cell held an instruction at one time, and data or another instruction at another time. Also, instructions modified other instructions once they were not needed, and therefore executed different code next time the program executed that section of code.

Nowadays there are few memory limitations on computers, and therefore self-modifying code is not used as often. It is still used though when writing encrypting programs or virus code (see Section 1.2.5). A sample self-modifying code for the Intel architecture is given in Figure 1-3. The `inst` definition is modified by the `mov` instruction to the data bytes `E920`. After the move, `inst` is treated as yet another instruction, which is now `0E9h 20h`; that is, an unconditional jump with offset `20h`. Before the `mov`, the `inst` memory location held a `9090`, which would have been executed as two `nop` instructions.

```
        ...                   ; other code
        mov [inst], E920      ; E9 == jmp, 20 == offset
    inst db 9090              ; 90 == nop
```

Figure 1-3: Sample self-modifying Code

1.2.4 Idioms

An idiom or idiomatic expression is a sequence of instructions which form a logical entity, and which taken together have a meaning that cannot be derived by considering the primary meanings of the instructions[Gai65].

For example, the multiplication or division by powers of 2 is a commonly known idiom: multiplication is performed by shifting to the left, while division is performed by shifting to the right. Another idiom is the way long variables are added. If the machine has a word size of 2 bytes, a long variable has 4 bytes. To add two long variables, the low two bytes are added first, followed by the high two bytes, taking into account the carry from the first addition. These idioms and their meaning are illustrated in Figure 1-4. Most idioms are known in the computer community, but unfortunately, not all of them are widely known.

```
shl ax, 2              add ax, [bp-4]
                       adc dx, [bp-2]

                  ⇓

mul ax, 4         add dx:ax, [bp-2]:[bp-4]
```

Figure 1-4: Sample Idioms

1.2.5 Virus and Trojan "tricks"

Not only have virus programs been written to trigger malicious code, but also hide this code by means of tricks. Different methods are used in viruses to hide their malicious code, including self-modifying and encrypting techniques.

Figure 1-5 illustrates code for the Azusa virus, which stores in the stack a new return address for a procedure. As can be seen, the segment and offset addresses of the virus code are pushed onto the stack, followed by a return far instruction, which transfers control to the virus code. When disassembling code, most disassemblers would stop at the far return instruction believing an end of procedure has been met; which is not the case.

```
         ...             ; other code, ax holds segment SEG value
SEG:00C4  push ax         ; set up segment
SEG:00C5  mov ax, 0CAh    ; ax holds an offset
SEG:00C8  push ax         ; set up offset
SEG:00C9  retf            ; jump to virus code at SEG:00CA
SEG:00CA  ...             ; virus code is here
```

Figure 1-5: Modify the return address

One frequently used trick is the use of self-modifying code to modify the target address offset of an unconditional jump which has been defined as data. Figure 1-6 illustrates the relevant code of the Cia virus before execution. As can be seen, `cont` and `conta` define data items 0E9h and 0h respectively. During execution of this program, procX modifies the contents of conta with the offset of the virus code, and after procedure return, the instruction jmp virusOffset (0E9h virusOffset) is executed, treating data as instructions.

```
start:
        call  procX           ; invoke procedure
cont    db    0E9h            ; opcode for jmp
conta   dw    0
procX:
        mov   cs:[conta],virusOffset
        ret
virus:
        ...                   ; virus code
end.
```

Figure 1-6: Self-modifying Code Virus

Virus code can be present in an encrypted form, and decryption of this code is only performed when needed. A simple encryption/decryption mechanism is performed by the xor function, since two xors of a byte against the same constant are equivalent to the original byte. In this way, encryption is performed with the application of one xor through the code, and decryption is performed by xoring the code against the same constant value. This virus is illustrated in Figure 1-7, and was part of the LeprosyB virus.

```
encrypt_decrypt:
    mov  bx, offset virus_code    ; get address of start encrypt/decrypt
  xor_loop:
    mov  ah, [bx]                 ; get the current byte
    xor ah, encrypt_val           ; encrypt/decrypt with xor
    mov [bx], ah                  ; put it back where we got it from
    inc bx                        ; bx points to the next byte
    cmp bx, offset virus_code+virus_size  ; are we at the end?
    jle xor_loop                  ; if not, do another cycle
    ret
```

Figure 1-7: Self-encrypting Virus

Recently, polymorphic mutation is used to encrypt viruses. The idea of this virus is to self-generate sections of code based on the regularity of the instruction set. Figure 1-8 illustrates the encryption engine of the Nuke virus. Here, a different key is used each time around the encryption loop (ax), and the encryption is done by means of an xor instruction.

```
     Encryption_Engine:
07AB         mov    cx,770h
07AE         mov    ax,7E2Ch
07B1 encryption_loop:
07B1         xor    cs:[si],ax
07B4         inc    si
07B5         dec    ah
07B7         inc    ax
07B8         loop   encryption_loop
07BA         retn
```

Figure 1-8: Self-generating Virus

In general, virus programs make use of any flaw in the machine language set, self-modifying code, self-encrypting code, and undocumented operating system functions. This type of code is hard to disassemble automatically, given that most of the modifications to instructions/data are done during program execution. In these cases, human intervention is required.

1.2.6 Architecture-dependent Restrictions

Most of the contemporary machine architectures make use of a prefetch buffer to fetch instructions while the processor is executing instructions. This means that instructions that are prefetched are stored in a different location from the instructions that are already in main memory. When a program uses self-modifying code to attempt to modify an instruction in memory, if the instruction has already been prefetched, it is modified in memory but not in the pipeline buffer; therefore, the initial, unmodified instruction is executed. This example can be seen in Figure 1-9. In this case, the `jmpDef` data definition is really an instruction, `jmp codeExecuted`. This definition appears to be modified by the previous instruction, `mov [jumpDef],ax`, which places two `nop` instructions in the definition of `jmpDef`. This would mean that the code at `codeNotExecuted` is executed, displaying "Hello world!" and exiting. When running this program on an i80386 machine, "Share and Enjoy!" is displayed. The i80386 has a prefetch buffer of 4 bytes, so the `jmpDef` definition is not modified because it has been prefetched, and therefore the jump to `codeExecuted` is done, and "Share and Enjoy!" is displayed. This type of code cannot be determined by normal straight line step debuggers, unless a complete emulation of the machine is done.

1.2.7 Subroutines included by the compiler and linker

Another problem with decompilation is the great number of subroutines introduced by the compiler and the number of routines linked in by the linker. The compiler will always include start-up subroutines that set up its environment, and runtime support routines whenever required. These routines are normally written in assembler and in most cases are untranslatable into a higher-level representation. Also, most operating systems do not provide a mechanism for sharing libraries, consequently, binary programs are self-contained

```
           mov  ax, 9090        ; 90 == nop
           mov  [jumpDef], ax
jmpDef     db 0EBh 09h          ; jmp codeExecuted
codeNotExecuted:
           mov  dx, helloStr
           mov  ah,09
           int  21              ; display string
           int  20              ; exit
codeExecuted:
   mov  dx, shareStr
           mov  ah, 09
           int  21              ; display string
           int  20              ; exit

shareStr   db "Share and Enjoy!", 0Dh, 0Ah, "$"
helloStr   db "Hello World!", 0Dh, 0Ah, "$"
```

Figure 1-9: Architecture-dependent Problem

and library routines are bound into each binary image. Library routines are either written in the language the compiler was written in or in assembler. This means that a binary program contains not only the routines written by the programmer, but a great number of other routines linked in by the linker. For example, a program written in C to display "hello world" and compiled on a PC has over 25 different subroutines in the binary program. A similar program written in Pascal and compiled on the PC generates more than 40 subroutines in the executable program. Out of all these routines, the reverse engineer is normally interested in just the one initial subroutine; the main program.

1.3 The Phases of a Decompiler

Conceptually, a decompiler is structured in a similar way to a compiler, by a series of *phases* that transform the source machine program from one representation to another. The typical phases of a decompiler are shown in Figure 1-10. These phases represent the logical organization of a decompiler. In practice, some of the phases will be grouped together, as seen in Section 1.4.

A point to note is that there is no lexical analysis or scanning phase in the decompiler. This is due to the simplicity of machine languages; all tokens are represented by bytes or bits of a byte. Given a byte, it is not possible to determine whether that byte forms the start of a new token or not; for example, the byte `50` could represent the opcode for a `push ax` instruction, an immediate constant, or an offset to a data location.

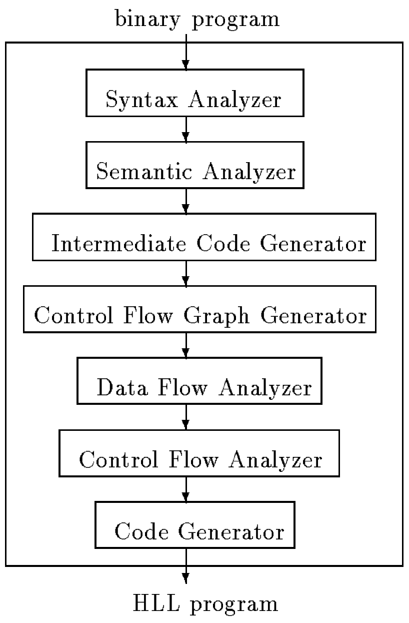


Figure 1-10: Phases of a Decompiler

1.3.1 Syntax Analysis

The parser or syntax analyzer groups bytes of the source program into grammatical phrases (or sentences) of the source machine language. These phrases can be represented in a parse tree. The expression `sub cx, 50` is semantically equivalent to `cx := cx - 50`. This latter expression can be represented in a parse tree as shown in Figure 1-11. There are two phrases in this expression: `cx - 50` and `cx := <exp>`. These phrases form a hierarchy, but due to the nature of machine language, the hierarchy will always have a maximum of two levels.

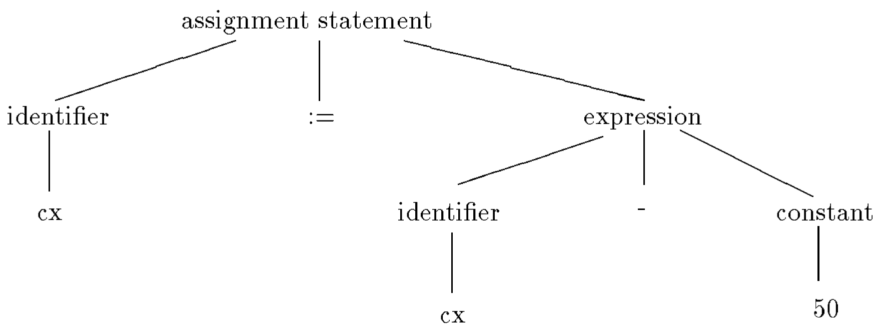


Figure 1-11: Parse tree for cx := cx - 50

The main problem encountered by the syntax analyzer is determining what is data and what is an instruction. For example, a `case` table can be located in the code segment and it is unknown to the decompiler that this table is data rather than instructions, due to the architecture of the von Neumann machine. In this case, instructions cannot be parsed sequentially assuming that the next byte will always hold an instruction. Machine dependent heuristics are required in order to determine the correct set of instructions. Syntax analysis is covered in Chapter 4.

1.3.2 Semantic Analysis

The semantic analysis phase checks the source program for the semantic meaning of groups of instructions, gathers type information, and propagates this type across the subroutine. Given that binary programs were produced by a compiler, the semantics of the machine language is correct in order for the program to execute. It is rarely the case in which a binary program does not run due to errors in the code generated by a compiler. Thus, semantic errors are not present in the source program unless the syntax analyzer has parsed an instruction incorrectly or data has been parsed instead of instructions.

In order to check for the semantic meaning of a group of instructions, idioms are looked for. The idioms from Figure 1-4 can be transformed into semantically equivalent instructions: the multiplication of `ax` by 4 in the first case, and the addition of long variables in the second case. `[bp-2]:[bp-4]` represent a long variable for that particular subroutine, and `dx:ax` holds the value of a long variable temporarily in this subroutine. These latter registers do not have to be used as a long register throughout the subroutine, only when needed.

Type propagation of newly found types by idiomatic expressions is done throughout the graph. For example, in Figure 1-4, two stack locations of a subroutine were known to be used as a long variable. Therefore, anywhere these two locations are used or defined independently must be converted to a use or definition of a long variable. If the following two statements are part of the code for that subroutine

```
asgn [bp-2], 0
asgn [bp-4], 14h
```

the propagation of the long type on `[bp-2]` and `[bp-4]` would merge these two statements into one that represents the identifiers as longs, thus

```
asgn [bp-2]:[bp-4], 14h
```

Finally, semantic errors are normally not produced by the compiler when generating code, but can be found in executable programs that run on a more advanced architecture than the one that is under consideration. For example, say we are to decompile binaries of the i80286 architecture. The new i80386 and i80486 architectures are based on this i80286 architecture, and their binary programs are stored in the same way. What is different in these new architectures, with respect to the machine language, is the use of more registers and instructions. If we are presented with an instruction

```
add ebx, 20
```

the register identifier `ebx` is a 32-bit register not present in the old architecture. Therefore, although the instruction is syntactically correct, it is not semantically correct for the machine language we are decompiling, and thus an error needs to be reported. Chapter 4 covers some of the analysis done in this phase.

1.3.3 Intermediate Code Generation

An explicit intermediate representation of the source program is necessary for the decompiler to analyse the program. This representation must be easy to generate from the source program, and must also be a suitable representation for the target language. The semantically equivalent representation illustrated in Section 1.3.1 is ideal for this purpose: it is a three-address code representation in which an instruction can have at most three operands. These operands are all identifiers in machine language, but can easily be extended to expressions to represent high-level language expressions (i.e. an identifier is an expression). In this way, a three-address representation is used, in which an instruction can have at most three expressions. Chapter 4 describes the intermediate code used by the decompiler.

1.3.4 Control Flow Graph Generation

A control flow graph of each subroutine in the source program is also necessary for the decompiler to analyse the program. This representation is suited for determining the high-level control structures used in the program. It is also used to eliminate intermediate jumps that the compiler generated due to the offset limitations of a conditional jump in machine language. In the following code

```
      ...       ; other code
      jne x     ; x <= maximum offset allowed for jne
      ...       ; other code
   x: jmp y     ; intermediate jump
      ...       ; other code
   y: ...       ; final target address
```

label `x` is the target address of the conditional jump `jne x`. This instruction is limited by the maximum offset allowed in the machine architecture, and therefore cannot execute a conditional jump to `y` on the one instruction; it has to use an intermediate jump instruction. In the control flow graph, the conditional jump to `x` is replaced with the final target jump to `y`.

1.3.5 Data Flow Analysis

The data flow analysis phase attempts to improve the intermediate code, so that high-level language expressions can be found. The use of temporary registers and condition flags is eliminated during this analysis, as these concepts are not available in high-level languages. For a series of intermediate language instructions

```
asgn ax, [bp-0Eh]
asgn bx, [bp-0Ch]
asgn bx, bx * 2
asgn ax, ax + bx
asgn [bp-0Eh], ax
```

the final output should be in terms of a high-level expression

```
asgn [bp-0Eh], [bp-0Eh] + [bp-0Ch] * 2
```

The first set of instructions makes use of registers, stack variables and constants; expressions are in terms of identifiers, with a maximum tree level of 2. After the analysis, the final instruction makes use of stack variable identifiers, `[bp-0Eh]`, `[bp-0Ch]`, and an expression tree of 3 levels, `[bp-0Eh] := [bp-0Eh] + [bp-0Ch] * 2`. The temporary registers used by the machine language to calculate the high-level expression, `ax` and `bx`, along with the loading and storing of these registers, has been eliminated. Chapter 5 presents an algorithm to perform this analysis, and to eliminate other intermediate language instructions such as `push` and `pop`.

1.3.6 Control Flow Analysis

The control flow analyzer phase attempts to structure the control flow graph of each subroutine of the program into a generic set of high-level language constructs. This generic set must contain control instructions available in most languages; such as looping and conditional transfers of control. Language-specific constructs should not be allowed. Figure 1-12 shows two sample control flow graphs: an `if..then..else` and a `while()`. Chapter 6 presents an algorithm for structuring arbitrary control flow graphs.

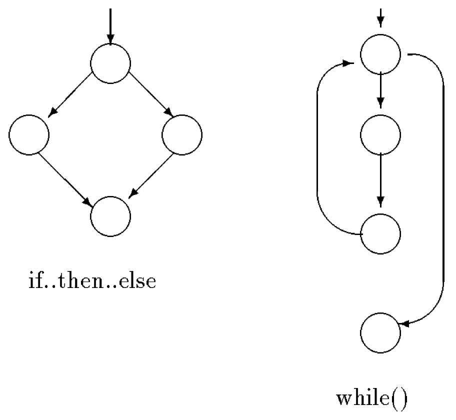


Figure 1-12: Generic Constructs

1.3.7 Code Generation

The final phase of the decompiler is the generation of target high-level language code, based on the control flow graph and intermediate code of each subroutine. Variable names are selected for all local stack, argument, and register-variable identifiers. Subroutine names are also selected for the different routines found in the program. Control structures and intermediate instructions are translated into a high-level language statement.

For the example in Section 1.3.5, the local stack identifiers `[bp-0Eh]` and `[bp-0Ch]` are given the arbitrary names `loc2` and `loc1` respectively, and the instruction is translated to say the C language as

```
  loc2 = loc2 + (loc1 * 2);
```

Code generation is covered in Chapter 7.

1.4 The Grouping of Phases

The decompiler phases presented in Section 1.3 are normally grouped in the implementation of the decompiler. As shown in Figure 1-13, three different modules are distinguished: **front-end**, **udm**, and **back-end**.

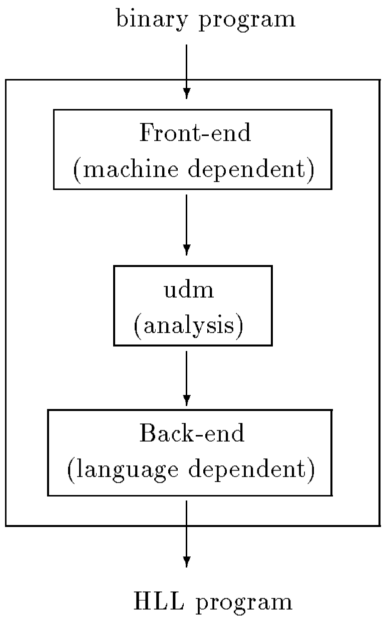


Figure 1-13: Decompiler Modules

The **front-end** consists of those phases that are machine and machine-language dependent. These phases include lexical, syntax and semantic analyses, and intermediate code and control flow graph generation. As a whole, these phases produce an intermediate, machine-independent representation of the program.

The **udm** is the universal decompiling machine; an intermediate module that is completely machine and language independent, and that performs the core of the decompiling analysis. Two phases are included in this module, the data flow and the control flow analyzers.

Finally, the **back-end** consists of those phases that are high-level or target language dependent. This module is the code generator.

In compiler theory, the grouping of phases is a mechanism used by compiler writers to generate compilers for different machines and different languages. If the back-end of the compiler is rewritten for a different machine, a new compiler for that machine is constructured by using the original front-end. In a similar way, a new front-end for another high-level language definition can be written and used with the original back-end. In practice there are some limitations to this method, inherent to the choice of intermediate code representation.

In theory, the grouping of phases in a decompiler makes it easy to write different decompilers for different machines and languages; by writing different front-ends for different machines, and different back-ends for different target languages. In practical applications, this result is always limited by the generality of the intermediate language used.

1.5 The Context of a Decompiler

In practice, several programs can be used with the decompiler to create the target high-level language program. In general, source binary programs have a relocation table of addresses that are to be relocated when the program is loaded into memory. This task is accomplished by the loader. The relocated or absolute machine code is then disassembled to produce an assembly representation of the program. The disassembler can use help from compiler and library signatures to eliminate the disassembling of compiler start-up code and library routines. The assembler program is then input to the decompiler, and a high-level target program is generated. Any further processing required on the target program, such as converting `while()` loops into `for` loops can be done by a postprocessor. Figure 1-14 shows the steps involved in a typical "decompilation". The user could also be a source of information, particularly when determining library routines and separation of data from instructions. Whenever possible, it is more reliable to use automatic tools. Decompiler helper tools are covered in Chapter 8. This section briefly explains their task.

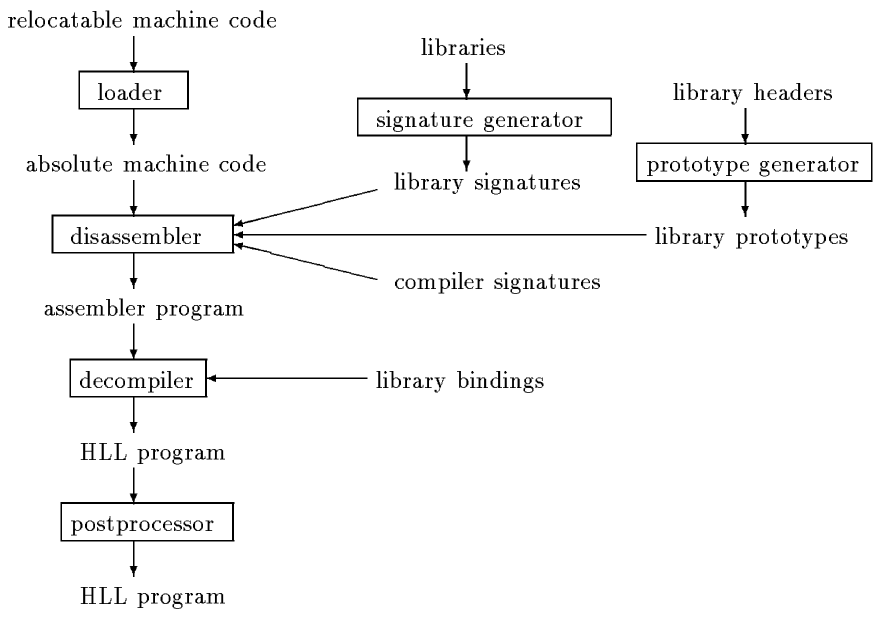


Figure 1-14: A Decompilation System

Loader

The loader is a program that loads a binary program into memory, and relocates the machine code if it is relocatable. During relocation, instructions are altered and placed back in memory.

Signature Generator

A signature generator is a program that automatically determines compiler and library signatures; a binary pattern that uniquely identifies each compiler and library subroutine. The use of these signatures attempts to reverse the task performed by the linker, which links in library and compiler start-up code into the program. In this way, the analyzed program consist only of user subroutines; the ones that the user compiled in the initial high-level language program.

For example, in the compiled C program that displays "hello world" and has over 25 different subroutines in the binary program, 16 subroutines were added by the compiler to set-up its environment, 9 routines that form part of `printf()` were added by the linker, and 1 subroutine formed part of the initial C program.

The use of a signature generator not only reduces the number of subroutines to analyze, but also increases the documentation of the target programs by using library names rather than arbitrary subroutine names.

Prototype Generator

The prototype generator is a program that automatically determines the types of the arguments of library subroutines, and the type of the return value in the case of functions. These prototypes are derived from the library header files, and are used by the decompiler to determine the type of the arguments to library subroutines and the number of such arguments.

Disassembler

A disassembler is a program that transforms a machine language into assembler language. Some decompilers transform assembler programs to a higher representation (see Chapter 2). In these cases, the assembler program has been produced by a disassembler, was written in assembler, or the compiler compiled to assembler.

Library Bindings

Whenever the target language of the decompiler is different to the original language used to compile the binary source program, if the generated target code makes use of library names (i.e. library signatures were detected), although this program is correct, it cannot be recompiled in the target language since it does not use library routines for that language but for another one. The introduction of library bindings solves this problem, by binding the subroutines of one language to the other.

Postprocessor

A postprocessor is a program that transforms a high-level language program into a semantically equivalent high-level program written in the same language. For example, if the target language is C, the following code

```
loc1 = 1;
while (loc1 < 50) {
  /* some code in C */
  loc1 = loc1 + 1;
}
```

would be converted by a postprocessor into

```
for (loc1 = 1; loc1 < 50; loc1++) {
  /* some code in C */
}
```

which is a semantically equivalent program that makes use of control structures available in the C language, but not present in the generic set of structures decompiled by the decompiler.

1.6 Uses of Decompilation

Decompilation is a tool for a computer professional. There are two major areas where decompilation is used: software maintenance and security. In the former area, decompilation is used to recover lost or inaccessible source code, translate code written in an obsolete language into a newer language, structure old code written in an unstructured way (i.e. spaghetti code) into a structured program, migrate applications to a new hardware platform, and debug binary programs that are known to have bugs but for which the source code is unavailable. In the latter area, decompilation is used as a tool to verify the object code produced by a compiler in software-critical systems, since the compiler cannot be trusted in these systems, and to check for the existence of malicious code such as viruses.

1.6.1 Legal Aspects

Several questions have been raised in the last years regarding the legality of decompilation. A debate between supporters of decompilation who claim fair competition is possible with the use of decompilation tools, and the opponents of decompilation who claim copyright is infringed by decompilation, is currently being held. The law in different countries is being modified to determine in which cases decompilation is lawful. At present, commercial software is being sold with software agreements that ban the user from disassembling or decompiling the product. For example, part of the Lotus software agreement reads like this:

> You may not alter, merge, modify or adapt this Sofware in any way including disassembling or decompiling.

It is not the purpose of this thesis to debate the legal implications of decompilation. This topic is not further covered in this thesis.

Chapter 2

Decompilation – What has been done?

Different attempts at writing decompilers have been made in the last 20 years. Due to the amount of information lost in the compilation process, to be able to regenerate high-level language code all of these experimental decompilers have limitations in one way or another, including decompilation of assembly files[Hou73, Fri74, Wor78, Hop78, Bri81] or object files with or without symbolic debugging information[Reu88, PW93], simplified high-level language[Hou73], and the requirement of the compiler's specification[BB91, BB93]. Assembly programs have helpful data information in the form of symbolic text, such as data segments, data and type declarations, subroutine names, subroutine entry point, and subroutine exit statement. All this information can be collected in a symbol table and then the decompiler would not need to address the problem of separating data from instructions, or the naming of variables and subroutines. Object files with debugging information contain the program's symbol table as constructed by the compiler. Given the symbol table, it is easy to determine which memory locations are instructions, as there is a certainty on which memory locations represent data. In general, object files contain more information than binary files. Finally, knowledge of the compiler's specifications is impractical, as these specifications are not normally disclosed by compiler manufacturers.

2.1 Previous Work

Decompilers have been considered a useful software tool since they were first used in the 1960s. At that time, decompilers were used to aid in the program conversion process from second to third generation computers; in this way, manpower would not be spent in the time-consuming task of rewriting programs for the third generation machines. During the 70s and 80s, decompilers were used for the portability of programs, documentation, debugging, re-creation of lost source code, and the modification of existing binaries. In the 90s, decompilers have become a reverse engineering tool capable of helping the user with such tasks as checking software for the existence of illegal code, checking that a compiler generates the right code, and translation of binary programs from one machine to another. It is noted that decompilation is not being used for software piracy or breach of copyright, as the process is incomplete in general, and can be used only as a tool to help develop a task.

The following descriptions illustrate the best-known decompilers and/or research performed into decompiler topics by individual researchers or companies:

D-Neliac decompiler, 1960. As reported by Halstead in [Hal62], the Donnelly-Neliac (D-Neliac) decompiler was produced by J.K.Donnelly and H.Englander at the Navy

Electronics Laboratory (NEL) in 1960. Neliac is an Algol-type language developed at the NEL in 1955. The D-Neliac decompiler produced Neliac code from machine code programs; different versions were written for the Remington Rand Univac M-460 Countess computer and the Control Data Corporation 1604 computer.

D-Neliac proved useful for converting non-Neliac compiled programs into Neliac, and for detecting logic errors in the original high-level program. This decompiler proved the feasibility of writing decompilers.

W.Sassaman, 1966. Sassaman developed a decompiler at TRW Inc., to aid in the conversion process of programs from 2nd to 3rd generation computers. This decompiler took as input symbolic assembler programs for the IBM 7000 series and produced Fortran programs. Binary code was not chosen as input language because the information in the symbolic assembler was more useful. Fortran was a standard language in the 1960s and ran on both 2nd and 3rd generation computers. Engineering applications which involved algebraic algorithms were the type of programs decompiled. The user was required to define rules for the recognition of subroutines. The decompiler was 90% accurate, and some manual intervention was required[Sas66].

This is the first decompiler that makes use of assembler input programs rather than pure binary code. Assembler programs contain useful information in the form of names, macros, data and instructions, which are not available in binary or executable programs, and therefore eliminate the problem of separating data from instructions in the parsing phase of a decompiler.

M.Halstead, 1967. The Lockheed Missiles and Space Company (LMSC) added some enhancements to the Neliac compiler developed at the Navy Electronics Laboratory, to cater for decompilation[Hal67]. The LMSC Neliac decompiler took as input machine code for the IBM 7094 and produced Neliac code for the Univac 1108. It proved successful by decompiling over 90% of instructions and leaving the programmer to decompile the other 10%. This decompiler was used at LMSC and under contract for customers in the U.S.A. and Canada[Hal70].

Halstead analyzed the implementation effort required to raise the percentage of correctly decompiled instructions half way to 100%, and found that it was approximately equal to the effort already spent[Hal70]. This was because decompilers from that time handled straightforward cases, but the harder cases were left for the programmer to consider. In order to handle more cases, more time was required to code these special cases into the decompiler, and this time was proportionately greater than the time required to code simple cases.

Autocoder to Cobol Conversion Aid Program, 1967. Housel reported on a set of commercial decompilers developed by IBM to translate Autocoder programs, which were business data processing oriented, to Cobol. The translation was a one-to-one mapping and therefore manual optimization was required. The size of the final programs occupied 2.1% times the core storage of the original program[Hou73].

This decompiler is really a translation tool of one language to another. No attempt is made to analyze the program and reduce the number of instructions generated. Inefficient code was produced in general.

C.R.Hollander, 1973. Hollander's PhD dissertation[Hol73] describes a decompiler designed around a formal syntax-oriented metalanguage, and consisting of 5 cooperating sequential processes; initializer, scanner, parser, constructor, and generator; each implemented as an interpreter of sets of metarules. The decompiler was a metasystem that defined its operations by implementing interpreters.

The initializer loads the program and converts it into an internal representation. The scanner interacts with the initializer when finding the fields of an instruction, and interacts with the parser when matching source code templates against instructions. The parser establishes the correspondence between syntactic phrases in the source language and their semantic equivalents in the target language. Finally, the constructor and generator generate code for the final program.

An experimental decompiler was implemented to translate a subset of IBM's System/360 assembler into an Algol-like target language. This decompiler was written in Algol-W, a compiler developed at Stanford University, and worked correctly on the 10 programs it was tested against.

This work presents a novel approach to decompilation, by means of a formal syntax-oriented metalanguage, but its main drawback is precisely this methodology, which is equivalent to a pattern-matching operation of assembler instructions into high-level instructions. This limits the amount of assembler instructions that can be decompiled, as instructions that belong to a pattern need to be in a particular order to be recognized; intermediate instructions, different control flow patterns, or optimized code is not allowed. In order for syntax-oriented decompilers to work, the set of all possible patterns would need to be enumerated for each high-level instruction of each different compiler. Another approach would be to write a decompiler for a specific compiler, and make use of the specifications of that compiler; this approach is only possible if the compiler writer is willing to reveal the specifications of his compiler. It appears that Hollander's decompiler worked because the compiler specifications for the Algol-W compiler that he was using were known, as this compiler was written at the University where he was doing this research. The set of assembler instructions generated for a particular Algol-W instruction were known in this case.

B.C.Housel, 1973. Housel's PhD dissertation[Hou73] describes a clear approach to decompilation by borrowing concepts from compiler, graph, and optimization theory. His decompiler involves 3 major phases: partial assembly, analyzer, and code generation.

The partial assembly phase separates data from instructions, builds a control flow graph, and generates an intermediate representation of the program. The analyzer analyzes the program in order to detect program loops and eliminate unnecessary intermediate instructions. Finally, the code generator optimizes the translation of arithmetic expressions, and generates code for the target language.

An experimental decompiler was written for Knuth's MIX assembler (MIXAL), producing PL/1 code for the IBM 370 machines. 6 programs were tested, 88% of the instructions were correct, and the remaining 12% of the instructions required manual intervention[HH73].

This decompiler proved that by using known compiler and graph methods, a decompiler could be written that produced good high-level code. The use of an intermediate *representation made the analysis completely machine independent. The main objection to this methodology is the choice of source language, MIX assembler, not only for the greater amount of information available in these programs, but for being a simplified non-real-life assembler language.*

The Piler System, 1974. Barbe's Piler system attempts to be a general decompiler that translates a large class of source–target language pairs to help in the automatic translation of computer programs. The Piler system was composed of three phases: interpretation, analysis, and conversion. In this way, different interpreters could be written for different source machine languages, and different converters could be written for different target high-level languages, making it simple to write decompilers for different source–target language pairs. Other uses for this decompiler included documentation, debugging aid, and evaluation of the code generated by a compiler.

During interpretation, the source machine program was loaded into memory, parsed and converted into a 3-address microform representation. This meant that each machine instruction required one or more microform instructions. The analyzer determined the logical structure of the program by means of data flow analysis, and modified the microform representation to an intermediate representation. A flowchart of the program after this analysis was made available to users, and they could even modify the flowchart, if there were any errors, on behalf of the decompiler. Finally, the converter generated code for the target high-level language[Bar74].

Although the Piler system attempted to be a general decompiler, only an interpreter for machine language of the GE/Honeywell 600 computer was written, and skeletal converters for Univac 1108's Fortran and Cobol were developed. The main effort of this project concentrated on the analyzer.

The Piler system was a first attempt at a general decompiler for a large class of source and target languages. Its main problem was to attempt to be general enough with the use of a microform representation, which was even lower-level than an assembler-type representation.

F.L.Friedman, 1974. Friedman's PhD dissertation describes a decompiler used for the transfer of mini-computer operating systems within the same architectural class[Fri74]. Four main phases are described: pre-processor, decompiler, code generator, and compiler.

The pre-processor converts assembler code into a standard form (descriptive assembler language). The decompiler takes the standard assembler form, analyses it, and decompiles it into an internal representation, from which FRECL code is then generated by the code generator. Finally, a FRECL compiler compiles this program into machine code for another machine. FRECL is a high-level language for program transport and development; it was developed by Friedman, who also wrote a compiler for it. The decompiler used in this project was an adaptation of Housel's decompiler[Hou73].

Two experiments were performed; the first one involved the transport of a small but self-contained portion of the IBM 1130 Disk Monitor System to Microdata 1600/21; up to 33% manual intervention was required on the input assembler programs. Overall, the amount of effort required to prepare the code for input to the transport system was too great to be completed in a reasonable amount of time; therefore, a second experiment was conducted. The second experiment decompiled Microdata 1621 operating system programs into FRECL and compiled them back again into Microdata 1621 machine code. Some of the resultant programs were re-inserted into the operating system and tested. On average, only 2% of the input assembler instructions required manual intervention, but the final machine program had a 194% increase in the number of machine instructions.

This dissertation is a first attempt at decompiling operating system code, and it illustrates the difficulties faced by the decompiler when decompiling machine-dependent code. Input programs to this transport system require a large amount of effort to be presented in the format required by the system, and the final produced programs appear to be inefficient; both in the size of the program and the time to execute many more machine instructions.

Ultrasystems, 1974. Hopwood reported on a decompilation project at Ultrasystems, Inc., in which he was a consultant for the design of the system[Hop78]. This decompiler was to be used as a documentation tool for the Trident submarine fire control software system. It took as input Trident assembler programs, and produced programs in the Trident High-Level Language (THLL) that was being developed at this company. Four main stages were distinguished: normalization, analysis, expression condensation, and code generation.

The input assembler programs were normalized so that data areas were distinguished with pseudo-instructions. An intermediate representation was generated, and the data analyzed. Arithmetic and logical expressions were built during a process of expression condensation, and finally, the output high-level language program was generated by matching control structures to those available in THLL.

This project attempts to document assembler programs by converting them into high-level language. The fact is, given the time constraints of the project, the expression condensation phase was not coded, and therefore the output programs were hard to read, as several instructions were required for a single expression.

V.Schneider and G.Winiger, 1974. Schneider and Winiger presented a notation for specifying the compilation and decompilation of high-level languages. By defining a context-free grammar for the compilation process (i.e. describe all possible 2-address object code produced from expressions and assignments), the paper shows how this grammar can be inverted to decompile the object code into the original source program[SW74]. Even more, an ambiguous compilation grammar will produce optimal object code, and will generate an unambiguous decompilation grammar. A case study showed that the object code produced by the Algol 60 constructs could not be decompiled deterministically. This work was part of a future decompiler, but further references in the literature about this work were not found.

This work presents, in a different way, a syntax-oriented decompiler[Hol73]; that is, a decompiler that uses pattern matching of a series of object instructions to reconstruct the original source program. In this case, the compilation grammar needs to be known in order to invert the grammar and generate a decompilation grammar. Note that no optimization is possible if it is not defined as part of the compilation grammar.

Decompilation of Polish code, 1977, 1981, 1988. Two papers in the area of decompilation of Polish code into Basic code are found in the literature. The problem arises in connection with highly interactive systems, where a fast response is required to every input from the user. The user's program is kept in an intermediate form, and then "decompiled" each time a command is issued. An algorithm for the translation of reverse Polish notation to expressions is given[BP79].

The second paper presents the process of decompilation as a two step problem: the need to convert machine code to Polish representation, and the conversion of Polish code to source form. The paper concentrates on the second step of the decompilation problem, but yet claims to be decompiling Polish code to Basic code by means of a context-free grammar for Polish notation and a left-to-right or right-to-left parsing scheme[BP81].

This technique was recently used in a decompiler that converted reverse Polish code into spreadsheet expressions[May88]. In this case, the programmers of a product that included a spreadsheet-like component wanted to speed up the product by storing user's expressions in a compiled form, reverse Polish notation in this case, and *decompile* these expressions whenever the user wanted to see or modify them. Parentheses were left as part of the reverse Polish notation to reconstruct the exact same expression the user had input to the system.

The use of the word decompilation in this sense is a misuse of the term. All that is being presented in these papers is a method for re-constructing or deparsing the original expression (written in Basic or Spreadsheet expressions) given an intermediate Polish representation of a program. In the case of the Polish to Basic translators, no explanation is given as to how to arrive at such an intermediate representation given a machine program.

G.L.Hopwood, 1978. Hopwood's PhD dissertation[Hop78] describes a 7-step decompiler designed for the purposes of transferability and documentation. It is stated that the decompilation process can be aided by manual intervention or other external information.

The input program to the decompiler is formatted by a preprocessor, then loaded into memory, and a control flow graph of the program is built. The nodes of this graph represent one instruction. After constructing the graph, control patterns are recognized, and instructions that generate a `goto` statement are eliminated by the use of either node splitting or the introduction of synthetic variables. The source program is then translated into an intermediate machine independent code, and analysis of variable usage is performed on this representation in order to find expressions and eliminate unnecessary variables by a method of forward substitution. Finally, code is generated for each intermediate instruction, functions are implemented to represent operations not supported by the target language, and comments are provided. Manual intervention was required to prepare the input data, provide additional information that the decompiler needed during the translation process, and to make modifications to the target program.

An experimental decompiler was written for the Varian Data machines 620/i. It decompiled assembler into MOL620, a machine-oriented language developed at University of California at Irvine by M.D.Hopwood and the author. The decompiler was tested with a large debugger program, Isadora, which was written in assembler. The generated decompiled program was manually modified to recompile it into machine code, as there were calls to interrupt service routines, self-modifying code, and extra registers used for subroutine calls. The final program was better documented than the original assembler program.

The main drawbacks of this research are the granularity of the control flow graph and the use of registers in the final target program. In the former case, Hopwood chose to build control flow graphs that had one node per instruction; this means that the size of the control flow graph is quite large for large programs, and there is no benefit gained as opposed to using nodes that are basic blocks (i.e. the size of the nodes is dependent on the number of changes of flow of control). In the latter case, the MOL620 language allows for the use of machine registers, and sample code illustrated in Hopwood's dissertation shows that registers were used as part of expressions and arguments to subroutine calls. The concept of registers is not a high-level concept available in high-level languages, and it should not be used if wanting to generate high-level code.

D.A.Workman, 1978. This work describes the use of decompilation in the design of a high-level language suitable for real time training device systems, in particular the F4 trainer aircraft[Wor78]. The operating system of the F4 was written in assembler, and it was therefore the input language to this decompiler. The output language was not determined as this project was to design one, thus code generation was not implemented.

Two phases of the decompiler were implemented: the first phase, which mapped the assembler to an intermediate language and gathered statistics about the source program, and the second phase, which generated a control flow graph of basic blocks, classified the instructions according to their probable type, and analyzed the flow of control in order to determine high-level control structures. The results indicated the need of a high-level language that handled bit strings, supported looping and conditional control structures, and did not require dynamic data structures or recursion.

This work presents a novel use of decompilation techniques, although the input language was not machine code but assembler. A simple data analysis was done by classifying instructions, but did not attempt to analyze them completely as there was no need to generate high-level code. The analysis of the control flow is complete and considers 8 different categories of loops and 2-way conditional statements.

Zebra, 1981. The Zebra prototype was developed at the Naval Underwater Systems Centre in an attempt to achieve portability of assembler programs. Zebra took as input a subset of the ULTRA/32 assembler, called AN/UYK-7, and produced assembler for the PDP11/70. The project was described by D.L.Brinkley in [Bri81].

The Zebra decompiler was composed of 3 passes: a lexical and flow analysis pass, which parsed the program and performed control flow analysis in the graph of basic blocks. The second pass was concerned with the translation of the program to an intermediate form, and the third pass simplified the intermediate representation by eliminating extraneous loads and stores, in much the same way described by Housel[Hou73, HH73].

It was concluded that it was hard to capture the semantics of the program and that decompilation was economically impractical, but it could aid in the transportation process.

This project made use of known technology to develop a decompiler of assembler programs. No new concepts were introduced by this research, but it raised the point that decompilation is to be used as a tool to aid in the solution of a problem, but not as tool that will give all solutions to the problem, given that a 100% correct decompiler cannot be built.

Decompilation of DML programs, 1982. A decompiler of database code was designed to convert a subset of Codasyl DML programs, written with procedural operations, into a relational system with a nonprocedural query specification. An Access Path Model is introduced to interpret the semantic accesses performed by the program. In order to determine how FIND operations implement semantic accesses, a global data flow reaching analysis is performed on the control flow graph, and operations are matched to templates. The final graph structures are remapped into a relational structure. This method depends on the logical order of the objects and a standard ordering of the DML statements[KW82].

Another decompiler of database code was proposed to decompile well-coded application programs into a proposed semantic representation is described in [DS82]. This work was induced by changes in the use requirements of a Database Management System (DBMS), where application programs were written in Cobol-DML. A decompiler of Cobol-DML programs was written to analyse and convert application programs into a model and schema-independent representation. This representation was later modified or restructured to account for database changes. Language templates were used to match against key instructions of a Cobol-DML programs.

In the context of databases, decompilation is viewed as the process of grouping a sequence of statements which represent a query into another (nonprocedural) specification. Data flow analysis is required, but all other stages of a decompiler are not implemented for this type of application.

Forth Decompiler, 1982, 1984. A recursive Forth decompiler is a tool that scans through a compiled dictionary entry and decompiles words into primitives and addresses[Dud82]. Such a decompiler is considered one of the most useful tools in the Forth toolbox[HM84]. The decompiler implements a recursive descent parser so that decompiled words can be decompiled in a recursive fashion.

These works present a deparsing tool rather than a decompiler. The tool recursively scans through a dictionary table and returns the primitives or addresses associated with a given word.

Software Transport System, 1985. C.W.Yoo describes an automatic Software Transport System (STS) that moves assembler code from one machine to another. The process involves the decompilation of an assembler program for machine m_1 to a high-level language, and the compilation of this program in a machine m_2 to assembler. An experimental decompiler was developed on the Intel 8080 architecture; it took as input assembler programs and produced PL/M programs. The recompiled PL/M programs were up to 23% more efficient than their assembler counterpart. An experimental

STS was developed to develop a C cross-compiler for the Z-80 processor. The project encountered problems in the lack of data type in the STS[Yoo85].

The STS took as input an assembler program for machine m_1 and an assembler grammar for machine m_2, and produced an assembler program for machine m_2. The input grammar was parsed and produced tables used by the abstract syntax tree parser to parse the input assembler program and generate an abstract syntax tree (AST) of the program. This AST was the input to the decompiler, which then performed control and data flow analyses, in much the same way described by Hollander[Hol73], Friedman[Fri74], and Barbe[Bar74], and finally generated high-level code. The high-level language was then compiled for machine m_2.

This work does not present any new research into the decompilation area, but it does present a novel approach to the transportation of assembler programs by means of a grammar describing the assembler instructions of the target architecture.

Decomp, 1988. J.Reuter wrote **decomp**, a decompiler for the Vax BSD 4.2 which took as input object files with symbolic information and produced C-like programs. The nature of this decompiler was to port the Empire game to the VMS environment, given that source code was not available. The decompiler is freely available on the Internet[Reu88].

Decomp made use of the symbol table to find the entry points to functions, determine data used in the program, and the names of that data. Subroutines were decompiled one at a time, in the following way: a control flow graph of basic blocks was built and optimised by the removal of arcs leading to intermediate unconditional branches. Control flow analysis was performed in the graph to find high-level control constructs, converting the control flow graph into a tree of generic constructs. The algorithm used by this analysis was taken from the *struct* program, a program that structures graphs produced by Fortran programs, which was based on the structuring algorithm described by B.Baker in [Bak77]. Finally, the generic constructs in the tree were converted to C-specific constructs, and code was generated. The final output programs required manual modifications to place the arguments on the procedure's argument list, and determine that a subroutine returned a value (i.e. was a function). This decompiler was written in about 5 man-months[Reu91].

Sample programs were written and compiled in C in a Vax BSD 4.2 machine, thanks to the collaboration of Pete French[Fre91], who provided me with an account in a Vax BSD 4.2 machine. The resulting C programs are not compilable, but require some hand editing. The programs have the correct control structures, due to the structuring algorithm implemented, and the right data type of variables, due to the embedded symbol table in the object code. The names of library routines and procedures, and the user's program entry point are also known from the symbol table; therefore, no extraneous procedures (e.g. compiler start up code, library routines) are decompiled. The need for a data flow analysis stage is vital, though, as neither expressions, actual arguments, nor function return value are determined. An interprocedural data flow analysis would eliminate much of the hand-editing required to recompile the output programs.

exe2c, 1990. The Austin Code Works sponsored the development of the `exe2c` decompiler, targetted at the PC compatible family of computers running the DOS operating system[Wor91]. The project was announced in April 1990[Gut90], tested by about 20 people, and it was decided that it needed some more work to decompile in C. A year later, the project reached a β operational level[Gut91a], but was never finished[Gut91b]. I was a beta tester of this release.

`exe2c` is a multipass decompiler that consists of 3 programs: `e2a`, `a2aparse`, and `e2c`. `e2a` is the disassembler. It converts executable files to assembler, and produces a commented assembler listing as well. `e2aparse` is the assembler to C front-end processor, which analyzes the assembler file produced by `e2a` and generates `.cod` and `.glb` files. Finally, the `e2c` program translates the files prepared by `a2aparse` and generates pseudo-C. An integrated environment, `envmnu`, is also provided.

Programs decompiled by `exe2c` make use of a header file that defines registers, types and macros. The output C programs are hard to understand because they rely on registers and condition codes (represented by Boolean variables). Normally, one machine instruction is decompiled into one or more C instructions that perform the required operation on registers, and set up condition codes if required by the instruction. Expressions and arguments to subroutines are not determined, and a local stack is used for the final C programs. It is obvious from this output code that a data flow analysis was not implemented in `exe2c`. This decompiler has implemented a control flow analysis stage; looping and conditional constructs are available. The choice of control constructs is generally adequate. `Case` tables are not detected correctly, though. The number and type of procedures decompiled shows that all library routines, and compiler start-up code and runtime support routines found in the program are decompiled. The nature of these routines is normally low-level, as they are normally written in assembler. These routines are hard to decompile as, in most cases, there is no high-level counterpart (unless it is low-level type C code).

This decompiler is a first effort in many years to decompile executable files. The results show that a data flow analysis and heuristics are required to produce better C code. Also, a mechanism to skip all extraneous code introduced by the compiler and to detect library subroutines would be beneficial.

PLM-80 Decompiler, 1991. The Information Technology Division of the Australian Department of Defence researched into decompilation for defence applications, such as maintenance of obsolete code, production of scientific and technical intelligence, and assessment of systems for hazards to safety or security. This work was described by S.T. Hood in [Hoo91].

Techniques for the construction of decompilers using definite-clause grammars, an extension of context-free grammars, in a Prolog environment are described. A Prolog database is used to store the initial assembler code and the recognised syntactic structures of the grammar. A prototype decompiler for Intel 8085 assembler programs compiled by a PLM-80 compiler was written in Prolog. The decompiler produced target programs in Small-C, a subset of the C language. The definite-clause grammar given in this report was capable of recognizing `if..then` type structures, and `while()` loops, as well as static (global) and automatic (local) variables of simple types (i.e. character, integers, and longs). A graphical user interface was written to display the assembler and pseudo-C programs, and to enable the user to assign variable names, and comments. This interface also asked the user for the entry point to the main program, and allowed him to select the control construct to be recognized.

The analysis performed by this decompiler is limited to the recognition of control structures and simple data types. No analysis on the use of registers is done or mentioned. Automatic variables are represented by an indexed variable that represents the stack. The graphical interface helps the user document the decompiled program by means of comments and meaningful variable names. This analysis does not support optimized code.

Decompiler compiler, 1991–1994. A decompiler compiler is a tool that takes as input a compiler specification *and* the corresponding portions of object code, and returns the code for a decompiler; i.e. it is an automatic way of generating decompilers, much in the same way that *yacc* is used to generate compilers[BBL91, BB91, BB94].

Two approaches are described to generate such a decompiler compiler: a logic and a functional programming approach. The former approach makes use of the bidirectionality of logic programming languages such as Prolog, and runs the specification of the compiler backwards to obtain a decompiler[BBL91, BB91, BBL93]. In theory this is correct, but in practice this approach is limited to the implementation of the Prolog interpreter, and therefore problems of strictness and reversibility are encountered[BB92, BB93]. The latter approach is based on the logic approach but makes use of lazy functional programming languages like Haskell, to generate a more efficient decompiler[BBL91, BB91, BBL93]. Even if a non-lazy functional language is to be used, laziness can be simulated in the form of objects rather than lists.

The decompiler produced by a decompiler compiler will take as input object code and return a list of source codes that can be compiled to the given object code. In order to achieve this, an enumeration of all possible source codes would be required, given a description of an arbitrary inherited attribute grammar. It is proved that such an enumeration is equivalent to the Halting Problem[BB92, BB93], and is therefore non-computable. Even further, there is no computable method which takes an attribute grammar description and decides whether or not the compiled code will give a terminating enumeration for a given value of the attribute[BB92, BB93], so it is not straightforward which grammars can be used. Therefore, the class of grammars acceptable to this method needs to be restricted to those that produce a complete enumeration, such as non left-recursive grammars.

An implementation of this method was firstly done for a subset of an Occam-like language using a functional programming language. The decompiler grammar was an inherited attribute grammar which took the intended object code as an argument[BB92, BB93]. A Prolog decompiler was also described based on the compiler specification. This decompiler applied the clauses of the compiler in a selective and ordered way, so that the problem of non-termination would not be met, and only a subset of the source code programs would be returned (rather than an infinite list)[Bow91, Bow93]. Recently, this method made use of an imperative programming language, C++, due to the inefficiencies of the functional and logic approach. In this prototype, C++ object's were used as lazy lists, and a set of library functions was written to implement the operators of the intermediate representation used[BB94]. Problems with optimized code have been detected.

As illustrated by this research, decompiler compilers can be constructed automatically if the set of compiler specifications and object code produced for each clause of the specification is known. In general, this is not the case as compiler writers do not disclose their compiler specifications. Only customized compilers and decompilers can be built by this method. It is also noted that optimizations produced by the optimization stage of a compiler are not handled by this method, and that real executable programs cannot be decompiled by the decompilers generated by the method described. The problem of separating instructions from data is not addressed, nor is the problem of determining the data types of variables used in the executable program. In conclusion, decompiler compilers can be generated automatically if the object code produced by a compiler is known, but the generated decompilers cannot decompile arbitrary executable programs.

8086 C Decompiling System, 1991–1993. This decompiler takes as input executable files from a DOS environment and produces C programs. The input files need to be compiled with Microsoft C version 5.0 in the small memory model[FZL93]. Five phases were described: recognition of library functions, symbolic execution, recognition of data types, program transformation, and C code generation. The recognition of library functions and intermediate language was further described in [FZ91, HZY91].

The recognition of library functions for Microsoft C was done to eliminate subroutines that were part of a library, and therefore produce C code for only the user routines. A table of C library functions is built-into the decompiling system. For each library function, its name, characteristic code (sequence of instructions that distinguish this function from any other function), number of instructions in the characteristic code, and method to recognize the function were stored. This was done manually by the decompiler writer. The symbolic execution translated machine instructions to intermediate instructions, and represented each instruction in terms of its symbolic contents. The recognition of data types is done by a set of rules for the collection of information on different data types and analysis rules to determine the data type in use. The program transformation transforms storage calculation into address expressions, e.g. array addressing. Finally, the C code generator transforms the program structure by finding control structures, and generates C code.

This decompiling system makes use of library function recognition to generate more readable C programs. The method of library recognition is hand-crafted, and therefore inefficient if other versions of the compiler, other memory models, or other compilers were used to compile the original programs. The recognition of data types is a first attempt to recognize types of arrays, pointers and structures, but not much detail is given in the paper. No description is given as to how an address expression is reached in the intermediate code, and no examples are given to show the quality of the final C programs.

Alpha AXP Migration Tools, 1993. When Digital Equipment Corporation designed the Alpha AXP architecture, the AXP team got involved in a project to run existing VAX and MIPS code on the new Alpha AXP computers. They opted for a binary translator which would convert a sequence of instructions of the old architecture into a sequence of instructions of the new architecture. The process needed to be fully automatic and to cater for code created or modified during execution. Two parts to the migration process were defined: a binary translation, and a runtime environment[SCK+93].

The binary translation phase took binary programs and translated them into AXP opcodes. It made use of decompilation techniques to understand the underlying meaning of the machine instructions. Condition code usage analysis was performed as these conditions do not exist on the Alpha architecture. The code was also analyzed to determine function return values and find bugs (e.g. uninitialized variables). MIPS has standard library routines which are embedded in the binary program. In this case, a pattern matching algorithm was used to detect routines that were library routines, such routines were not analysed but replaced by their name. Idioms were also found and replaced by an optimal instruction sequence. Finally, code was generated in the form of AXP opcodes. The new binary file had both, the new code and the old code.

The runtime environment executes the translated code and acts as a bridge between the new and old operating systems (e.g. different calling standards, exception handling). It had a built-in interpreter of old code to run old code not discovered or nonexistent at translation time. This was possible because the old code was also saved as part of the new binary file.

Two binary translators were written: VEST, to translate from the OpenVMS VAX system to the OpenVMS AXP system, and `mx`, to translate ULTRIX MIPS images to DEC OSF/1 AXP images. The runtime environments for these translators were TIE and `mxr` respectively.

This project illustrates the use of decompilation techniques in a modern translation system. It proved successful for a large class of binary programs. Some of the programs that could not be translated were programs that were technically infeasible to translate, such as programs that use privileged opcodes, or run with superuser privileges.

Source/PROM Comparator, 1993. A tool to demonstrate the equivalence of source code and PROM contents was developed at the Nuclear Electric plc, UK, to verify the correct translation of PL/M-86 programs into PROM programs executed by safety critical computer controlled systems[PW93].

Three stages are identified: the reconstitution of object code files from the PROM files, the disassembly of object code to an assembler-like form with help from a name-table built up from the source code, and decompilation of assembler programs and comparison with the original source code. In the decompiling stage, it was noted that it was necessary to eliminate intermediate jumps, registers and stack operations, identify procedure arguments, resolve indexes of structures, arrays and pointers, and convert the expresssions to a normal form. In order to compare the original program and the decompiled program, an intermediate language was used. The source program was translated to this language with the use of a commercial product, and the output of the decompilation stage was written in the same language. The project proved to be a practical way of verifying the correctness of translated code, and to demonstrate that the tools used to create the programs (compiler, linker, optimizer) behave reliably for the particular safety system analyzed.

This project describes a use of decompilation techniques, to help demonstrate the equivalence of high-level and low-level code in a safety-critical system. The decompilation *stage performs much of the analysis, with help from a symbol table constructed from the original source program. The task is simplified by the knowledge of the compiler used to compile the high-level programs.*

In the last years, commercial vendor-specific decompilers have been manufactured. These decompilers are targetted at the decompilation of binary files produced by database languages, such as Clipper and FoxPro. No information on the techniques used to decompile these programs is given by their manufacturers. The following list mentions some of these commercial decompilers:

Valkyrie, 1993. Visual decompiler for Clipper Summer '87, manufactured by CodeWorks [Val93].

OutFox, 1993. Decompiler for encrypted FoxBASE+ programs [Out93].

ReFox, 1993. Decompiles encrypted FoxPro files, manufactured by Xitech Inc [HHB+93].

DOC, 1993. COBOL decompiler for AS/400 and System/38. Converts object programs into COBOL source programs which can be modified by the programmer. Manufactured by Harman Resources [Cob93].

Uniclip, 1993. Decompiler for Clipper Summer '87 EXE files, manufactured by Stro Ware [Unc93].

Clipback, 1993. Decompiler for Summer '87 executables, manufactured by Intelligent Information Systems [Unc93].

Brillig, 1993. Decompiler for Clipper 5.X **.exe** and `.obj` files, manufactured by APTware [Bri93].

Chapter 3

Run-time Environment

Before considering decompilation, the relations between the static binary code of the program and the actions performed at run-time to implement the program are presented. The representation of objects in a binary program differs between compilers; elementary data types such as integers, characters, and reals are often represented by an equivalent data object in the machine (i.e. a fixed size number of bytes), whereas aggregate objects such as arrays, strings, and structures are represented in various different ways.

Throughout this thesis, the word **subroutine** is used as a generic word to denote a procedure or a function; the latter two words are used only when there is certainty as to what the subroutine really is, that is, a subroutine that returns a value is a **function**, and a subroutine that does not return a value is a **procedure**.

3.1 Storage Organization

A high-level language program is composed of one or more subroutines, called the user subroutines. The corresponding binary program is composed of the user subroutines, library routines that were invoked by the user program, and other subroutines linked in by the linker to provide support for the compiler at run-time. The general format of the binary code of a program is shown in Figure 3-1. The program starts by invoking compiler start-up subroutines that set up the environment for the compiler; this is followed by the user's main program subroutine, which invokes library routines linked in by the linker; and is finalized by a series of compiler subroutines that restore the state of the machine before program termination.

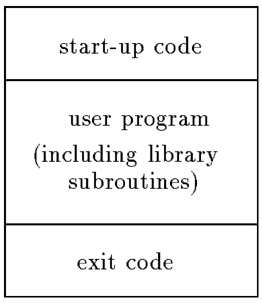


Figure 3-1: General Format of a Binary Program

For example, a "hello world" C program compiled with Borland Turbo C v2.01 has over 25 different subroutines. The start-up code invokes up to 16 different subroutines to set up the compiler's environment. The user's main program is composed of one procedure. This procedure invokes the `printf()` procedure which then invokes up to 8 different subroutines to display the formatted string. Finally, the exit code invokes 3 subroutines to restore the environment and exit back to DOS. Sample skeleton code for this program is shown in Figure 3-2.

```
helloc  proc  far
        mov   dx,DGROUP       ; dx == GROUP segment adr
        mov   cs:DGROUP@@,dx
        ; save several vectors and install default divide by zero handler
        call  SaveVectors
        ; calculate environment size, determine amount of memory needed,
        ; check size of the stack, return to DOS memory allocated in excess,
        ; set far heap and program stack, reset uninitialized data area,
        ; install floating point emulator
        push  cs
        call  ds:[__emu1st]
        ; prepare main arguments
        call  _setargv@
        call  _setenvp@
        ; initialize window sizes
        call  ds:[__crt1st]
        ; invoke main(argc,argv,envp)
        push  word ptr environ@
        push  word ptr _argv@
        push  word ptr _argc@
        call  main@           ; user's main() program
        ; flush and close streams and files
        push  ax
        call  exit@
helloc  endp
```

Figure 3-2: Skeleton Code for a "hello world" Program

In a binary program, subroutines are identified by their entry address; there are no names associated with subroutines, and it is unknown whether the subroutine is a procedure or a function before performing a data flow analysis on the registers defined and used by these subroutines. It is said that a subroutine that invokes another subroutine is the caller, and the invoked subroutine is the callee.

3.1.1 The Stack Frame

Each subroutine is associated with a stack frame during run-time. The stack frame is the set of parameters, local variables, and return address of the caller subroutine, as shown in Figure 3-3. The parameters in the stack frame represent the actual parameters of a particular invocation of the subroutine: information on the formal parameters of the subroutine are not stored elsewhere in the binary file. The stack mark represents the return address of the caller (so that control can be transferred to the caller once the callee is finished), and the caller's frame pointer (register `bp` in the Intel architecture), which is a reference point for offsets into the stack frame. The local variables represent the space allocated by the subroutine once control has been transferred to it; this space is available to the subroutine only while it is active (i.e. not terminated).

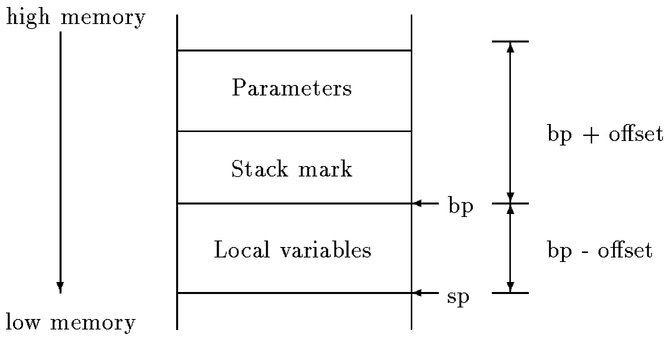


Figure 3-3: The Stack Frame

Once the frame pointer has been set (i.e. register `bp`), positive offsets from the frame pointer access parameters and the stack mark, and negative offsets access local variables. The convention used in diagrams relating to the stack frame is as follows: the stack grows downwards from high to low memory, as in the Intel architecture.

The stack frame may also contain other fields, as shown in Figure 3-4. These fields are not used by all languages nor all compilers[ASU86a]. The return value field is used in some languages by the callee to return the value of a function back to the caller; these values are more often returned in registers for efficiency. The control link points to the stack frame of the caller, and the access link points to the stack frame of an enclosing subroutine that holds non-local data that is accessible from this subroutine.

3.2 Data Types

Data objects are normally stored in contiguous memory locations. Elementary data types such as characters, integers, and longs, can be held in registers while an operation is performed on them. Aggregate data types such as arrays, strings, and records, cannot be held in registers in their entirety because their size is normally beyond the size of a register, therefore it is easier to access them through a pointer to their starting address.

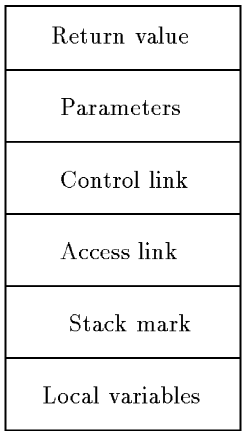


Figure 3-4: The Stack Frame

The sizes of different data types for the i80286 architecture are shown in Figure 3-5. This machine has a word size of 16 bits. Sizes are given in 8-bit bytes.

Data Type	Size (bytes)
character	1
integer	2
long	4
real	4
long real	8
near pointer	2
far pointer	4
other types	≥ 1

Figure 3-5: Size of Different Data Types in the i80286

3.2.1 Data Handling in High-level Languages

Aggregate data types are handled in several different ways by different compilers. This section describes different formats used by C, Pascal, Fortran, and Basic compilers, according to [Mic87].

Array

An array is a contiguous piece of memory that holds one or more items of a certain type. Arrays are implemented in memory as a series of rows or columns, depending on the order used by the language:

- Row-major order: the elements of a multidimensional array are stored by row order; that is, one row after the other. This order is used by C and Pascal compilers.

- Column-major order: the elements of a multidimensional array are stored in column order rather than row order. This order is used by Fortran and Basic compilers. Some Basic compilers have a compile option to use row-major order.

In most languages, the size of the array is known at compile time; this is the case of C, Pascal and Fortran. Basic allows for run-time declared array sizes, therefore an array needs to have an array-descriptor to hold the size of the array and a pointer to the physical location in memory where the array is stored.

String

A string is a sequence of characters. Different languages use different representations for a string, such as the following:

- C format: a string is an array of bytes terminated by a null character (i.e. 0).
- Fortran format: a string is a series of bytes at a a fixed memory location, hence no delimiter is used or needed at the end of the string.
- Pascal format: common Pascal compilers have 2 types of strings: STRING and LSTRING. The former is a fixed-length string and is implemented in the Fortran format. The latter is a variable-length string and is implemented as an array of characters that holds the length of the string in the first byte of the array. Standard Pascal does not have a STRING or LSTRING type.
- Basic format: a string is implemented as a 4-byte string-descriptor; the first 2 bytes hold the length of the string, and the next 2 bytes are an offset into the default data area which holds the string. This area is assigned by Basic's string-space management routines, and therefore is not a fixed location in memory.

Record

A record is a contiguous piece of memory that holds related items of one or more data types. Different names are used for records in different languages; `struct` in C, `record` in Pascal, and user-defined type in Basic. By default, C and Pascal store structures in unpacked storage, word-aligned, except for byte-sized objects and arrays of byte-sized objects. Basic and some C and Pascal compilers store structures in packed storage.

Complex Numbers

The Fortran COMPLEX data type stores floating point numbers in the following way:

- COMPLEX*8: 4 bytes represent the real part, and the other 4 bytes represent the floating point number of the imaginary part.
- COMPLEX*16: 8 bytes represent the real part, and the other 8 bytes the imaginary part.

Boolean

The Fortran LOGICAL data type stores Boolean information in the following way:

- LOGICAL*2: 1 byte holds the Boolean value (0 or 1), and the other byte is left unused.
- LOGICAL*4: 1 byte holds the Boolean value, and the other 3 bytes are left unused.

3.3 High-Level Language Interface

Compilers of high-level languages use a series of conventions to allow mixed-language programming, so that a program can have some subroutines written in one language, and other subroutines written in a different language, and all these subroutines are linked in together in the same program. The series of conventions relate to the way the stack frame is set up, and the calling conventions used to invoke subroutines.

3.3.1 The Stack Frame

The stack mark contains the caller's return address and frame pointer. The return address varies in size depending on whether the callee is invoked using a near or far call. Near calls are within the same segment and therefore can be referenced by an offset from the current segment base address. Far calls are in a different segment, so both segment and offset of the callee are stored. For a 2-byte machine word architecture, the near call stores 2 bytes for the offset of the caller, whereas the far call stores 4 bytes for the segment and offset of the caller. Register `bp` is used as the frame pointer, the contents of the caller's frame pointer is pushed onto the stack at subroutine entry so that it can be restored at subroutine termination.

Entering a Subroutine

Register `bp` is established as the frame pointer by pushing its address onto the stack (i.e. storing the frame pointer of the caller on the stack), and copying the current stack pointer register (`sp`) to `bp`. The following code is used in the i80286 architecture:

```
push bp          ; save old copy of bp
mov  bp, sp      ; bp == frame pointer
```

Allocating Local Data

A subroutine may reserve space on the stack for local variables. This is done by decrementing the contents of the stack register `sp` by an even amount of bytes. For example, to allocate space for 2 integer variables, 4 bytes are reserved on the stack:

```
sub  sp, 4
```

Preserving Register Values

The most widely used calling convention for DOS compilers demands that a subroutine should always preserve the values of registers `si`, `di`, `ss`, `ds`, and `bp`. If any of these registers is used in the callee subroutine, their values are pushed onto the stack, and restored before subroutine return. For example, if `si` and `di` are used by a subroutine, the following code is found after local data allocation:

```
push si
push di
```

Accessing Parameters

Parameters are located at positive offsets from the frame pointer register, `bp`. In order to access a parameter n, the offset from `bp` is calculated by adding the size of the stack mark, plus the size of the parameters between `bp` and parameter n, plus the size of parameter n.

Returning a Value

Functions returning a value in registers use different registers according to the size of the returned value. Data values of 1 byte are returned in the `al` register, 2 bytes are returned in the `ax` register, and 4 bytes are returned in the `dx:ax` registers, as shown in Figure 3-6.

Data Size (bytes)	Register
1	AL
2	AX
4	DX = high byte
	AX = low byte

Figure 3-6: Register Conventions for Return Values

Larger data values are returned using the following conventions:

- Function called by C: the callee must allocate space from the heap for the return value and place its address in `dx:ax`.
- Function called by Pascal, Fortran or Basic: the caller reserves space in the stack segment for the return value, and pushes the offset address of the allocated space on the stack as the last parameter. Therefore, the offset address of the return value is at `bp + 6` for far calls, and `bp + 4` for near calls, as shown in Figure 3-7.

Exiting the Subroutine

The stack frame is restored by popping any registers that were saved at subroutine entry, deallocating any space reserved for local variables, restoring the old frame pointer (`bp`), and returning according to the convention in use.

- C convention: the caller adjusts the stack for any parameters pushed on the stack. A `ret` instruction is all that is needed to end the subroutine.

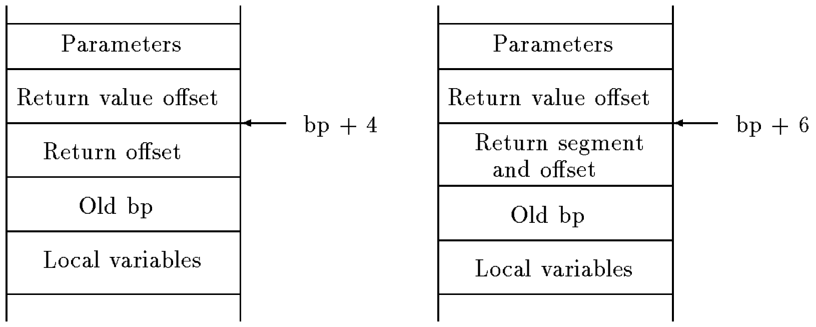


Figure 3-7: Return Value Convention

- Pascal, Fortran, Basic convention: the callee adjusts the stack by cutting back the stack with the required number of parameter bytes. A `ret n` instruction is used, where `n` is the number of bytes of the parameters.

For example, the following code restores the registers `di` and `si` from the stack, deallocates the space of the local variables by copying `bp` to `sp`, restores the frame pointer by popping `bp` from the stack, and returns using the C convention:

```
pop di              ; restore registers
pop si
mov sp, bp          ; deallocate local variables
pop bp              ; restore bp
ret
```

3.3.2 Parameter Passing

Three different methods are used to pass parameters on the Intel architecture under the DOS operating system; C, Pascal, and register calling conventions. Mixtures of these calling conventions are available in other operating systems and architectures. For example, in OS/2, the standard call uses C ordering to pass parameters, but the callee cuts back the parameters from the stack in system calls.

C Calling Convention

The caller is responsible for pushing the parameters on the stack, and restoring them after the callee returns. The parameters are pushed in right to left order, so that a variable number of parameters can be passed to the callee. For example, for a C function prototype `void procX (int, char, long)`, and a caller procedure that invokes the `procX()` procedure:

```
procN()
{ int i;     /* bp - 8 */
  char c;    /* bp - 6 */
  long l;    /* bp - 4 */
    procX (i, c, l);
}
```

the following assembler code is produced:

```
push word ptr [bp-2]     ; high word of l
push word ptr [bp-4]     ; low word of l
push [bp-6]              ; c
push word ptr [bp-8]     ; i
call procX               ; call function
add  sp, 8               ; restore the stack
```

Note that due to word alignment, the character `c` is stored as 2 bytes on the stack even though its size is one byte only.

Pascal Calling Convention

The caller is responsible for pushing the arguments on the stack, and the callee adjusts the stack before returning. Arguments are pushed on the stack in left to right order, hence a fixed number of arguments are used in this convention. For the previous example, the calling of `procX (i, c, l)` produces the following assembler code in Pascal convention:

```
push word ptr [bp-8]     ; i
push [bp-6]              ; c
push word ptr [bp-2]     ; high word of l
push word ptr [bp-4]     ; low word of l
call procX               ; call procX (procX restores stack)
```

Register Calling Convention

This convention does not push arguments on the stack but passes them in registers, therefore the generated code is faster. Predetermined registers are used to pass arguments between subroutines, and different registers are used for different argument sizes. Figure 3-8 shows the set of registers used by Borland Turbo C++ [Bor92]; a maximum of 3 parameters can be passed in registers. Far pointers, unions, structures, and real numbers are pushed on the stack.

Parameter Type	Register
Character	al, dl, bl
Integer	ax, dx, bx
Long	dx:ax
Near pointer	ax, dx, bx

Figure 3-8: Register Parameter Passing Convention

3.4 Symbol Table

A decompiler uses a symbol table to store information on variables used throughout the program. In a binary program, variables are identified by their address; there are no names associated with variables. Variables that have a physical memory address are global variables; their segment and offset are used to access them. Variables that are located at a negative offset from the frame pointer are local variables to the corresponding stack frame's subroutine, and variables at positive offsets are actual arguments to the subroutine. Since register variables are used by compilers for efficiency purposes, all registers are also considered variables initially; further analysis on registers determines whether they represent register variables or not (see Chapter 5, Section 5.2.9). Variables are assigned unique names during code generation, as explained in Chapter 7.

The symbol table must be able to provide information on an entry efficiently, and handle a varying number of variables; hence, a symbol table that grows dynamically if necessary is desirable. The performance of the symbol table is measured in terms of the time taken to access an entry and insert a new item to the table.

3.4.1 Data Structures

Symbol tables are represented by a variety of data structures. Some are more efficient than others, at the expense of more coding. To illustrate the differences between the various data structures, let us assume the following data items are to be placed in the symbol table:

```
cs:01F8   ; global variable
bp + 4    ; parameter
bp - 6    ; local variable
ax        ; register ax
bp - 2    ; local variable
```

Unordered List

An unordered list is a linked-list or an array of data items. Items are stored in the list in a first-in basis (i.e. on the next available position). An array implementation presents the limitation of size; these limitations are avoided by the use of a linked-list implementation. An access to this symbol table, for a list of n items is $O(n)$. Figure 3-9 shows the list built for our example.

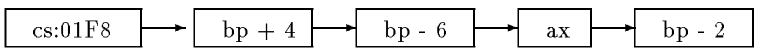


Figure 3-9: Unordered List Representation

Ordered List

An ordered list is easier to access, since not all items of the list need to be checked to determine whether an item is already in the list or not. Ordered lists can be searched using a binary search, which provides an access time of $O(\log n)$. Insertion of an item is costly, since the list needs to remain ordered.

Since there are different types of variables in a binary program, and these items are identified in a different way based on their type, an ordering within a type is possible, but the four different types must be access independently. Figure 3-10 shows the ordered list representation of our example: a record that determines the type of the data item is used first, and each of the data types has an ordered list associated with it.

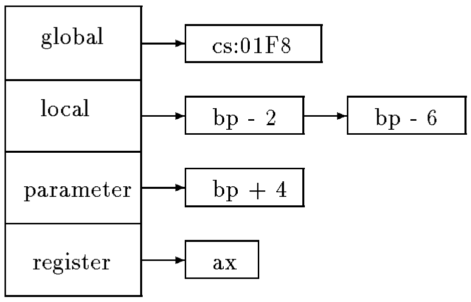


Figure 3-10: Ordered List Representation

Hash Table

A hash table is a mapping between a fixed number of positions in a table, and a possibly large number of variables. The mapping is done via a hashing function which is defined for all possible variables, can be computed quickly, provides an uniform probability for all variables, and randomizes similar variables to different table locations.

In open hashing, a hash table is represented by an array of a fixed size, and a linked-list attached to each array position (bucket). The linked-list holds different variables that hash to the same bucket. Figure 3-11 shows the hash table built for our example; as for ordered lists, a record which determines the type of the variable is used first, and a hash table is associated with each different variable type.

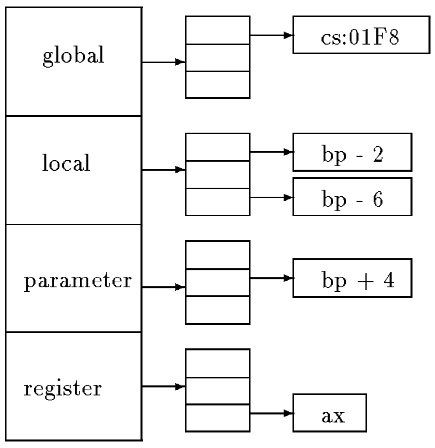


Figure 3-11: Hash Table Representation

Symbol Table Representation for Decompilation

A combination of the abovementioned methods is used for the purposes of decompilation. The symbol table is defined in terms of the different types of variables; global, local, parameter, and register. Each of these types is implemented in a different way. For global variables, since their address range is large, a hash table implementation is most suited. For local variables and parameters, since these variables are offsets from the frame pointer, and are always allocated in an ordered way (i.e. there are no "gaps" in the stack frame), they are implemented by an ordered list on the offset; the register `bp` does not need to be stored since it is always the same. Finally, for registers, since there is a fixed number of registers, an array indexed by register number can be implemented; array positions that have an associated item represent registers that are defined in the symbol table. This representation is shown in Figure 3-12.

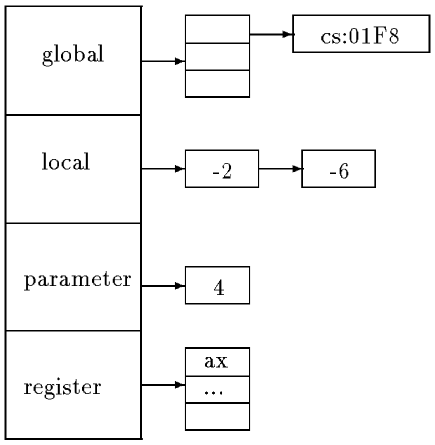


Figure 3-12: Symbol Table Representation

Symbol tables are widely discussed in the literature, refer to [ASU86a, FJ88a, Gou88] for more information on symbol tables from a compiler point of view.

Chapter 4

The Front-end

The front-end is a machine dependent module which takes as input the binary source program, parses the program, and produces as output the control flow graph and intermediate language representation of the program. The phases of the front-end are shown in Figure 4-1.

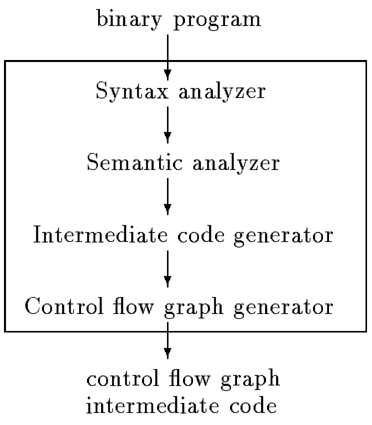


Figure 4-1: Phases of the Front-end

Unlike compilers, there is no lexical analysis phase in the front-end of the decompiler. The lexical analyzer or scanner is the phase that groups characters from the input stream into tokens. Specialized tools such as *lex* and *scanGen* have been designed to help automate the construction of scanners for compilers[FJ88a]. Given the simplicity of machine language, there is no need to scan the input bytes to recognize different words that belong to the language; all information is stored in terms of bytes or bits from a byte, and it is not possible to determine what a particular byte represents (i.e. opcode, register, offset) out of context. The syntax analyzer determines what a series of bytes represent based on the language description of the machine language.

4.1 Syntax Analysis

The syntax analyzer is the first phase of the decompiler. Its role is to group a sequence of bytes into a phrase or sentence of the language. This sequence of bytes is checked for syntactic structure, that is, that the string belongs to the language. Valid strings are represented by a parse tree, which is input into the next phase, the semantic analyzer. The relation between the syntax analyzer and the semantic analyzer is shown in Figure 4-2. The syntax analyzer is also known as the parser.

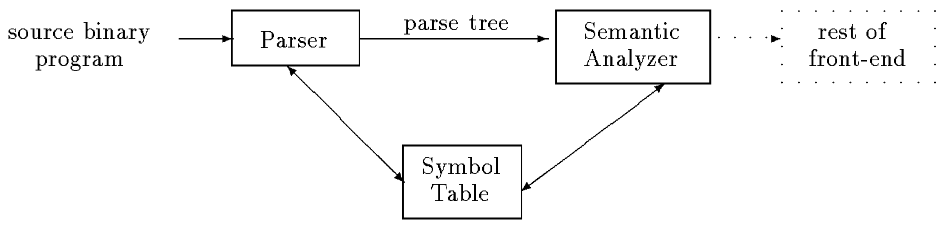


Figure 4-2: Interaction between the Parser and Semantic Analyzer

The syntax of a machine language can be precisely specified by a grammar. In machine languages, only instructions or statements are specified; there are no control structures as in high-level languages. In general, a grammar provides a precise notation for specifying any language.

The main difficulty of a decompiler parser is the separation of code from data, that is, the determination of which bytes in memory represent code and which ones represent data. This problem is inherent to the von Neumann architecture, and thus, needs to be addressed by means of heuristic methods.

Syntax Errors

Syntax errors are seldom found in binary programs, as compilers generate correct code for a compiled program to run on a machine. But, given that upgrades of a machine architecture result in new machines that support all predecessor machines, the machine instruction set of the new architecture is an extension of the instruction set of the old architecture. This is the case of the i80486, which supports all predecessor i8086, i80186, i80286, and i80386 instruction sets. Therefore, if a parser is written for the i8086, all new machine instructions are not recognized by the parser and must result in an error. On the other hand, if the parser is written for the newest machine, the i80486, all instructions should be recognized and no syntactic errors are likely to be encountered.

4.1.1 Finite State Automaton

A Finite State Automaton (FSA) is a recognizer for a language. It takes as input a string, and answers *yes* if the string belongs to the language and *no* otherwise. A string is a sequence of symbols of a given alphabet. Given an arbitrary string, an FSA can determine whether the string belongs to the language or not.

Definition 1 *A Finite State Automaton is a mathematical model that consists of:*

- *A finite set of states S*
- *An initial state s_0*

- *A set of final or accept states F*
- *An alphabet of input symbols Σ*
- *A transition function T: state $\times$ symbol $\longrightarrow$ state*

An FSA can be graphically represented by transition diagrams. The components of these diagrams are shown in Figure 4-3. The symbols from the alphabet label the transitions. Error transitions are not explicitly represented in the diagram, and it is assumed that any non-valid symbol out of a state labels a transition to an error state.

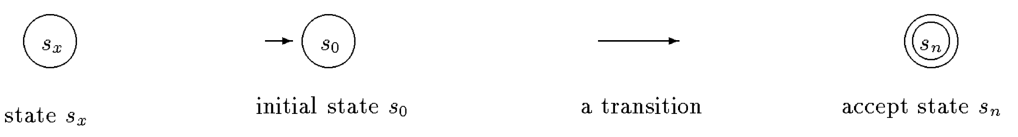
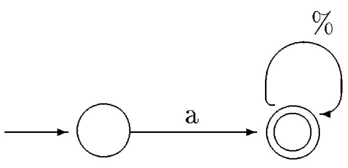


Figure 4-3: Components of a FSA Transition Diagram

A **wildcard language** is a meta-language used to specify wildcard conditions in a language[Gou88]. Two meta-symbols are used '*' and '%'. The meta-symbol '*' represents any sequence of zero or more symbols from the alphabet Σ, and '%' represents any single symbol from Σ.

Example 1 *Let $\Sigma = \{a, b, c\}$. The language that accepts all strings starting with an* **a** *is described by the wildcard expression* **aa***, *and is represented in an FSA in the following way:*



Non-deterministic Finite State Automaton

An FSA is said to be non-deterministic (NFSA) whenever there are two or more transitions out of a state labelled with the same symbol, or when the empty string (ε) labels a transition. In these cases, the next state is not uniquely identified by a (state, symbol) tuple.

Deterministic Finite State Automaton

A deterministic finite state automaton (DFSA) is a FSA that has no transitions labelled by the ε string, and that uniquely identifies or determines the next state of a (state, symbol) tuple.

Any NFSA can be converted into an equivalent DFSA by a method of *subset construction.* This method has been explained in the literature, for example, refer to [ASU86b, Gou88, FJ88a] for details. A method to construct a minimum-state DFSA is also described in [ASU86b].

4.1.2 Finite State Automatons and Parsers

Any machine language can be represented by an FSA that accepts or rejects arbitrary strings. The alphabet Σ is the finite set of hexadecimal numbers `00..FF` (i.e. numbers represented by a byte), and a string is a sequence of bytes. The strings that belong to this language are those instructions recognized by the particular machine language.

Example 2 *An FSA to recognize the i80286 machine instruction*

```
83E950      ; sub cx, 50
```

needs to first determine that `83` *is an opcode (*`sub`*), and that it takes two or more bytes as operands. The second byte encodes the destination register operand (lower 3 bits), and how many bytes of other information there are after this byte: whenever the upper two bits are 0 or 2, 2 more bytes follow this second byte, if these bits represent 1, 1 byte follows the second byte, otherwise there are no more bytes as part of the destination operand. In our example, the lower three bits are equivalent to 1, which is register* `cx`*, and the upper two bits are 3, which means there are no more bytes as part of the destination operand. Finally, the last byte is the immediate constant operand,* `50` *in this example. The FSA for this example is shown in Figure 4-4.*

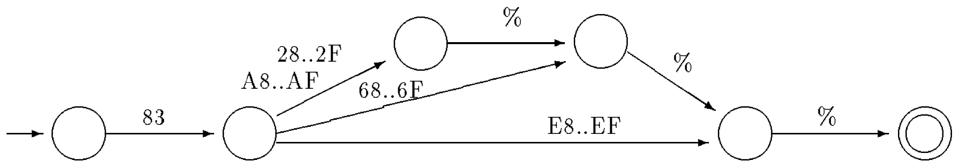


Figure 4-4: FSA example

A machine language can also be described in terms of a context-free grammar (CFG); as regular expressions are a subset of context-free grammars. An algorithm to mechanically convert an NFSA into a CFG is presented in [ASU86a]. CFGs are used to specify high-level constructs that have an inherent recursive structure, and since machine languages do not make use of recursive constructs, it is not necessary to define them in terms of CFGs.

4.1.3 Separation of Code and Data

Given the entry point to a program, it is the function of the syntax analyzer to parse machine instructions following all possible paths of the program. The main problem faced by the parser is that data and code are represented in the same way in a von Neumann machine, thus it is not easy to determine if the byte(s) that follows an instruction belongs to another instruction or represents data. Heuristic methods need to be used in order to determine data from code, as explained in this section.

Once the source binary program has been loaded into memory, the loader returns the initial start address of the program in memory. This address is the starting address for the complete binary program, and thus, must be the address of an instruction in order for the program to run. Furthermore, if the binary program has been checked against compiler signatures, the initial starting address is the entry point to the `main` of the program; i.e.

the start address of the user-written program, skipping all compiler start up code. Figure 4-5 illustrates sample code for a "hello world" program. The entry point returned by the loader is CS:0000, which is the entry point to the complete program (including compiler start up code). The entry point given by the compiler signature analyzer is CS:01FA, which is the start address of the main program. Throughout this thesis we will assume the entry point is the one given by the compiler signature analyzer without loss of generality. The explained methods are applicable to both cases, but more interesting examples are found in the latter case. The technique for generating compiler signatures and detecting them is given in Chapter 8.

```
        helloc  proc  far
CS:0000 start:  mov   dx,*
CS:0003         mov   cs:*,dx
...             ...             ; start-up code
CS:011A         call  _main
CS:011D         ...             ; exit code
        helloc  endp
...             ...
        _main   proc  near
CS:01FA         push  bp
CS:01FB         mov   bp,sp
CS:01FD         mov   ax,194h
CS:0200         push  ax
CS:0201         call  _printf
CS:0204         pop   cx
CS:0205         pop   bp
CS:0206         ret
        _main   endp
```

Figure 4-5: Sample Code for a "hello world" Program

A paper by R.N. Horspool and N. Marovac focused on the problem of separation of code from data. This paper mentioned that this problem is equivalent to the halting problem, as it is impossible to separate data from instructions in a von Neumann architecture that computes both data addresses and branch destination addresses at execution time[HM79]. An algorithm to find the maximum set of locations holding instructions was given. This modification of the original problem is equivalent to a combinatorial problem of searching for a maximal set of trees out of all the candidate trees, for which a branch-and-bound method is applied. The algorithm is proved to be NP-Complete.

As it turns out, in dense machine instruction sets (such as in the Intel architecture), the given algorithm does not work, as almost any byte combination is a valid machine instruction, and therefore it is hard to determine the bounds of the code since it is hard to know when data has been reached. A simple counter-example to this algorithm is given by a case *table stored in the code segment (see Figure 4-6). After the indexed jump instruction at* `CS:0DDB`*, which indexes into the* `case` *table, the table itself is defined starting at* `CS:0DE0`*, but yet it is treated as code by the algorithm as it includes valid bytes for instructions. In this i80286 code example,* `0E` *is equivalent to* `push CS`*,* `2B` *is equivalent to* `sub` *which takes one other byte as argument,* `0E` *in this case, to result in* `sub ax,[bp]`*, and so on. The produced code is therefore wrong.*

```
CS:0DDB  jmp CS:0DE0[bx]
CS:0DE0  0E2B      ; push CS
CS:0DE2  0E13      ; sub ax,[bp]
         ...
```

Figure 4-6: Counter-example

This section presents a different method to determine code from instructions in a binary program that has been loaded into memory. It provides heuristic methods to determine special cases of data found in between sections of code.

The Process

As previously mentioned, the process of determining data from code is based on the knowledge that the initial entry point to the program is an instruction. From this instruction onwards, instructions are parsed sequentially along this path, until a change in the flow of control or an end of path is reached. In the former case, the target address(es) acts as new entry points into memory, as the address must hold a valid instruction in order for the program to continue execution. In the latter case, the end of the current path is reached and no more instructions are scanned along this path as we cannot determine whether these next bytes are code or data.

Changes in the flow of control are due to jumps and procedure calls. A *conditional jump* branches the flow of control in two: the target branch address is followed whenever the condition is true, otherwise the address following the conditional branch is followed. Both paths are followed by the parser in order to get all possibly executable code. An *unconditional jump* transfers the flow of control to the target address; this unique path is followed by the parser. A *procedure call* transfers control to the invoked procedure, and once it returns, the instructions following the procedure call are parsed. In the case that the procedure does not return, the bytes following the procedure call are not parsed as it is not certain what these bytes are (code or data).

An end of path is reached whenever a procedure return instruction or an end of program is met. The end of program is normally specified by a series of instructions that make the operating system terminate the current process (i.e. the program). This sequence of instructions varies between operating systems, so they need to be coded for the specific source machine. Determining whether the end of program is met (i.e. the program finishes or halts) is not equivalent to solving the halting problem though, as the path that is being followed is not necessarily a path that the executable program will follow, i.e. the condition that branches onto this path might never become true during program execution; for example, programs in an infinite loop.

Example 3 *On the Intel architecture, the end of a program is specified via interrupt instructions. There are different methods to terminate a program, some of these methods make use of the program segment prefix, commonly referred to as the PSP; refer to Appendix B for more information on the PSP. There are 7 different ways of terminating a program under DOS:*

1. *Terminate process with return code:* `int 21h`*, function 4Ch. This is the most commonly used method in* **.exe** *files.*
2. *Terminate process:* `int 20h`*. The code segment,* `cs`*, needs to be pointing to the PSP. This method is normally used in* **.com** *files as* `cs` *already points to the PSP segment.*
3. *Warm boot/Terminate vector: offset 00h in the PSP contains an* `int 20h` *instruction. Register* `cs` *must be pointing to the PSP segment.*
4. *Return instruction: the return address is placed on the stack before the program starts. When the program is to be finished, it returns to this address on the stack. This method was used in the CP/M operating system as the address of the warm boot vector was on the stack. Initial DOS* **.com** *programs made use of this technique.*
5. *Terminate process function:* `int 21h`*, function 00h. Register* `cs` *must point to the PSP.*
6. *Terminate and stay resident:* `int 27h`*. Register* `cs` *must point to the PSP.*
7. *Terminate and stay resident function:* `int 21h`*, function 31h.*

Determining whether a procedure returns (i.e. finishes or halts) or not is difficult, as the procedure could make use of self-modifying code or execute data as code and terminate in an instruction within this data. In general, we are interested in a solution for normal cases, as aberrant cases require a step debugger tool and user input to solve the problem. A procedure does not return if it reaches the end of program or invokes a procedure that reaches the end of program (e.g. a procedure that invokes `exit(.)` in C). Determining whether a procedure has reached the end of program is possible by emulation of the contents of the registers that are involved in the sequence of instructions that terminate the program. In the case of Example 3, keeping track of registers `ah`, and `cs` in most cases.

This initial algorithm for separation of code from data is shown in Figure 4-7. In order to keep track of registers, a `machState` record of register values is used. A `state` variable of this type holds the current values of the registers (i.e. the current state of the machine). A bitmap of 2 bits per memory byte is used to store information regarding each byte that was loaded into memory:

- `0`: represents an unknown value (i.e. the memory location has not been analyzed).
- `1`: represents a code byte.

- 2: represents a data byte.
- 3: represents a byte that is used as both, data and code.

The algorithm is implemented recursively. Each time a non fall-through path needs to be followed, a copy of the current `state` is made, and the path is followed by a recursive call to the `parse` procedure with the copy of the state.

Indirect Addressing Mode

The indirect addressing mode makes use of the contents of a register or memory location to determine the target address of an instruction that uses this addressing mode. Indirect addressing mode can be used with the unconditional jump (e.g. to implement indexed case tables) and the procedure call instructions. The main problem with this addressing mode is that the contents of memory can be changed during program execution, and thus, a static analysis of the program will not provide the right value, and is not able to determine if the memory location has been modified. The same applies to register contents, unless the contents of registers is being emulated, but again, if the register is used within a loop, the contents of the register will most likely be wrong (unless loops are emulated also).

In the i80286, an indirect instruction can be intra-segment or inter-segment. In the former case, the contents of the register or memory location holds a 16-bit offset address, in the latter case, a 32-bit address (i.e. segment and offset) is given.

Indirect procedure calls are used in high-level languages like C to implement pointers to function invocation. Consider the following C program:

```
typedef char (*tfunc)();
tfunc func[2] = {func1, func2};

char func1() {/* some code here */}
char func2() {/* some code here */}

main()
{
   func[0]();
   func[1]();
}
```

In the `main` program, functions `func1()` and `func2()` are invoked by means of a function pointer and an index into the array of such functions. The disassembled code of this program looks like this:

```
CS:0094  B604  ; address of proc1 (04B6)
CS:0098  C704  ; address of proc2 (04C7)
...
          proc_1 PROC FAR
CS:04B6  55           push bp
...                   ...
CS:04C6  CB           retf
```

```
procedure parse (machState *state)
  done = FALSE;
  while (! done)
     getNextInst (state, &inst);
     if (alreadyParsed (inst))  /* check if instruction already parsed */
        done = TRUE;
        break;
     end if
     setBitmap (CODE, inst);
     case (inst.opcode) of
       conditional jump:
         *stateCopy = *state;
         parse (stateCopy);              /* fall-through */
         state->ip = targetAdr (inst);   /* target branch address */
         if (hasBeenParsed (state->ip))  /* check if code already parsed */
            done = TRUE;
         end if
       unconditional jump:
         state->ip = targetAdr (inst);   /* target branch address */
         if (hasBeenParsed (state->ip))  /* check if code already parsed */
            done = TRUE;
         end if
       procedure call:
         /* Process non-library procedures only */
         if (! isLibrary (targetAdr (inst)))
            *stateCopy = *state;
            stateCopy->ip = targetAdr (inst);
            parse (stateCopy);           /* process target procedure */
         end if
       procedure return:
         done = TRUE;                    /* end of procedure */
       move:
         if (destination operand is a register)
            updateState (state, inst.sourceOp, inst.destOp);
         end if
       interrupt:
         if (end of program via interrupt)
            done = TRUE;                 /* end of program */
         end if
     end case
  end while
end procedure
```

Figure 4-7: Initial Parser Algorithm

```
        proc_1 ENDP

        proc_2 PROC FAR
CS:04C7  55           push bp
...                   ...
CS:04D7  CB           retf
        proc_2 ENDP

        main PROC FAR
CS:04D8  55           push bp
CS:04D9  8BEC         mov  bp, sp
CS:04DB  FF1E9400     call 0094   ; intra-segment indirect call
CS:04DF  FF1E9800     call 0098   ; intra-segment indirect call
CS:04E3  5D           pop  bp
CS:04E4  CB           retf
        main ENDP
```

The function pointers have been replaced by the memory offset of the address that holds the address of each procedure (i.e. `04B6` and `04C7` respectively). If these addresses have not been modified during program execution, checking the contents of these memory locations provides us with the target address of the function(s). This is the implementation that we use. The target address of the function is replaced in the procedure call instruction, and an invocation to a normal procedure is done in our decompiled C program, as follows:

```
void proc_1() {/* some code */}
void proc_2() {/* some code */}

void main()
{
   proc_1();
   proc_2();
}
```

Case Statements

High-level languages implement multiway (or n-way) branches via a high-level construct known as a `case` statement. In this construct, there are n different possible paths to be executed (i.e. n different branches). There is no low-level machine instruction to represent this construct, therefore different methods are used by compiler writers to define a `case` table.

If the number of cases is not too great (i.e. less than 10), a `case` is implemented by a sequence of conditional jumps, each of which tests for an individual value and transfers control to the code for the corresponding statement. Consider the following fragment of code in assembler

```
        cmp  al, 8    ; start of case
        je   lab1
        cmp  al, 7Fh
```

```
       je   lab2
       cmp  al, 4
       je   lab3
       cmp  al, 18h
       je   lab4
       cmp  al, 1Bh
       je   lab5
       jmp  endCase
lab1:  ...
...
lab5:  ...
...
endCase: ...          ; end of case
```

In this code fragment, register `al` is compared against 5 different byte values, if the result is equal, an unconditional jump is performed to the label that handles the case. If the register is not equal to any of the 5 options, the program unconditionally jumps to the end of the `case`.

A more compact way to implement a `case` statement is to use an indexed table that holds n target label addresses; one for each of the corresponding n statements. The table is indexed into by an indexed jump instruction. Before indexing into the table, the lower and upper bounds of the table are checked for, so that no erroneous indexing is done. Once it has been determined that the index is within the bound, the indexed jump instruction is performed. Consider the following fragment of code:

```
cs:0DCF      cmp   ax, 17h    ; 17h == 24
cs:0DD2      jbe   startCase
cs:0DD4      jmp   endCase
cs:0DD7 startCase:
             mov   bx, ax
cs:0DD9      shl   bx, 1
cs:0DDB      jmp   word ptr cs:0DE0[bx]  ; indexed jump
cs:0DE0      0E13  ; dw lab1  ; start of indexed table
cs:0DE2      0E1F  ; dw lab2
             ...
cs:0E0E      11F4  ; dw lab24 ; end of indexed table
cs:0E10 lab1:
             ...
cs:11C7 lab24:
             ...
cs:11F4 endCase:           ; end of case
             ...
```

The `case` table is defined in the code segment as data, and is located immediately after the indexed jump and before any target branch labels. Register `ax` holds the index into the table. This register is compared against the upper bound, `24`. If the register is greater than `24`, the rest of the sequence is not executed and the control is transferred to `labZ`, the first instruction after the end of the `case`. On the other hand, if the register is less or equal to

24, labA is reached and register bx is set up as the offset into the table. Since the size of the word is 2, the case table has offset labels of size 2, so the initial index into the table is multiplied by two to get the correct offset into the 2-byte table. Once this is done, the indexed jump instruction determines that the case table is in segment cs and offset 0DE0 (i.e. the next byte in memory in this case). Therefore, the target jump address is any of the 24 different options available in this table.

A very similar implementation of case statements is given by a case table that is located after the end of the procedure, and the index register into the table is the same as the register that holds the offset into the table (register bx in the following fragment of code):

```
cs:0BE7       cmp   bx, 17h    ; 17h == 24
cs:0BEA       jbe   startCase
cs:0BEC       jmp   jumpEnd
cs:0BEF startCase:
              shl   bx, 1
cs:0BF1       jmp   word ptr cs:0FB8[bx]  ; indexed jump
cs:0BF6 jumpEnd:
              jmp   endCase
cs:0BF9 lab1:
              ...
cs:0F4C lab24:
              ...
cs:0F88 endCase:                ; end of case
              ...
cs:0FB5       ret                 ; end of procedure
cs:0FB8       0BF9   ; dw lab1    ; start of indexed table
cs:0FBA       0C04   ; dw lab2
              ...
cs:0FE6       0F4C   ; dw lab24   ; end of indexed table
```

A third way to implement a case statement is to have the case table following all indexed branches. In this way, the code jumps over all target jump addresses, checks for upper bounds of the indexed table (31 in the following fragment of code), adjusts the register that indexes into the table, and branches to this location:

```
cs:0C65      jmp startCase
cs:0C68 lab5:
...          ...
cs:1356 lab31:
...          ...
cs:1383 lab2:
...          ...
cs:13B8      1403  ; dw endCase  ; Start of indexed table
cs:13BA      1383  ; dw lab2
...          ...
cs:13F4      1356  ; dw lab31 ; End of indexed table
cs:13F6 startCase:
```

```
             cmp  ax, 1Fh   ; 1Fh == 31
cs:13F9      jae  endCase
cs:13FB      xchg ax, bx
cs:13FC      shl  bx, 1
cs:13FE      jmp word ptr cs:13B8[bx]  ; indexed jump
cs:1403 endCase:
             ...
cs:1444      ret
```

A different implementation of `case` statements is by means of a string of character options, as opposed to numbers. Consider the following code fragment:

```
cs:246A 4C6C68464E6F 785875646973  ; db 'LlhFNoxXudiscpneEfgG%'
cs:2476 63706E654566 674725
cs:247F      256C ; dw lab1  ; start of table
cs:2481      2573 ; dw lab2
...          ...
cs:24A7      24DF ; dw lab21
...          ...
cs:24C4 procStart:
             push   bp
...          ...
cs:2555      mov   di, cs
cs:2557      mov   es, di     ; es = cs
cs:2559      mov   di, 246Ah  ; di = start of string
cs:255C      mov   cx, 15h    ; cx = upper bound
cs:255F      repne scasb
cs:2561      sub   di, 246Bh
cs:2565      shl   di, 1
cs:2567      jmp   word ptr cs:247F[di] ; indexed jump
cs:256C lab1:
...          ...
cs:26FF lab12:
...          ...
cs:2714      ret
```

The string of character options is located at `cs:246A`. Register `al` holds the current character option to be checked, `es`:`di` points to the string in memory to be compared against, and the `repne scasb` instruction finds the first match of register `al` in the string pointed to by `es`:`di`. Register `di` is left pointing to the character after the match. This register is then subtracted from the string's initial address plus one, and it now indexes into an indexed jump table located before the procedure on the code segment. This method is compact and elegant.

Unfortunately, there is no fixed representation of a `case` statement, and thus, the binary code needs to be manually examined in the first instance to determine how the `case` statement was implemented. Different compilers use different implementations, but normally a specific vendor's compiler uses only one or two different representations of `case` tables.

The determination of a `case` table is a heuristic method that handles a predefined set of generalized implementations. The more implementation methods that are handled by the decompiler, the better output it can generate. As heuristic methods are used, the right preconditions need to be satisfied before applying the method; i.e. if and indexed table is met and the bound of the indexed table cannot be determined, the proposed heuristic method cannot be applied.

Final Algorithm

The final algorithm used for data/code separation is shown in Figure 4-8. The algorithm is based on the algorithm of Figure 4-7, but expands on the cases of indexed jumps and indirect jumps and calls.

4.2 Semantic Analysis

The semantic analysis phase determines the meaning of a group of machine instructions, collects information on the individual instructions of a subroutine, and propagates this information across the instructions of the subroutine. In this way, base data types such as integers and long integers are propagated across the subroutine. The relation of this phase with the syntax analyzer and intermediate code generator is shown in Figure 4-9.

Definition 2 *An identifier (<ident>) is either a register, local variable (negative offset from the stack), parameter (positive offset from the stack), or a global variable (location in memory).*

4.2.1 Idioms

The semantic meaning of a series of instructions is sometimes given by an idiom. These are sequences of instructions that represent a high-level instruction.

Definition 3 *An idiom is a sequence of instructions that has a logical meaning which cannot be derived from the individual instructions.*

Most idioms are widely known to the compiler community, as they are a series of instructions that perform an operation in a unique or more efficient way than doing it with different instructions. The following sections illustrate some of the best known idioms.

Subroutine Idioms

When entering a subroutine, the base register, `bp`, is established to be the frame pointer by copying the value of the stack pointer (`sp`) into `bp`. The frame pointer is used to access parameters and local data from the stack within that subroutine. This sequence of instructions is shown in Figure 4-10. The high-level language subroutine prologue sets up register `bp` to point to the current stack pointer, and optionally allocates space on the stack for local static variables, by decreasing the contents of the stack pointer `sp` by the required number of bytes. This idiom is represented by an `enter` instruction that takes the number of bytes reserved for local storage.

```
procedure parse (machState *state)
  done = FALSE;
  while (! done)
     getNextInst (state, &inst);
     if (alreadyParsed (inst))  /* check if instruction already parsed */
        done = TRUE;  break;
     end if
     setBitmap (CODE, inst);
     case (inst.opcode) of
       conditional jump:
         *stateCopy = *state;
         parse (stateCopy);                  /* fall-through */
         state->ip = targetAdr (inst);       /* target branch address */
         if (hasBeenParsed(state->ip))       /* check if code already parsed */
            done = TRUE;
         end if
       unconditional jump:
         if (direct jump)
            state->ip = targetAdr(inst);     /* target branch address */
            if (hasBeenParsed(state->ip))    /* check if code already parsed */
               done = TRUE;
            end if
         else                        /* indirect jump */
            check for case table, if found,  determine bounds of the table.
            if (bounds determined)
               for (all entries i in the table)
                  *stateCopy = *state;
                  stateCopy->ip = targetAdr(targetAdr(table[i]));
                  parse (stateCopy);
               end for
            else                     /* cannot continue along this path */
               done = TRUE;
            end if
         end if
       procedure call:               /* Process non-library procedures only */
         if (! isLibrary (targetAdr (inst)))
            *stateCopy = *state;
            if (direct call)
               stateCopy->ip = targetAdr(inst);
            else                    /* indirect call */
               stateCopy->ip = targetAdr(targetAdr(inst));
            end if
            parse (stateCopy);               /* process target procedure */
         end if
       /* other cases (procedure return, move, interrupt) remain the same */
     end case
  end while
end procedure
```

Figure 4-8: Final Parser Algorithm

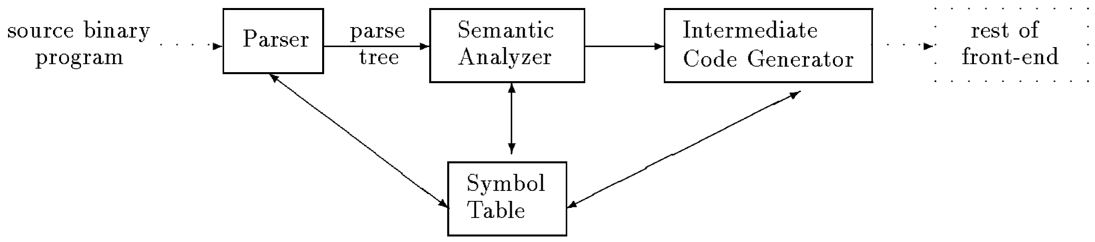


Figure 4-9: Interaction of the Semantic Analyzer

```
push bp
mov  bp, sp
[sub  sp, immed]

      ⇓

enter immed, 0
```

Figure 4-10: High-level Subroutine Prologue

Once the subroutine prologue is encountered, any `push`es on the stack represent registers whose values are to be preserved during this subroutine. These registers could act as register variables (i.e. local variables) in the current subroutine, and thus are flagged as possibly being register variables. Figure 4-11 shows registers `si` and `di` begin `push`ed on the stack.

```
push si
push di
```

Figure 4-11: Register Variables

Finally, to exit a subroutine, any registers saved on the stack need to be popped, any data space that was allocated needs to be freed, `bp` needs to be restored to point to the old frame pointer, and the subroutine then returns with a near or far return instruction. Figure 4-12 shows sample trailer code.

Calling Conventions

The C calling convention is also known as the C parameter-passing sequence. In this convention, the caller pushes the parameters on the stack, in the reverse order in which they appear in the source code (i.e. right to left order), and then invokes the procedure. After procedure return, the caller restores the stack by either `pop`ping the parameters from the

```
pop    di              ; Restore registers
pop    si
mov    sp,bp           ; Restore sp
pop    bp              ; Restore bp
ret(f)                 ; Return
```

Figure 4-12: Subroutine Trailer Code

stack, or adding the number of parameter bytes to the stack pointer. In either case, the total number of bytes used in arguments is known, and is stored for later use. The instruction(s) involved in the restoring of the stack are eliminated from the code. The C calling convention is used when passing a variable number of arguments, as the callee does not need to restore the stack. Figure 4-13 shows the case in which `pop` instructions are used to restore the stack. The total number of bytes is computed by multiplying the number of `pop`s by 2.

```
call(f)  proc_X
pop  reg
[pop reg]           reg in {ax, bx, cx, dx}
                        ⇓
proc_X.numArgBytes = 2 * numPops
sets CALL_C flag
```

Figure 4-13: C Calling Convention - Uses pop

Figure 4-14 shows the case in which the stack is restored by adding the number of argument bytes to the stack. This value is stored for later use, and the instruction that restores the stack is eliminated for further analysis. It has been found that when 2 or 4 bytes were used for arguments, the stack is restored by `pop`ping these bytes from the stack. This is due to the number of cycles involved in the two different operations: each `pop reg` instruction takes 1 byte, and an `add sp,immed` instruction takes 3 bytes. Most likely, the binary code had been optimized for space rather than speed, because a `pop reg` instruction on the i8086 takes 8 cycles, where as an `add sp,immed` instruction takes 4 cycles.

The Pascal calling convention is also known as the Pascal parameter-passing sequence. In this convention, the caller pushes the parameters on the stack in the same order as they appear in the source code (i.e. left to right order), the callee procedure is invoked, and the callee is responsible for adjusting the stack before returning. It is therefore necessary for the callee to know how many parameters are passed, and thus, it cannot be used for variable argument parameters. Figure 4-15 shows this convention.

```
call(f)  proc_X
add  sp, immed

          ⇓

proc_X.numArgBytes = immed
sets CALL_C flag
```

Figure 4-14: C Calling Convention - Uses add

```
ret(f) immed

          ⇓

proc_X.numArgBytes = immed
sets CALL_PASCAL flag
```

Figure 4-15: Pascal Calling Convention

Long Variable Operations

Long variables are stored in memory as two consecutive memory or stack locations. These variables are normally identified when simple addition or subtraction operations are performed on them. The idioms used for these operations are generally used due to their simplicity in number of instructions.

Figure 4-16 shows the instructions involved in long addition. The low parts of the long variable(s) are added with an `add` instruction, which sets up the `carry` flag if there is an overflow. The high parts are then added taken into account the `carry` flag, as if there were an overflow of 1 in the low part, this 1 needs to be added to the high part. Thus, a `adc` (add with carry) instruction is used to add the high parts.

```
        add ax, [bp-4]
        adc dx, [bp-2]

              ⇓

dx:ax = dx:ax + [bp-2]:[bp-4]
```

Figure 4-16: Long Addition

In a similar way, long subtraction is performed. The low parts are first subtracted with a sub instruction. If there is a borrow, the carry flag is set. Such underflow is taken into consideration when subtracting the high parts, as if there were an overflow in the low part, a borrow needs to be subtracted from the source high part operand. Thus, an sbb (subtract with borrow) instruction is used. Figure 4-17 shows this case.

```
sub ax, [bp-4]
sbb dx, [bp-2]

      ⇓

dx:ax = dx:ax - [bp-2]:[bp-4]
```

Figure 4-17: Long Subtraction

The negation of a long variable is done by a sequence of 3 instructions: the high part is negated, then the low part is negated, and finally, zero is subtracted with borrow from the high part in case there was an underflow in the negation of the low part. This idiom is shown in Figure 4-18.

```
neg regH
neg regL
sbb regH, 0

      ⇓

regH:regL = - regH:regL
```

Figure 4-18: Long Negation

Long shifts by 1 are normally performed using the carry flag and rotating that flag onto the high or low part of the answer. A left shift is independent of the sign of the long operand, and generally involves the low part to be shifted left (shl), the high bit of the low part will be in the carry flag. The high part is then shifted left, but making use of the carry flag, which contains the bit to be placed on the lowest bit of the high part answer, thus, a rcl (rotate carry left) instruction is used. This idiom is shown in Figure 4-19.

A long right shift by 1 needs to retain the sign of the long operand, so two different idioms are used for signed and unsigned long operands. Figure 4-20 shows the idiom for signed long operands. The high part of the long operand is shifted right by 1, and an arithmetic shift right (sar) instruction is used, so that the number is treated as a signed number. The lower bit of the high part is placed on the carry flag. The low part of the operand is then shifted right, taking into account the bit in the carry flag, so a rotate carry right (rcr)

```
        shl regL, 1
        rcl regH, 1
             ⇓
regH:regL = regH:regL << 1
```

Figure 4-19: Shift Long Variable Left by 1

instruction is used.

```
                  sar regH, 1
                  rcr regL, 1
                       ⇓
regH:regL = regH:regL >> 1  (regH:regL is signed long)
```

Figure 4-20: Shift Signed Long Variable Right by 1

In a similar way, a long shift right by 1 of an unsigned long operand is done. In this case, the high part is shifted right, moving the lower bit into the `carry` flag. This bit is then shifted into the low part by a rotate carry right instruction. See Figure 4-21.

```
                  shr regH, 1
                  rcr regL, 1
                       ⇓
regH:regL = regH:regL >> 1  (regH:regL is unsigned long)
```

Figure 4-21: Shift Unsigned Long Variable Right by 1

Miscellaneous Idioms

A widely known machine idiom is the assignment of zero to a variable. Rather than using a `mov` instruction, an `xor` is used: whenever a variable is `xor`ed to itself, the result is zero. This machine idiom uses fewer machine cycles and bytes than its counterpart, and is shown in Figure 4-22.

```
xor reg/stackOff, reg/stackOff

              ⇓

     reg/stackOff = 0
```

Figure 4-22: Assign Zero

Different machine architectures restrict the number of bits that are shifted in the one shift instruction. In the case of the i8086, the shift instruction allows only one bit to be shifted in the one instruction, thus, several shift instructions have to be coded when shifting two or more bits. Figure 4-23 shows this idiom. In general, a shift by constant `n` can be done by `n` different shift 1 instructions.

```
shl reg, 1  --\
[...]         |  n times
shl reg, 1  --/

          ⇓

    reg = reg << n
```

Figure 4-23: Shift Left by n

Bitwise negation of an integer/word variable is done as shown in Figure 4-24. This idiom negates (2's complement) the register, then subtracts it from itself with borrow in case there was an underflow in the initial negation of the register, and finally increments the register by one to get a 0 or 1 answer.

```
neg reg
sbb reg, reg
inc reg

      ⇓

reg = !reg
```

Figure 4-24: Bitwise Negation

4.2.2 Simple Type Propagation

The sign of elementary data types such as byte and integer is easily determined by the type of conditional jump used to compare an operand. Such a technique is also used to determine the sign of more complex elementary data types such as long and real. The following sections illustrate the techniques used to determine whether a word-sized operand is a signed or unsigned integer, and whether a two word-sized operand is a signed or unsigned long. These techniques are easily extended to other elementary data types.

Propagation of Integers

A word-sized operand can be a signed integer or an unsigned integer. Most instructions that deal with word-sized operands do not make any distinction between signed or unsigned operands; conditional jump instructions are an exception. There are different types of conditional jumps for most relational operations, for example, the following code:

```
cmp  [bp-0Ah], 28h
jg   X
```

checks whether the word operand at `bp-0Ah` is greater than `28h`. The following code:

```
cmp  [bp-0Ah], 28h
ja   X
```

checks whether the word operand at `bp-0Ah` is above `28h`. This latter conditional jump tests for unsigned word operands, while the former conditional jump tests for signed word operands; hence, the local variable at `bp-0Ah` is a signed integer in the former case, and an unsigned integer in the latter case. This information is stored as an attribute of the local variable `bp-0Ah` in the symbol table.

In the same way, whenever the operands of a conditional jump deal with registers, the register is determined to be a signed or unsigned integer register, and this information is propagated backwards on the basic block to which the register belongs, up to the definition of the register. Consider the following code:

```
1  mov  ax, [bp-0Ch]
2  cmp  ax, 28h
3  ja   X
```

By instruction 3 the operands of the conditional jump are determined to be unsigned integers; hence, register `ax` and constant `28h` are unsigned integer operands. Since register `ax` is not a local variable, this information is propagated backwards until the definition of `ax` is found. In this example, instruction 1 defines `ax` in terms of local variable `bp-0Ch`, therefore, this local variable represents an unsigned integer and this attribute is stored in the symbol table entry for `bp-0Ch`.

The set of conditional jumps used to distinguish a signed from an unsigned integer are shown in Figure 4-25. These conditional jumps are for the Intel architecture.

Signed Conditional	Unsigned Conditional
jg	ja
jge	jae
jl	jb
jle	jbe

Figure 4-25: Sign Determination According to Conditional Jump

Propagation of Long Variables

The initial recognition of long variables is determined by idiom analysis, as described in Section 4.2.1. Once a pair of identifiers is known to be a long variable, all references to these identifiers must be changed to reflect them being part of a long variable (i.e. the high or low part of the long variable). Also, couples of instructions that deal with the high and low parts of the long variable can be merged into the one instruction. Consider the following code:

```
108  mov  dx, [bp-12h]
109  mov  ax, [bp-14h]
111  add  dx:ax, [bp-0Ah]:[bp-0Ch]
112  mov  [bp-0Eh], dx
113  mov  [bp-10h], ax
```

Instructions 110 and 111 were merged into the one `add` instruction by idiom analysis, leading to the identifiers `bp-0Ah` and `bp-0Ch` to become a long variable, as well as the registers `dx:ax`. Identifiers other than registers are propagated throughout the whole subroutine intermediate code, in this example, no other references to `bp-0Ah` are done. Registers are propagated within the basic block they were used in, by backward propagation until the register definition is found, and forward propagation until a redefinition of the register is done. In this example, by backward propagation of `dx:ax`, we arrive at the following code:

```
109  mov  dx:ax, [bp-12h]:[bp-14h]
111  add  dx:ax, [bp-0Ah]:[bp-0Ch]
112  mov  [bp-0Eh], dx
113  mov  [bp-10h], ax
```

which merges instructions 108 and 109 into the one `mov` instruction. Also, this merge has determined that the local identifiers `bp-12h` and `bp-14h` are a long variable, and hence, this information is stored in the symbol table. By forward propagation of `dx:ax` within the basic block we arrive at the following code:

```
109  mov  dx:ax, [bp-12h]:[bp-14h]
111  add  dx:ax, [bp-0Ah]:[bp-0Ch]
113  mov  [bp-0Eh]:[bp-10h], dx:ax
```

which merges instructions 112 and 113 into the one `mov` instruction. In this case, the local identifiers `bp-0Eh` and `bp-10h` are determined to be a long variable, and this information is also stored in the symbol table and propagated.

Propagation of long variables across conditional jumps is done in two or more steps. The high and low part of the long identifier are compared against another identifier in different basic blocks. The notion of basic block is simple: a sequence of instructions that have one entry and one exit point; this notion is explained in more detail in Section 4.4.3. Consider the following code:

```
115  mov  dx:ax, [bp-0Eh]:[bp-10h]
116  cmp  dx, [bp-0Ah]
117  jl   L21
118  jg   L22
119  cmp  ax, [bp-0Ch]
120  jbe  L21
```

At instruction 115, registers `dx:ax` are determined to be a long register, hence, the `cmp` opcode at instruction 116 is only checking for the high part of this long register, a further instruction (119) checks for the low part of the long register. By analysing the instructions it is seen that whenever `dx:ax` are less or equal to the identifier `[bp-0Ah]:[bp-0Ch]`, the label `L21` is reached; otherwise the label `L22` is reached. These three basic blocks can be transformed into a unique basic block that contains this condition, as follows:

```
115  mov  dx:ax, [bp-0Eh]:[bp-10h]
116  cmp  dx:ax, [bp-0Ah]:[bp-0Ch]
117  jle  L21
```

This basic block branches to label `L21` whenever the condition is true, and branches to label `L22` whenever the condition is false. The presence of label `L22` is not made explicit in the instructions, but is implicit in the out-edges of this basic block.

In general, long conditional branches are identified by their graph structure. Figure 4-26 shows five graphs. Four of these represent six different conditions. Graphs (a) and (b) represent the same condition. These graphs represent different long conditions depending on the instructions in the nodes associated with these graphs, the conditions are: $<=$, $<$, $>$, and $>=$. Graphs (c) and (d) present equality and inequality of long variables. These four graphs are translated into graph (e) when the following conditions are satisfied:

- Graphs (a) and (b):
  - Basic block x is a conditional node that compares the high parts of the long identifiers.
  - Basic block y is a conditional node that has one instruction ; a conditional jump, and has one in-edge; the one from basic block x.
  - Basic block z is a conditional node that has two instructions; a compare of the low parts of the long identifiers, and a conditional jump.
- Graphs (c) and (d):
  - Basic block x is a conditional node that compares the high parts of the long identifiers.

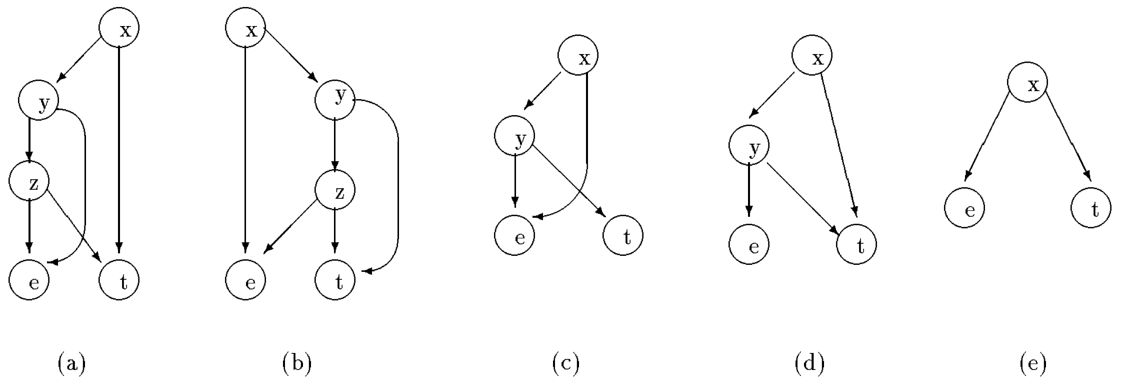


Figure 4-26: Long Conditional Graphs

 - Basic block y is a conditional node that has two instructions; a compare of the low parts of the long identifiers, and a conditional jump, and has one in-edge; the one from basic block x.

Figure 4-27 shows sample code for the graphs (c) and (d) of Figure 4-26; equality and inequality of long identifiers. This code is for the Intel i80286 architecture.

Node x	Node y	Boolean Condition
cmp dx, offHi jne t	cmp ax, offLow jne t	!=
cmp dx, offHi jne e	cmp ax, offLow je t	==

Figure 4-27: Long Equality Boolean Conditional Code

Sample code for the nodes of graph (a), Figure 4-26 is given in Figure 4-28. The code associated with each node represents different non-equality Boolean conditions, namely, less or equal, less than, greater than, and greater and equal. Similar code is used for the nodes of graph (b) which represent the same exact Boolean conditions. This code is for the Intel i80286 architecture.

4.3 Intermediate Code Generation

In a decompiler, the front-end translates the machine language source code into an intermediate representation which is suitable for analysis by the universal decompiling machine. Figure 4-29 shows the relation of this phase with the semantic analyzer and the last phase of the front-end; the control flow graph generator. A target language independent representation is used, so that retargeting to a different language is feasible, by writing a back-end for that language and attaching it to the decompiler.

Node x	Node y	Node z	Boolean Condition
cmp dx, offHi jl t	jg e	cmp ax, offLow jbe t	<=
cmp dx, offHi jl t	jne e	cmp ax, offLow jb t	<
cmp dx, offHi jg t	jne e	cmp ax, offLow ja t	>
cmp dx, offHi jg t	jl e	cmp ax, offLow jae t	>=

Figure 4-28: Long Non-Equality Boolean Conditional Code

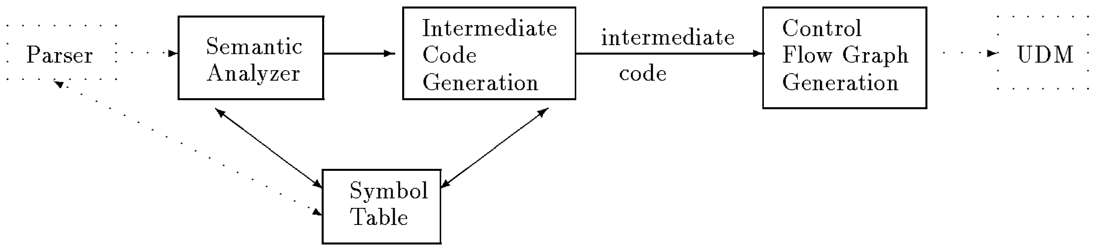


Figure 4-29: Interaction of the Intermediate Code Generator

A two-step approach is taken for decompilation: a low-level intermediate representation is first used to represent the machine language program. Idiom analysis and type propagation can be done in this representation, as well as generating assembler code from it (i.e. it is an intermediate code suitable for a disassembler, which does not perform high-level analysis on the code). This representation is then converted into a high-level intermediate representation that is suitable for high-level language generation. The representation needs to be general enough to generate code for any high-level language.

4.3.1 Low-level Intermediate Code

A low-level intermediate representation that resembles the assembler language for the machine that is being decompiled is a good choice of low-level intermediate code, as it is possible to perform semantic analysis on the code, as well as generate assembler programs from it. The intermediate code must have a one instruction for each complete instruction of the machine language. Compound machine instructions must also be represented by one intermediate instruction. For example, in Figure 4-30, the machine instruction `B720` is a `mov bh,20` intermediate instruction. The machine instruction `2E` followed by `FFEFC006` (a `jmp` with a `cs` segment override) is replaced by a `jmp` instruction that makes explicit the use of register `cs`. And finally, the compound machine instructions `F3A4` are equivalent to the assembler instructions `rep` and `movs di,si`. These two instructions are represented by the unique intermediate instruction `rep_movs`, which makes explicit the destination and source registers of the move.

```
                2E                  F3
B720            FFEFC006            A4

                       ⇓

mov bh,20       jmp cs:06C0[bx]     rep_movs di,si
```

Figure 4-30: Low-level Intermediate Instructions - Example

Implementation of Low-Level Intermediate Code

The low-level intermediate representation is implemented in quadruples which make explicit the operands use in the instruction, as shown in Figure 4-31. The `opcode` field holds the low-level intermediate opcode, the `dest` field holds the destination operand (i.e. an identifier), and the `src1` and `src2` fields hold the source operands of the instruction. Some instructions do not use two source operands, so only the `src1` field is used.

opcode	dest	src1	src2

Figure 4-31: General Representation of a Quadruple

Example 4 *An* `add bx,3` *machine instruction is represented in a quadruple in the following way:*

add	bx	bx	3

where register `bx` *is source and destination operand, and constant* `3` *is the second source operand.*

Example 5 *A* `push cx` *machine instruction is represented in the following way:*

push	sp	cx	

where register `cx` *is the source operand and register* `sp` *is the destination operand.*

4.3.2 High-level Intermediate Code

Three-address code is a generalized form of assembler code for a three-address machine. This intermediate code is most suited for a decompiler, given that the three-address code is a linearized representation of an abstract syntax tree (AST) of the program. In this way, the complete AST of the program can be reconstructed during the data flow analysis. A three-address instruction has the general form:

```
x := y op z
```

where `x`, `y`, and `z` are identifiers, and `op` is an arithmetic or logic operation. The result address is `x`, and the two operand addresses are `y` and `z`.

Types of Three-Address Statements

Three-address statements are similar to high-level language statements. Given that the data flow analysis will reconstruct the AST of the program, a three-address instruction is going to represent not only individual identifiers, but expressions. An identifier can be viewed as the minimal form of an expression. The different types of instructions are:

1. `asgn` <exp>, <arithExp>
   The `asgn` instruction assigns an arithmetic expression to an identifier or an expression (i.e. an identifier that is represented by an expression, such as indexing into an array). This statement represents three different types of high-level assignment instructions:

   - `x := y op z`. Where `x`, `y`, and `z` are identifiers, and `op` is a binary arithmetic operator.
   - `x := op y`. Where `x` and `y` are identifiers, and `op` is a unary arithmetic operator.
   - `x := y`. Where `x` and `y` are identifiers.

   After data flow analysis, the arithmetic expression represents not only a binary operation, but holds a complete parse tree of arithmetic operators and identifiers. This transformation is described in Chapter 5.

   In this context, a subroutine that returns a value (i.e. a function), is also considered an identifier, as its invocation returns a result that is assigned to another identifier (e.g. `a := sum(b,c)`).

2. `jmp`
   The unconditional jump instruction has no associated expression attached to it, other than the target destination address of the jump. This instruction transfers control to the target address. Since the address is coded in the out-edge of the basic block that includes this instruction, it is not explicitly described as part of the instruction. This instruction is equivalent to the high-level instruction:

   ```
   goto L
   ```

   where `L` is the target address of the jump.

3. `jcond` <boolExp>
   The conditional jump instruction has a Boolean expression associated with it, which determines whether the branch is taken or not. The Boolean expression is of the form `x relop y`, where `x` and `y` are identifiers, and `relop` is a relational operator, such as $<, \geq, =$. This statement is equivalent to the high-level statement:

   ```
   if x relop y goto L
   ```

   In this intermediate instruction, the target branch address (`L`) and the fall-through address (i.e. address of the next instruction) are not part of the instruction as these are coded in the out-edges from the basic block that has this instruction in the control flow graph.

4. `call` <procId> <actual parameters>
   The `call` instruction represents a subroutine call. The procedure identifier (<procId>) is a pointer to the flow graph of the invoked procedure. The actual parameter list is

constructed during data flow analysis. If the subroutine called is a function, it also defines the registers that hold the returned value. In this case, the instruction is equivalent to `asgn` <regs>, <procId> <actual parameters>.

5. `ret` [<arithExp>]
   The return instruction determines the end of a procedure along a path. If there is nothing to return, the subroutine is a procedure, otherwise it is a function.

There are also two pseudo high-level intermediate instructions that are used as intermediate instructions in the data flow analysis, but are eliminated by the end of the analysis. These instructions are:

1. `push` <arithExp>
   The `push` instruction places the associated arithmetic expression on a temporary stack.
2. `pop` <ident>
   The `pop` instruction takes the expression or identifier at the top of the temporary stack and assigns it to the identifier ident.

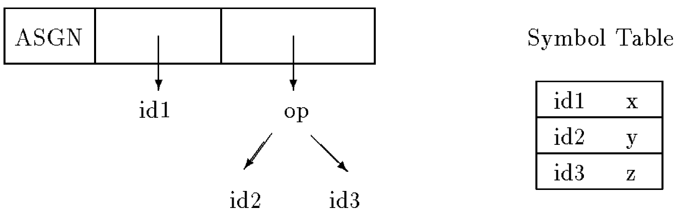
Implementation of High-Level Intermediate Code

The high-level intermediate representation is implemented by triplets. In a triplet, the two expressions are made explicit, as well as the instruction opcode, such as shown in Figure 4-32. The `result` and `arg` fields are pointers to an expression, which in its minimal form is an identifier which points to the symbol table.

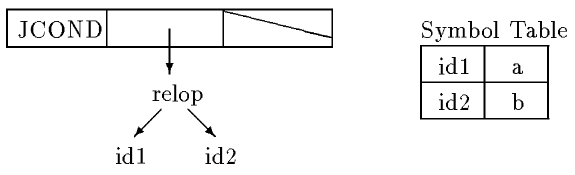
op	result	arg

Figure 4-32: General Representation of a Triplet

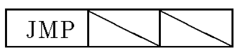
An assignment statement `x := y op z` is represented in a triplet in the following way: the `op` field is the `asgn` opcode, the `result` field has a pointer to the identifier `x` (which in turn has a pointer to the identifier in the symbol table), and the `arg` field has a pointer to a binary expression; this expression is represented by an abstract syntax tree with pointers to the symbol table entries of `y` and `z`, as follows:



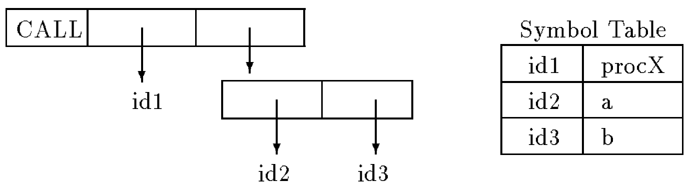
In a similar way, a conditional jump statement `if a relop b` is represented in a triplet in the following way: the `op` field is the `jcond` opcode, the `result` field has a pointer to the abstract syntax tree of the relational test, and the `arg` field is left empty, as follows:



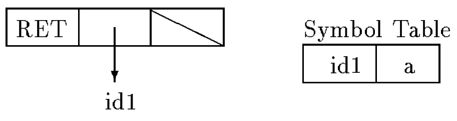
An unconditional jump statement `goto L` does not use the `result` or `arg` field. The `op` field is set to `jmp`, and the other fields are left empty, as follows:



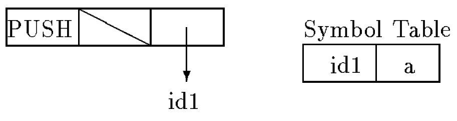
A procedure call statement `procX (a,b)` uses the `op` field for the `call` opcode, the `result` field for the procedure's name, which is pointed to in the symbol table, and the `arg` field for the procedure arguments, which is a list of arguments that point to the symbol table, as follows:



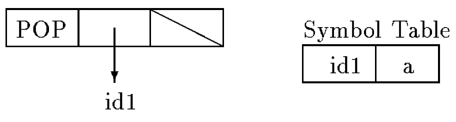
A procedure return statement `ret a` uses the `op` field for the `ret` opcode, the `result` field for the identifier/expression that is being returned, and the `arg` field is left empty, as follows:



The pseudo high-level instruction `push a` is stored in a triplet by using the `op` field as the `push` opcode, the `arg` field as the identifier that is being pushed, and the `result` field is left empty, as follows:



In a similar way, the `pop a` instruction is stored in a triplet, using the `op` field for the `pop` opcode, the `result` field for the identifier, and eventually (during data flow analysis), the `arg` field is filled with the expression that is being popped. Initially, this field is left empty. The triplet representation is as follows:



4.4 Control Flow Graph Generation

The control flow graph generation phase constructs a call graph of the source program, and a control flow graph of basic blocks for each subroutine of the program. These graphs are used to analyze the program in the universal decompiling machine (UDM) module. The interaction of this phase with the intermediate code generator and the udm is shown in Figure 4-33.

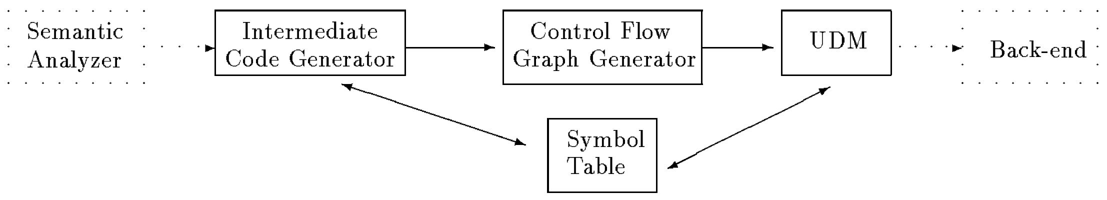


Figure 4-33: Interaction of the Control Flow Graph Generator

4.4.1 Basic Concepts

This section describes definitions of mathematical and graph theory. These terms are defined here to eliminate any ambiguity of terminology.

Definition 4 *A* **graph** *G is a tuple (V, E, h) where V is a set of nodes, E is a set of edges, and h is the root of the graph. An edge is a pair of nodes (v, w), with $v, w \in V$.*

Definition 5 *A* **directed graph** *$G = (N, E, h)$ is a graph that has directed edges; i.e. each $(n_i, n_j) \in E$ has a direction, and is represented by $n_i \rightarrow n_j$.*

Definition 6 *A* path *from n_1 to n_m in graph $G = (N, E, h)$, represented $n_1 \rightarrow^* n_n$, is a sequence of edges $(n_1, n_2), (n_2, n_3), \ldots, (n_{n-1}, n_m) \in N, m \geq 1$.*

Definition 7 *If $G = (V, E, h)$ is a graph, $\exists!\ h \in V$, and $E = \emptyset$, then G is called a* **trivial graph**.

Definition 8 *If $G = (N, E, h)$ is a graph, and $\forall n \in N, h \rightarrow^* n$, then G is a* **connected graph**.

A connected graph is a graph in which all nodes can be reached from the header node. A sample directed, connected graph is shown in Figure 4-34.

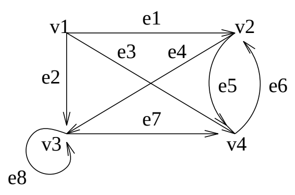


Figure 4-34: Sample Directed, Connected Graph

Graph Representation

A graph $G = (V, E, h)$ is represented in several different ways, including, incidence matrices, adjacency matrices, and predecessor-successor tables.

Definition 9 *The incidence matrix for a graph $G = (V, E, h)$ is the $v \times e$ matrix $M(G) = [m_{ij}]$, where m_{ij} is the number of times (0, 1 or 2) that vertex v_i and edge e_j are incident.*

Definition 10 *The adjacency matrix for a graph $G = (V, E, h)$ is the $v \times v$ matrix $A(G) = [a_{ij}]$, where a_{ij} is the number of edges joining vertices v_i and v_j.*

Definition 11 *The predecessor-successor table for a graph $G = (V, E, h)$ is the $v \times 2$ table $T(G) = [t_{i1}, t_{i2}]$, where t_{i1} is the list of predecessor vertices of vertex v_i, and t_{i2} is the list of successor vertices of vertex v_i.*

Example 6 *The graph in Figure 4-34 is represented by the following matrices:*

- *Incidence matrix:*

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
v_1	1	1	1	0	0	0	0	0
v_2	1	0	0	1	1	1	0	0
v_3	0	1	0	1	0	0	1	2
v_4	0	0	1	0	1	1	1	0

- *Adjacency matrix:*

	v_1	v_2	v_3	v_4
v_1	0	1	1	1
v_2	1	0	1	2
v_3	1	1	1	1
v_4	1	2	1	0

- *Predecessor successor table:*

	Predecessor	*Successor*
v_1	$\emptyset$	$\{v_2, v_3, v_4\}$
v_2	$\{v_1, v_4\}$	$\{v_3, v_4\}$
v_3	$\{v_1, v_2, v_3\}$	$\{v_3, v_4\}$
v_4	$\{v_1, v_2, v_3\}$	$\{v_2\}$

4.4.2 Basic Blocks

In this section we formalize the definition of basic blocks. In order to characterize it, we need some more definitions of program structure. We start by defining the components of a program, that is, data and instructions. Note that this definition does not associate instructions or data to memory locations.

Definition 12 *Let*

- *P be a program*
- *$I = \{i_1, \ldots, i_n\}$ be the instructions of P*
- *$D = \{d_1, \ldots, d_m\}$ be the data of P*

Then $P = I \bigcup D$

For the purpose of this research, programs are restricted to containing no self-modifying code, and make no use of data as instructions or vice versa ($I(P) \bigcap D(P) = \emptyset$). An instruction sequence is a set of instructions physically located one after the other in memory.

Definition 13 *Let*

- *P be a program*
- *$\mathcal{I} = \{i_1, \ldots, i_n\}$ be the instructions of P*

Then $\mathcal{S}$ is an **instruction sequence** *if and only if*
$\mathcal{S} = [i_j, i_{j+1}, \ldots, i_{j+k}] \bullet 1 \leq j < j + k \leq n \wedge i_{j+1}$ is in a consecutive memory location to $i_j, \forall 1 \leq j \leq k - 1$.

Intermediate instructions are classified in two sets for the purposes of control flow graph generation:

- Transfer Instructions (TI): the set of instructions that transfer flow of control to an address in memory different from the address of the next instruction. These instructions are:
  - Unconditional jumps: the flow of control is transferred to the target jump address.
  - Conditional jumps: the flow of control is transferred to the target jump address if the condition is true, otherwise the control is transferred to the next instruction in the sequence.
  - Indexed jumps: the flow of control is transferred to one of many target addresses.
  - Subroutine call: the flow of control is transferred to the invoked subroutine.
  - Subroutine return: the flow of control is transferred to the subroutine that invoked the subroutine with the return instruction.
  - End of program: the program ends.
- Non transfer instructions (NTI): the set of instructions that transfer control to the next instruction in the sequence, i.e. all instructions that do not belong to the TI set.

Having classified the intermediate instructions, a basic block is defined in terms of its instructions:

Definition 14 *A* **basic block** *$b = [i_1, \ldots, i_{n-1}, i_n], n \geq 1$ is an instruction sequence that satisfies the following conditions:*

1. $[i_1, \ldots, i_{n-1}] \in NTI$

2. $i_n \in TI$

or

1. $[i_1, \ldots, i_{n-1}, i_n] \in NTI$

2. i_{n+1} *is the first instruction of another basic block.*

A basic block is a sequence of instructions that has one entry point and one exit point. If one instruction of the basic block is executed, all other instructions are executed as well.

The set of instructions of a program can be uniquely partioned into a set of non-overlapping basic blocks, starting from the program's entry point.

Definition 15 *Let*

- *I be the instructions of program P*
- *h be P's entry point*

Then $\exists \mathcal{B} = \{b_1, \ldots, b_n\} \bullet b_1 \bigcap b_2 \ldots \bigcap b_n = \emptyset \wedge \mathcal{I} = b_1 \bigcup b_2 \bigcup \ldots \bigcup b_n \wedge b_1$*'s entry point* $= h$.

4.4.3 Control Flow Graphs

A control flow graph is a directed graph that represents the flow of control of a program, thus, it only represents the instructions of such a program. The nodes of this graph represent basic blocks of the program, and the edges represent the flow of control between nodes. More formally,

Definition 16 *A* **control flow graph** $G = (N, E, h)$ *for a program P is a connected, directed graph, that satisfies the following conditions:*

- $\forall n \in N, n$ *represents a basic blocks of P.*
- $\forall e = (n_i, n_j) \in E, e$ *represents flow of control from one basic block to another and* $n_i, n_j \in N$.
- $\exists f : \mathcal{B} \to N \bullet \forall b_i \in \mathcal{B}, f(b_i) = n_k$ *for some* $n_k \in N \wedge \nexists b_j \in \mathcal{B} \bullet f(b_j) = n_k$

For the purpose of control flow graph (cfg) generation, basic blocks are classified into different types, according to the last instruction of the basic block. The available types of basic blocks are:

- 1-way basic block: the last instruction in the basic block is an unconditional jump. The block has one out-edge.
- 2-way basic block: the last instruction is a conditional jump, thus, the block has two out-edges.
- n-way basic block: the last instruction is an indexed jump. The n branches located in the `case` table become the n out-edges of this node.

- call basic block: the last instruction is a call to a subroutine. There are two out-edges from this block: one to the instruction following the subroutine call (if the subroutine returns), and the other to the subroutine that is called.
- return basic block: the last instruction is a procedure return or an end of program. There are no out-edges from this basic block.
- fall basic block: the next instruction is the target address of a branching instruction (i.e. the next instruction has a label). This node is seen as a node that *falls through* the next one, thus, there is only one out-edge.

The different types of basic block are represented in a control flow graph by named nodes, as shown in Figure 4-35. Whenever a node is not named in a graph, it means that the type of the basic block is irrelevant, or obvious from the context (i.e. the exact number of out-edges are specified in the graph).

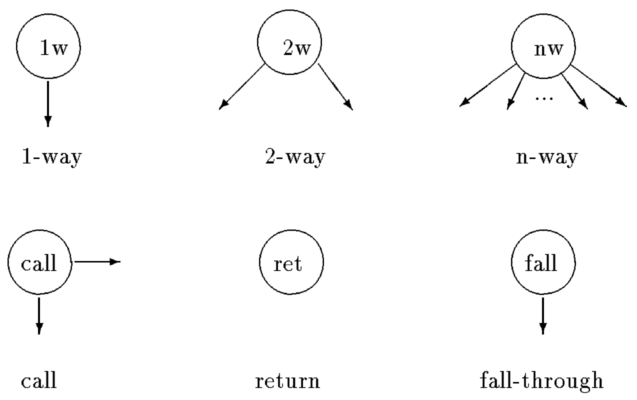


Figure 4-35: Node Representation of Different Types of Basic Blocks

Example 7 *Consider the following fragment of code:*

```
 0      PUSH            bp
 1      MOV             bp, sp
 2      SUB             sp, 4
 3      MOV             ax, 0Ah
 4      MOV             [bp-2], ax
 5      MOV             [bp-4], ax
 6      LEA             ax, [bp-4]
 7      PUSH            ax
 8      CALL  near ptr  proc_1
 9      POP             cx
10 L1:  MOV             ax, [bp-4]
11      CMP             ax, [bp-2]
12      JNE             L2
13      PUSH  word ptr  [bp-4]
14      MOV             ax, 0AAh
15      PUSH            ax
16      CALL  near ptr  printf
```

```
17        POP              cx
18        POP              cx
19        MOV              sp, bp
20        POP              bp
21        RET
22 L2:    MOV              ax, [bp-4]
23        CMP              ax, [bp-2]
24        JGE              L1
25        LEA              ax, [bp-2]
26        PUSH             ax
27        CALL  near ptr   proc_1
28        POP              cx
29        JMP              L1
```

This code has the following basic blocks:

Basic Block Type	*Instruction Extent*
call	*0 to 8*
fall	*9*
2w	*10 to 12*
call	*13 to 16*
ret	*17..21*
2w	*22..24*
call	*25..27*
1w	*28, 29*

The control flow graph that represents these instructions is shown in Figure 4-36.

From here onwards, the word graph is used to represent a control flow graph, and the word node is used to represent a basic block, unless otherwise stated.

Implementation

Control flow graphs have on average close 2 out edges per node, thus, a matrix representation (e.g. incident and adjacent matrices) is very sparse and memory inefficient (i.e. most of the matrix is zero). It is therefore better to implement control flow graphs in predecessor-successor tables, so that only the existing edges in the graph are represented in this relationship. Note that the successor is all that is needed to represent the complete graph; the predecessor is also stored to make access to the graph easily during different traversals of the graph.

If the size of the graph is unknown (i.e. the number of nodes is not fixed), it is possible to construct the graph dynamically as a pointer to a basic block, which has a list of predecessor and a list of successors attached to it. The predecessors and successors are pointers to basic block nodes as well; in this way, a basic block is only represented once. This representation is plausible in any high-level language that allows dynamic allocation of memory. Consider the C definition of a basic block in Figure 4-37. The `BB` structure defines a basic block node, the `numInEdges` and `numOutEdges` hold the number of predecessor and successor nodes,

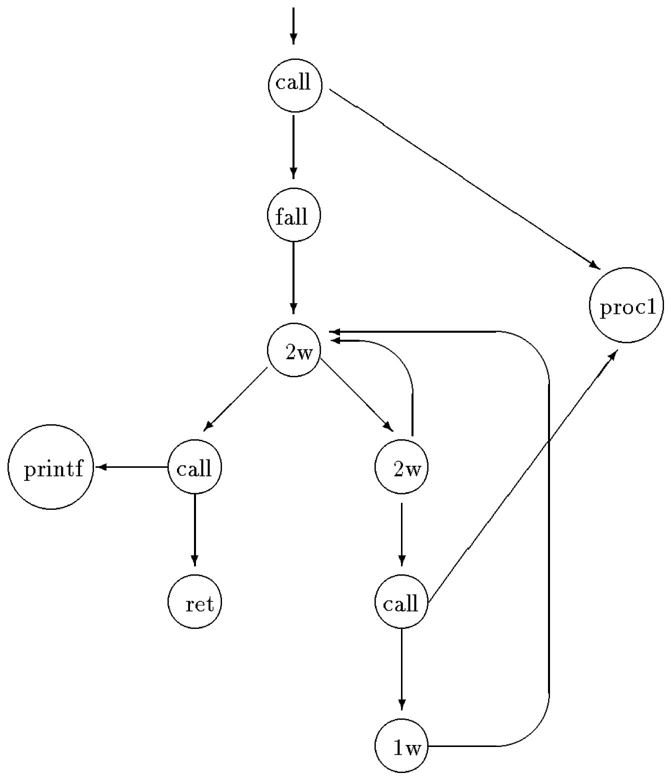


Figure 4-36: Control Flow Graph for Example 10

respectively, the **inEdges is a dynamically allocated array of pointers to predecessor basic blocks, and the **outEdges is a dynamically allocated array of pointers to successor basic blocks. In this representation, a graph is a pointer to the header basic block (i.e. a PBB).

```
typedef struct _BB{
   byte       nodeType;     /* Type of node                       */
   int        numInEdges;   /* Number of in edges                 */
   struct _BB **inEdges;    /* Array of pointers to predecessors */
   int        numOutEdges;  /* Number of out-edges                */
   struct _BB **outEdges;   /* Array of pointers to successors   */
   /* other fields go here */
} BB;
typedef BB *PBB;            /* Pointer to a basic block          */
```

Figure 4-37: Basic Block Definition in C

Graph Optimization

One pass compilers generate machine code that makes use of redundant or unnecessary jumps in the form of jumps to jumps, conditional jumps to jumps, and jumps to conditional jumps. These unnecessary jumps can be eliminated by a peephole optimization on flow-of-control. This optimization is not always used, though.

Peephole optimization is a method for improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence of instructions. The peephole is a small, moving window of target code; the code in the peephole does not need to be contiguous. Each improvement made through a peephole optimization may spawn opportunities for additional improvements, thus, repeated passes over the code is necessary.

Flow-of-control optimization is the method by which redundant jumps are eliminated. For decompilation, we are interested in eliminating all jumps to jumps, and conditional jumps to jumps, as the target jump holds the address of the target branch, and makes use of an intermediate basic blocks that can be removed from the graph. The removal of jumps to conditional jumps is not desired, as it involves the rearrangement of several instructions, not just the modification of one target branch address.

The following jump sequence jumps to label `Lx` to jump to label `Ly` afterwards, without any other instructions executed between the jumps:

```
     jmp Lx
     ...        ; other code here
Lx:  jmp Ly
```

This sequence is replaced by the sequence

```
     jmp Ly
     ...        ; other code here
Lx:  jmp Ly
```

where the first jump branches to the target `Ly` label, rather than the intermediate `Lx` label. The number of predecessors to basic block starting at `Lx` is decremented by one, and the number of predecessors to block startin at `Ly` is incremented by one, to reflect the change of edges in the graph. If at any one time the number of predecessors to the basic block starting at label `Lx` becomes zero, the node is removed from the graph because it is unreachable, and thus was unnecessary in the first place.

In a similar way, an unconditional jump to a jump sequence like the following

```
     jZ  Lx
     ...       ; other code here
Lx:  jmp Ly
```

is replaced by the code sequence

```
     jZ  Ly
     ...       ; other code here
Lx:  jmp Ly
```

The Call Graph

The call graph is a mathematical representation of the subroutines of a program. Each node represents a subroutine, and each edge represents a call to another subroutine. More formally,

Definition 17 *Let* $\mathcal{P}=\{p_1, p_2, \ldots\}$ *be the finite set of procedures of a program. A* call graph *C is a tuple (N, E, h), where N is the set of procedures and $n_i \in N$ represents one and only one $p_i \in \mathcal{P}$, E is the set of edges and $(n_i, n_j) \in E$ represents one or more references of p_i to p_j, and h is the main procedure.*

The construction of the call graph is simple if the invoked subroutines are statically bound to subroutine constants, that is, the program does not make use of procedure parameters or procedure variables. The presence of recursion introduces cycles in the call graph. An efficient algorithm to construct the call graph in the presence of procedure parameters, for languages that do not have recursion is given in [Ryd79]. This method was later extended to support recursion, and is explained in [CCHK90]. Finally, a method that handles procedure parameters and a limited type of procedure variables is described in [HK92].

It should be noted that this method is not able to reconstruct all graphs to their original form, as a compiler optimisation could have changed an implicit `call` instruction into an unconditional jump. In these cases, the call graph (and hence the decompiler) will treat the code of both subroutines as being one, unless the invoked subroutine is also called elsewhere via an implicit `call` instruction.

Chapter 5

Data Flow Analysis

The low-level intermediate code generated by the front-end is an assembler-type representation that makes use of registers and condition codes. This representation can be transformed into a higher level representation that does not make use of such low-level concepts, and that regenerates the high-level concept of expression. The transformation of low-level to high-level intermediate code is done by means of program transformations, traditionally referred to as *optimizations*. These transformations are applied to the low-level intermediate code, to transform it into the high-level intermediate code described in Chapter 4, Section 4.3.2. The relation of this phase with the front-end and the control flow analysis phase is shown in Figure 5-1.

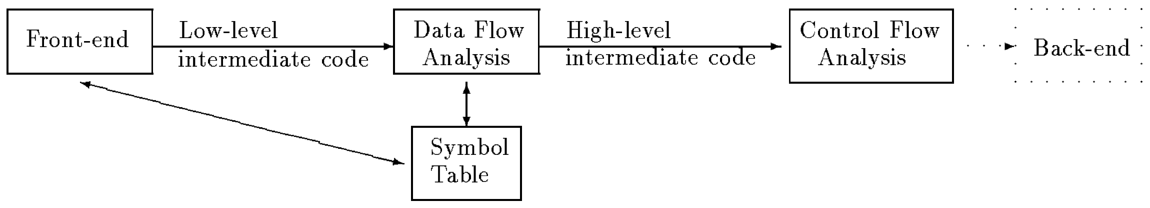


Figure 5-1: Context of the Data Flow Analysis Phase

The types of transformations that are required by the data flow analysis phase include, the elimination of useless instructions, the elimination of condition codes, the determination of register arguments and function return register(s), the elimination of registers and intermediate instructions by the regeneration of expressions, the determination of actual paramaters, and the propagation of data type across subroutine calls. Most of these transformations are required to improve the quality of the low-level intermediate code, and to reconstruct some of the information lost during the compilation process. In the case of the elimination of useless instructions, this step is required even for optimising compilers when there exist machine instructions that perform more than one function at a time (for an example, refer to Section 5.2.1).

Conventional data flow analysis is the process of collecting information about the way variables are used in a program, and summarizing it in the form of sets. This information is used by the decompiler to transform and improve the quality of the intermediate code. Several properties are required by code-improving transformations, including[ASU86b]:

1. A transformation must preserve the meaning of programs.

2. A transformation must be worth the effort.

Techniques for decompilation optimization of the intermediate code are presented in this chapter. The transformations are firstly illustrated by means of examples, and algorithms are later provided for each optimization.

5.1 Previous Work

Not much work has been done in the area of data flow analysis of a decompiler, mainly due to the limitations placed on many of the decompilers available in the literature; decompilation of assembler source files[Hou73, Fri74, Wor78, Bri81], decompilation of object files with symbolic debugging information[Reu88], and the compiler specification requirements to build a decompiler[BB91, Bow93, BB93]. Data flow analysis is essential when decompiling pure binary files, as there is no extra information on the way data is used, and the type of it. The following sections summarize all the work that has been done in this area.

5.1.1 Elimination of Condition Codes

A program which translates microprocessor object code (i8085) into a behaviorally equivalent PL/1 program was described by Marshall and Zobrist[MZBR85], and was used for electronic system simulation. The final PL/1 programs contained a large number of statements that defined flags, even if these flags were not used or referenced later on the program. This prompted DeJean and Zobrist to formulate an optimization of flag definitions by means of a reach algorithm[DZ89]. This method eliminated over 50% of the flag definitions in the translation process, generating PL/1 programs that defined only the necessary flags for a later condition.

The method presented in this thesis goes beyond this optimization of flag definitions, in that it not only determines which flag definitions are extraneous and therefore unnecessary, but also determines which Boolean conditional expression is represented by the combined set of instructions that define and use the flag. In this way, the target HLL program does not rely on the use and concept of flags, as any real HLL program does not.

5.1.2 Elimination of Redundant Loads and Stores

A method of *text compression* was presented by Housel[Hou73] for the elimination of intermediate loads and stores. This method works on a 3-address intermediate representation of the program, and consists of two stages: forward-substitution and backward-substitution. The former stage substitutes the source operand of an assignment instruction into a subsequent instruction that uses the same result operand, if the result is found to be not busy within the same basic block. The latter stage substitutes the result operand of an assignment instruction into a previous instruction (other than an assignment instruction) that defines as result operand the source operand of the assignment instruction under consideration. This method provided a reduction of instruction of up to 40% in assembly code compiled by Knuth's MIXAL compiler.

A method of *expression condensation* was described by Hopwood[Hop78] to combine 2 or more intermediate instructions into an equivalent expression by means of forward substitution. This method specifies 5 necessary conditions and 6 sufficient conditions under which forward substitution of a variable or register can be performed. This method was based on variable usage analysis. The great number of conditions is inherent to the choice of control flow graph: one node per intermediate instruction, rather than basic blocks. This meant that variables were forward substituted across node boundaries, making the whole process much more complex than required.

The interprocedural data flow analyses presented in this thesis define two sufficient conditions under which a register can be substituted or replaced into another instruction, including such intermediate instructions as `push` *and* `pop`*. This method not only finds expressions by eliminating intermediate registers and instruction definitions, but also determines actual parameters of subroutines, values returned from functions, and eliminates pseudo high-level instructions. The method is based on the initial high-level intermediate representation of the binary program, which is semantically equivalent to the low-level intermediate representation, and transforms it into a HLL representation.*

5.2 Types of Optimizations

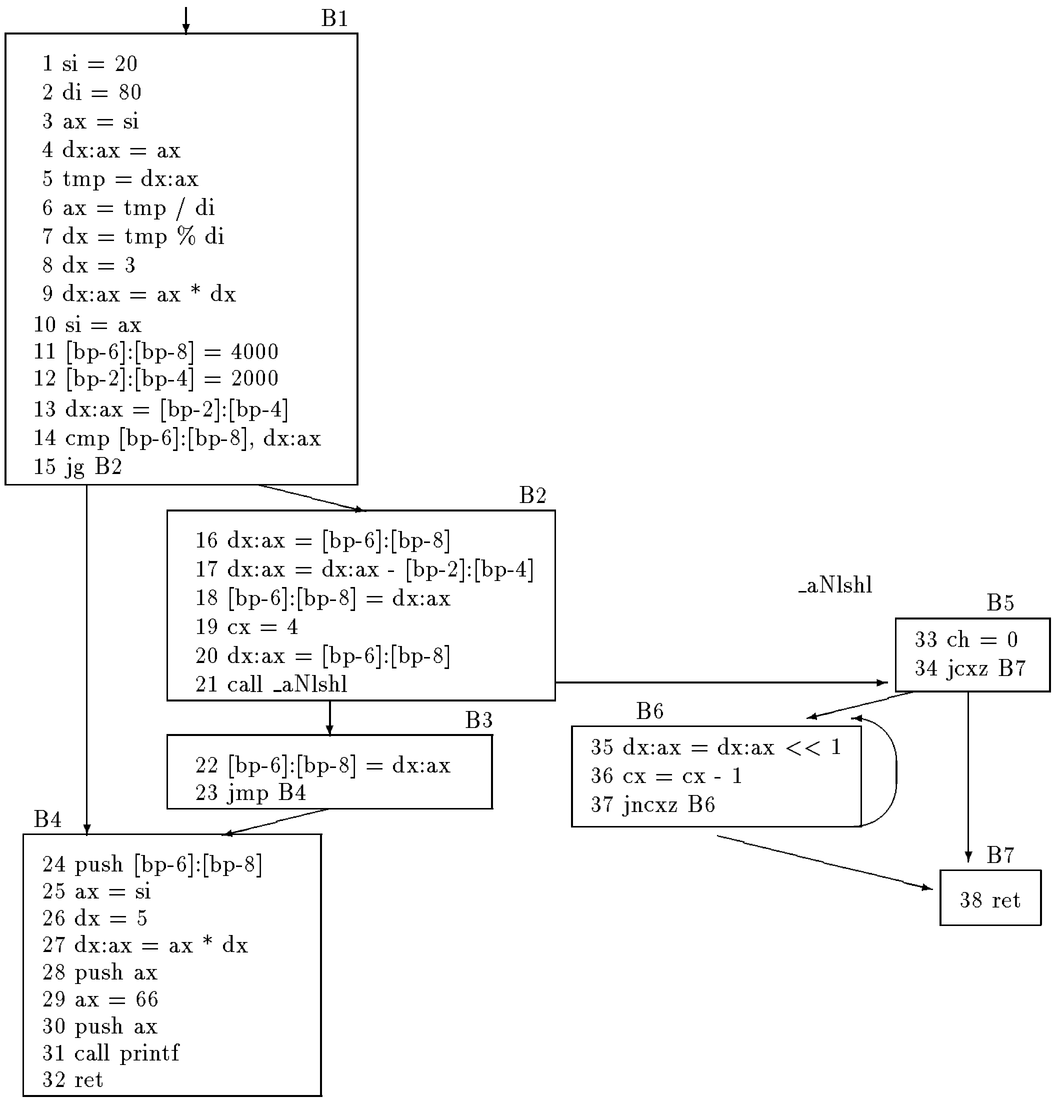
This section presents the code-improving transformations used by a decompiler. The techniques used to implement these transformations are explained in Sections 5.3 and 5.4. The optimizations presented in this section make use of the example flow graph in Figure 5-2, where basic blocks B1 ... B4 belong to the main program, and blocks B5 ... B7 belong to the subroutine `_aNlshl` (a runtime support routine). In the main program, registers `si` and `di` are used as register variables, and have been flagged by the parser as possibly being so (see Chapter 4, Section 4.2.1).

The aim of these optimizations is to eliminate the low-level language concepts of condition codes, registers, and intermediate instructions, and introduce the high-level concept of expressions of more than two operands. For this purpose, it is noted that `push` instructions are used in a variety of ways by today's compilers. Parameter passing is the most common use of this instruction, by `push`ing them before the subroutine call, in the order specified by the calling convention in use. Register spilling is used whenever the compiler runs out of registers to compute an expression. `push` and `pop` are also used to preserve the contents of registers across procedure calls, and to copy values into registers.

5.2.1 Dead-Register Elimination

An identifier is dead at a point in a program if its value is not used following the definition of the variable. It is said that the instruction that defines a dead identifier is useless, and thus can be eliminated or removed from the code. Consider the following code from basic block B1, Figure 5-2:

```
6  ax = tmp / di
7  dx = tmp % di
8  dx = 3
9  dx:ax = ax * dx
```



```
1 si = 20
2 di = 80
3 ax = si
4 dx:ax = ax
5 tmp = dx:ax
6 ax = tmp / di
7 dx = tmp % di
8 dx = 3
9 dx:ax = ax * dx
10 si = ax
11 [bp-6]:[bp-8] = 4000
12 [bp-2]:[bp-4] = 2000
13 dx:ax = [bp-2]:[bp-4]
14 cmp [bp-6]:[bp-8], dx:ax
15 jg B2

16 dx:ax = [bp-6]:[bp-8]
17 dx:ax = dx:ax - [bp-2]:[bp-4]
18 [bp-6]:[bp-8] = dx:ax
19 cx = 4
20 dx:ax = [bp-6]:[bp-8]
21 call _aNlshl

22 [bp-6]:[bp-8] = dx:ax
23 jmp B4

24 push [bp-6]:[bp-8]
25 ax = si
26 dx = 5
27 dx:ax = ax * dx
28 push ax
29 ax = 66
30 push ax
31 call printf
32 ret

33 ch = 0
34 jcxz B7

35 dx:ax = dx:ax << 1
36 cx = cx - 1
37 jncxz B6

38 ret
```

Figure 5-2: Sample Flow Graph

```
10 si = ax
```

Instruction 6 defines register `ax`, instruction 7 defines register `dx`, and instruction 8 redefines register `dx`. There is no use of register `dx` between the definition at instruction 7 and instruction 8, thus, the definition of register `dx` at instruction 7 is dead, and this instruction becomes useless since it defines only register `dx`. The previous sequence of instructions is replaced by the following code:

```
6  ax = tmp / di
8  dx = 3
9  dx:ax = ax * dx
10 si = ax
```

The definition of register `dx` at instruction 8 is used in the multiplication of instruction 9, where the register is redefined, as well as register `ax`. Instruction 10 uses register `ax`, and there are no further uses of register dx before redefinition of this register at instruction 13, thus, this last definition of dx is dead and must be eliminated. Since instruction 9 defines not only dx but also ax, and ax is not dead, the instruction is not useless as it still defines a live register; therefore, the instruction is modified to reflect the fact that only register ax is defined, as follows:

```
6  ax = tmp / di
8  dx = 3
9  ax = ax * dx
10 si = ax
```

5.2.2 Dead-Condition Code Elimination

In a similar way to dead-register elimination, a condition code is dead at a point in a program if its value is not used before redefinition. In this case, the definition of the condition code is useless, and is not required, but the instruction that defines this condition code is still useful if the identifiers that the instruction defines are not dead, hence, the instruction itself is not necessarily eliminated. Consider the following code from basic block B1, Figure 5-2:

```
14  cmp [bp-6]:[bp-8], dx:ax   ; cc-def = ZF, CF, SF
15  jg B2                      ; cc-use = SF
```

Instruction 14 defines the condition codes zero (ZF), carry (CF) and sign (SF). Instruction 15 uses the sign condition code. Neither of the following two basic blocks make use of the condition codes carry or zero before redefinition, thus, the definition of these condition codes in instruction 14 is useless and can be eliminated. We replace the information of instruction 14 to hold the following information:

```
14  cmp [bp-6]:[bp-8], dx:ax   ; cc-def = SF
```

5.2.3 Condition Code Propagation

Condition codes are flags used by the machine to signal the occurrence of a condition. In general, several machine instructions set these flags, ranging from 1 to 3 different flags being set by the one instruction, and fewer instructions make use of those flags, only using 1 or 2 flags. After dead-condition code elimination, the excess definitions of condition codes are eliminated, thus, all remaining flags are used by subsequent instructions. Consider the following code from basic block B1, Figure 5-2 after dead-condition code elimination:

```
14  cmp [bp-6]:[bp-8], dx:ax   ; cc-def = SF
15  jg B2                      ; cc-use = SF
```

Instruction 14 defines the sign flag by comparing two operands, and instruction 15 uses this flag to determine whether the first operand of the previous instruction was greater than the second operand. These two instructions are functionally equivalent to a high-level conditional jump instruction that checks for an operand being greater than a second operand. The instructions can be replaced by:

```
15  jcond ([bp-6]:[bp-8] > dx:ax) B2
```

eliminating instruction 14 and all references to the condition codes.

5.2.4 Register Arguments

Subroutines use register arguments to speed the access to those arguments and remove the overhead placed by the pushing of arguments on the stack before subroutine invocation. Register arguments are used by many runtime support routines, and by user routines compiled with the register calling convention (available in some compilers). Consider the following code of basic block B2, Figure 5-2:

```
19  cx = 4                  ; def = {cx}
20  dx:ax = [bp-6]:[bp-8]   ; def = {dx, ax}
21  call _aNlshl
```

Instruction 19 defines register cx, instruction 20 defines registers dx:ax , and instruction 21 invokes the subroutine _aNlshl. The first basic block of the subroutine _aNlshl, B5 in Figure 5-2, uses register cx after defining the high part of this register (i.e. register ch), thus, the low part of this register (i.e. register cl) contains whatever value the register had before the subroutine was invoked. In a similar way, basic block B6 uses registers dx:ax before they are defined within the subroutine, thus, the values of these registers before subroutine invocation are used. These three registers are used before being defined in the subroutine, and are defined by the caller, thus, they are register arguments to the _aNlshl subroutine. The formal argument list of this subroutine is modified to reflect this fact:

```
formal_arguments(_aNlshl) = {arg1 = dx:ax, arg2 = cl}
```

Within the subroutine, these registers are replaced by their formal argument name.

5.2.5 Function Return Register(s)

Subroutines that return a value are called functions. Functions usually return values in registers, and these registers are then used by the caller subroutine. Consider the following code from basic blocks B2 and B3, Figure 5-2:

```
20  dx:ax = [bp-6]:[bp-8]   ; def = {dx, ax}  use = {}
21  call _aNlshl            ; def = {}        use = {dx, ax, cl}
22  [bp-6]:[bp-8] = dx:ax   ; def = {}        use = {dx, ax}
```

Instruction 21 invokes the subroutine _aNlshl. After subroutine return, instruction 22 uses registers dx:ax. These registers have been defined in the previous basic block at instruction 20, but since there is a subroutine invocation in between these two instructions, the subroutine needs to be checked for any modification(s) to registers dx:ax. Consider the code of basic block B6, Figure 5-2 after dead-register elimination:

```
35  dx:ax = dx:ax << 1
36  cx = cx - 1
37  jcond (cx <> 0) B6
```

Recall from Section 5.2.4 that dx:ax are register arguments. These registers are modified in instruction 35 by a shift left. Actually, they form part of a loop as instruction 37 jumps back to the initial instruction 35 if register cx is not equal to zero. After the loop is finished, the flow of control is transfered to basic block B7, which returns from this subroutine. The reference to registers dx:ax in instruction 22 are the modified versions of these registers. We can think of subroutine _aNlshl as a function that returns both these registers, so the call to function _aNlshl in instruction 21 is replaced by:

```
21  dx:ax = call _aNlshl  ; def = {dx, ax}  use = {dx, ax, cl}
```

Instruction 22 uses the two registers defined in instruction 21, so, by register copy propagation, we arrive to the following code:

```
22  [bp-6]:[bp-8] = call _aNlshl
```

The return instruction of the function `_aNlshl` (instruction 38) is modified to return the registers `dx:ax`, leading to the following code:

```
38  ret dx:ax
```

5.2.6 Register Copy Propagation

An instruction is intermediate if it defines a register value that is used by a unique subsequent instruction. In machine language, intermediate instructions are used to move the contents of operands into registers, move the operands of an instruction into the registers that are used by a particular instruction, and to store the computed result in registers to a local variable. Consider the following code from basic block B2, Figure 5-2:

```
16  dx:ax = [bp-6]:[bp-8]           ; def = {dx, ax}  use = {}
17  dx:ax = dx:ax - [bp-2]:[bp-4]   ; def = {dx, ax}  use = {dx, ax}
18 [bp-6]:[bp-8] = dx:ax            ; def = {}        use = {dx, ax}
```

Instruction 16 defines the long register `dx:ax` by copying the contents of the long local variable at `bp-6`. This long register is then used in instruction 17 as an operand of a subtraction. The result is placed in the same long register, which is then copied to the long local variable at `bp-6` in instruction 18. As seen, instruction 16 defines the temporary long register `dx:ax` to be used in instruction 17, and this instruction redefines the register, and is then copied to the final local variable in instruction 18. These intermediate registers can be eliminated by replacing them with the local variable that was used to define them, thus, in instruction 17, registers `dx:ax` are replaced by the long local variable at `bp-6` which defined these registers in the previous instruction:

```
17  dx:ax = [bp-6]:[bp-8] - [bp-2]:[bp-4]
```

and instruction 16 is removed. In a similar way, the resultant long register `dx:ax` from instruction 17 is replaced in instruction 18, leading to the following code:

```
18  [bp-6]:[bp-8] = [bp-6]:[bp-8] - [bp-2]:[bp-4]
```

and instruction 17 is eliminated. The final instruction 18 is a reconstruction of the original high-level expression.

High-level language expressions are represented by parse trees of one or more operands, whereas machine language expressions allow only for at most two operands. In most cases, one of these operands needs to be in a register(s), and the result is also placed in a register(s). The final result is then copied to the appropriate identifier (i.e. local variable, argument, global variable). Consider the following code from basic block B1, Figure 5-2 after dead-register elimination:

```
3  ax = si         ; def = {ax}      use = {}
4  dx:ax = ax      ; def = {dx, ax}  use = {ax}
5  tmp = dx:ax     ; def = {tmp}     use = {dx, ax}
6  ax = tmp / di   ; def = {ax}      use = {tmp}
8  dx = 3          ; def = {dx}      use = {}
9  ax = ax * dx    ; def = {ax}      use = {ax, dx}
10 si = ax         ; def = {}        use = {ax}
```

Instruction 3 defines register `ax` by copying the contents of the integer register variable `si`. Register variables are treated as local variables rather than registers in this context. Instruction 4 uses register `ax` to define register `dx` by sign extension of register `ax`. Instruction 5 then uses these sign-extended registers to copy them to register `tmp`, which is used in instruction 6 as the dividend of a divide instruction. The local integer register variable `di` is used as the divisor, and the result is placed on register `ax`. This result is used in the multiplication in instruction 9, which also uses register `dx` and redefines register `ax`. Finally, the result is placed on the local register variable `si`. As seen, most of these instructions can be folded into a subsequent instruction, eliminating most of them as follows: instruction 3 is replaced into instruction 4, leading to:

```
4  dx:ax = si
```

and instruction 3 is eliminated. Instruction 4 is replaced into instruction 5, leading to:

```
5  tmp = si
```

and instruction 4 is eliminated. Instruction 5 is replaced into instruction 6, leading to:

```
6  ax = si / di
```

and instruction 5 is eliminated. Instruction 6 is replaced into instruction 9, leading to:

```
9  ax = (si / di) * dx
```

and instruction 6 is eliminated. Instruction 7 is replaced into instruction 9, leading to:

```
9  ax = (si / di) * 3
```

and instruction 7 is eliminated. Finally, instruction 9 is replaced into instruction 10, leading to the following final code:

```
10 si = (si / di) * 3
```

This final instruction 10 replaces all previous instructions 3 ... 10.

5.2.7 Actual Parameters

Actual parameters to a subroutine call are either pushed on the stack or placed on registers (for register arguments) before the subroutine is invoked. These arguments can be mapped against the formal argument list of the subroutine, and placed in the actual parameter list of the call instruction. Consider the following code from basic block B4, Figure 5-2 after register copy propagation:

```
24  push [bp-6]:[bp-8]
28  push (si * 5)
30  push 66
31  call printf
```

After parsing, the formal argument list of `printf` has one fixed argument of size 2 bytes, and a variable number of other arguments. The calling convention used for this procedure has been set to C. Instruction 31 has also saved the information regarding the number of bytes popped from the stack after subroutine call: 8 bytes in this case, thus, there are 8 bytes of actual arguments for this subroutine; the first 2 bytes are fixed. Instruction 24 pushes 4 bytes on the stack, instruction 28 pushes 2 bytes on the stack, and instruction 30 pushes another 2 bytes on the stack, for a total of 8 bytes required by `printf` in this instance. These identifiers can be replaced on the actual argument list of `printf` at instruction 31, in reverse order due to the C calling convention (i.e. last instruction pushed is the first one in the argument list). The modifications lead to the following code:

```
31  call printf (66, si * 5, [bp-6]:[bp-8])
```

and instructions 24, 28 and 30 are eliminated.

In a similar way, register arguments are placed on the actual argument list of the invoked subroutine. Consider the following code of basic blocks B2 and B3, Figure 5-2 after register argument and function return register detection and dead-register elimination:

```
19  cl = 4                          ; def = {cl}
20  dx:ax = [bp-6]:[bp-8]           ; def = {dx, ax}
22  [bp-6]:[bp-8] = call _aNlshl    ; use = {dx, ax, cl}
```

Instruction 19 and 20 define the register arguments used by function `_aNlshl`, the associated register definitions are placed in the function's actual argument list in the following way:

```
22  [bp-6]:[bp-8] = call _aNlshl ([bp-6]:[bp-8], 4)
```

eliminating instructions 19 and 20, and intermediate registers `dx`, `ax`, and `cl`.

5.2.8 Data Type Propagation Across Procedure Calls

The type of the actual arguments of a subroutine needs to be the same as the type of the formal arguments. In the case of library subroutines, the formal argument types are known with certainty, and thus, these types need to be matched against the actual types. If there are any differences, the formal type is to be propagated to the actual argument. Consider the following code from basic block B4, Figure 5-2 after register copy propagation and the detection of actual parameters:

```
31  call printf (66, si * 5, [bp-6]:[bp-8])
```

The first formal argument type of `printf` is a string (i.e. a `char *` in C). Strings are stored in machine language as data constants in the data or code segment. These strings are referenced by accessing the desired segment and an offset within that segment. In our example, `66` is a constant, and since it is the first argument to `printf` it is really an offset to a string located in the data segment. The string type is propagated to this first argument, the string is found in memory, and replaced in the actual argument list, leading to the following code:

```
31  call printf ("c * 5 = %d, a = %ld\n", si * 5, [bp-6]:[bp-8])
```

All other arguments to `printf` have undetermined type from the point of view of the formal argument list, so the types that the actual arguments have are trusted (i.e. the types used in the caller) and are not modified.

5.2.9 Register Variable Elimination

The register copy propagation optimization finds high-level expressions and eliminates intermediate instructions by eliminating most of the intermediate registers used in the computation of the expression, as seen in Section 5.2.6. After this optimization has been applied, there are only a few registers left (if any) in the intermediate code. These remaining registers represent register variables or common subexpressions, used by the compiler or the optimizer to speed up access time. These registers are equivalent to local variables in a high-level program, and are therefore replaced by new local variables in the corresponding subroutine that uses them. Consider the following code from basic block B1, Figure 5-2 after register copy propagation:

```
1  si = 20
2  di = 80
10 si = si / di * 3
```

Registers `si` and `di` are used as register variables in this procedure. These registers are initialized in instructions 1 and 2, and are later used in the expression of instruction 10. Let us rename register `si` by `loc1` and register `di` by `loc2`, then the previous code would look like:

```
1  loc1 = 20
2  loc2 = 80
10 loc1 = loc1 / loc2 * 3
```

and all references to registers have been eliminated.

After applying all of the previously explained transformations, the final intermediate code for Figure 5-2 is shown in Figure 5-3.

5.3 Global Data Flow Analysis

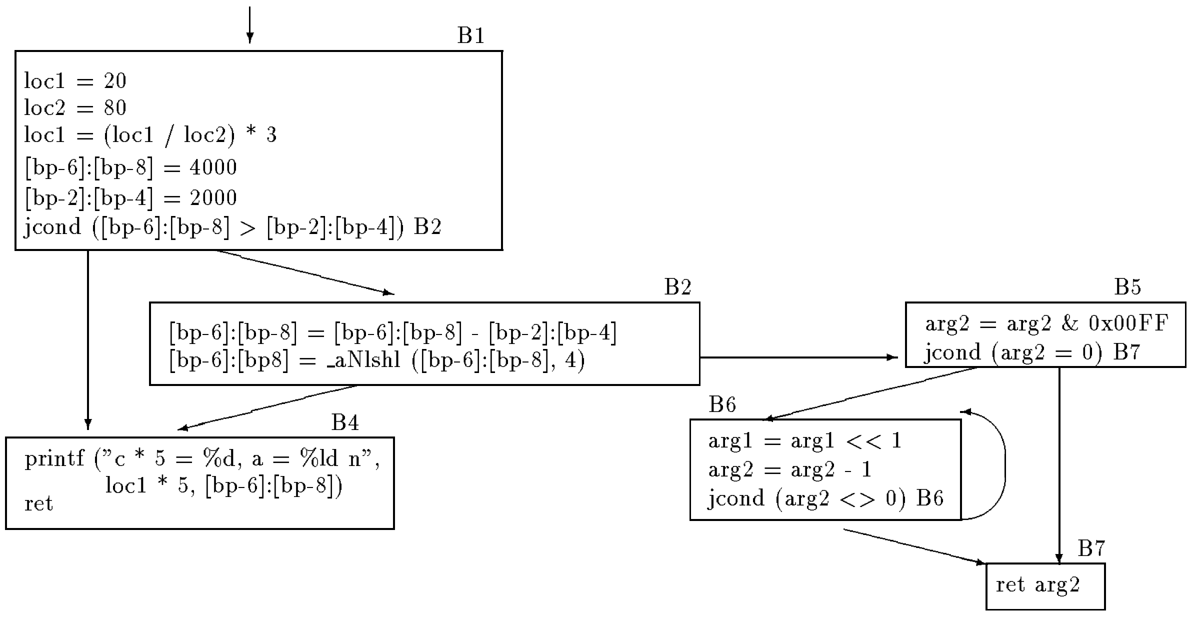
In order to perform code-improving transformations on the intermediate code, the decompiler needs to collect information on registers and condition codes about the whole program, and propagate this information across the different basic blocks. The information is collected by a *data flow analysis* process, which solves systems of equations that relate information at various points in the program. This section defines data flow problems and equations available in the literature; refer to [All72, AC76, ASU86b, FJ88b] for more information.

5.3.1 Data Flow Analysis Definitions

Definition 18 *A register is* **defined** *if the content of the register is modified (i.e. it is assigned a new value). In a similar way, a flag is defined if it is modified by an instruction.*

Definition 19 *A register is* **used** *if the register is referenced (i.e. the value of the register is used). In a similar way, a flag is used if it is referenced by an instruction.*

Definition 20 *A* **locally available definition** *d in a basic block B_i is the last definition of d in B_i.*



```
loc1 = 20
loc2 = 80
loc1 = (loc1 / loc2) * 3
[bp-6]:[bp-8] = 4000
[bp-2]:[bp-4] = 2000
jcond ([bp-6]:[bp-8] > [bp-2]:[bp-4]) B2
```

```
[bp-6]:[bp-8] = [bp-6]:[bp-8] - [bp-2]:[bp-4]
[bp-6]:[bp8] = _aNlshl ([bp-6]:[bp-8], 4)
```

```
printf ("c * 5 = %d, a = %ld n",
        loc1 * 5, [bp-6]:[bp-8])
ret
```

```
arg2 = arg2 & 0x00FF
jcond (arg2 = 0) B7
```

```
arg1 = arg1 << 1
arg2 = arg2 - 1
jcond (arg2 <> 0) B6
```

```
ret arg2
```

Figure 5-3: Flow graph After Code Optimization

Definition 21 *A* **locally upwards exposed** *use u in a basic block B_i is a use which has not been previously defined in B_i.*

Definition 22 *A definition d in basic block B_i* **reaches** *basic block B_j if*[1]

1. *d is a locally available definition from B_i.*
2. *$\exists B_i \rightarrow B_j$.*
3. *$\exists B_i \rightarrow B_j \bullet \forall B_k \in (B_i \rightarrow B_j), k \neq i \wedge k \neq j, B_k$ does not redefine d.*

Definition 23 *Any definition of a register/flag in a basic block B_i is said to* **kill** *all definitions of the same register/flag that reach B_i.*

Definition 24 *A definition d in a basic block B_i is* **preserved** *if d is not redefined in B_i.*

Definition 25 *The definitions* **available** *at the exit of a basic block B_i are either:*

1. *The locally available definitions of the register/flag.*
2. *The definitions of the register/flag reaching B_i.*

Definition 26 *A use u of a register/flag is* **upwards exposed** *in a basic block B_i if either:*

1. *u is locally upwards exposed from B_i.*
2. *$\exists B_i \rightarrow B_k \bullet u$ is locally upwards exposed from $B_k \wedge \nexists B_j, i \leq j < k$, which contains a definition of u.*

[1]The symbol $\rightarrow$ is used in this Chapter to represent a path. This symbol is defined in Chapter 6, Section 6.3.1.

Definition 27 *A definition d is* **live** *or* **active** *at basic block B_i if:*

1. *d reaches B_i*

2. *There is an upwards exposed use of d at B_i.*

Definition 28 *A definition d in basic block B_i is* **busy** *(sometimes called* **very busy***) if d is used before being redefined along all paths from B_i.*

Definition 29 *A definition d in basic block B_i is* **dead** *if d is not used before being redefined along all paths from B_i (i.e. d is not busy or live).*

Definition 30 *A* **definition-use chain** *(du-chain) for a definition d at instruction i is the set of instructions j where d could be used before being redefined (i.e. the instructions which can be affected by d).*

Definition 31 *A* **use-definition chain** *(ud-chain) for a use u at instruction j is the set of instructions i where u was defined (i.e. the statements which can affect u).*

Definition 32 *A path is d-***clear** *if there is no definition of d along that path.*

5.3.2 Taxonomy of Data Flow Problems

Data flow problems are solved by a series of equations that uses information collected in each basic block, and propagates it across the complete control flow graph. Information propagated within the flow graph of a procedure is called **intraprocedural** data flow analysis, and information propagated across procedure calls is called **interprocedural** data flow analysis.

Information on registers defined or killed is collected from within the basic block in the form of sets (e.g. gen() and kill()), and is then summarized at basic block entrance and exit in the form of sets (e.g. in() and out() sets). A typical data flow equation for basic block B_i has the following form:

$$out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i))$$

and stands for "the information at the end of basic block B_i is either the information generated on B_i, or the information that entered the basic block and was not killed within the basic block". The summary in() information is collected from the predecessor nodes of the graph, by an equation of the form:

$$in(B_i) = \cup_{p \in Pred(B_i)} out(p)$$

which collects information that is available at the exit of any predecessor node. This data flow problem is classified as an any-path problem, since the information collected from predecessors is derived from any path (i.e. not all paths need to have the same information). Any-path problems are represented in equations by a union of predecessors or successors, depending on the problem.

In a similar way, an all-paths problem is a data flow problem that is specified by an equation that collects information available in all paths from the current basic block to the successors or predecessors, depending on the type of problem.

Definition 33 *A data flow problem is said to be* **forward-flow** *if*

1. *The out() set is computed in terms of the in() set within the same basic block.*
2. *The in() set is computed from the out() set of predecessor basic blocks.*

Definition 34 *A data flow problem is said to be* **backward-flow** *if*

1. *The in() set is computed in terms of the out() set within the same basic block.*
2. *The out() set is computed from the in() set of successor basic blocks.*

This classification of data flow problems derives the taxonomy shown in Figure 5-4. For each forward- and backward-flow problem, all-path and any-path equations are defined in terms of successors and predecessors. This table is taken from [FJ88b].

	Forward-Flow	Backward-Flow

Any path	Out(B_i) = Gen(B_i) $\cup$ (In(B_i) - Kill(B_i))
	In(B_i) = $\cup_{p \in \text{Pred}(B_i)}$ Out(p)
		In(B_i) = Gen(B_i) $\cup$ (Out(B_i) - Kill(B_i))
		Out(B_i) = $\cup_{s \in \text{Succ}(B_i)}$ In(s)

All paths	Out(B_i) = Gen(B_i) $\cup$ (In(B_i) - Kill(B_i))
	In(B_i) = $\cap_{p \in \text{Pred}(B_i)}$ Out(p)
		In(B_i) = Gen(B_i) $\cup$ (Out(B_i) - Kill(B_i))
		Out(B_i) = $\cap_{s \in \text{Succ}(B_i)}$ In(s)

Figure 5-4: Data Flow Analysis Equations

Data Flow Equations

Data flow equations do not, in general, have unique solutions; but in data flow problems either the minimum or maximum fixed-point solution that satisfies the equations is the one of interest. Finding this solution is done by setting a boundary condition on the initial value of the *in*(B) set of the header basic block for forward-flow problems, and the value of the *out*(B) set of the exit basic block for backward-flow problems. Depending on the interpretation of the problem, these boundary condition sets are initialized to the empty or the universal set (i.e. all possible values).

Intraprocedural data flow problems solve equations for a subroutine without taking into account the values used or defined by other subroutines. As these problems are flow insensitive, the boundary conditions are set for all initial (for forward-flow problems) or all exit (for backward-flow problems) nodes. Interprocedural data flow problems solve equations for the subroutines of a program taking into account values use or defined by invoked subroutines. Information flows between subroutines of the call graph. These flow sensitive problems set the boundary condition only for the main subroutine of the program's call graph, all other subroutines summarize information from all predecessor (in the case of forward-flow problems) or all successor (for backward-flow problems) nodes in the call graph (i.e. the caller node). This section presents data flow equations used to solve reaching, live, available, and busy registers.

The reaching register definition analysis determines which registers reach a particular basic block along some path, thus, the following forward-flow, any-path equations are used:

Definition 35 *Let*

- B_i *be a basic block*
- *ReachIn(B_i) be the set of registers that reach the entrance to* B_i
- *ReachOut(B_i) be the set of registers that reach the exit from* B_i
- *Kill(B_i) be the set of registers killed in* B_i
- *Def(B_i) be the set of registers defined in* B_i

Then

$$\begin{aligned} \mathit{ReachIn}(B_i) &= \begin{cases} \bigcup_{p \in \mathit{Pred}(B_i)} \mathit{ReachOut}(p) & \textit{if } B_i \textit{ is not the header node} \\ \emptyset & \textit{otherwise} \end{cases} \\ \mathit{ReachOut}(B_i) &= \mathit{Def}(B_i) \cup (\mathit{ReachIn}(B_i) - \mathit{Kill}(B_i)) \end{aligned}$$

Live register analysis determines whether a register is to be used along some path, thus, the following backward-flow, any-path equations are used:

Definition 36 *Let*

- B_i *be a basic block*
- *LiveIn(B_i) be the set of registers that are live on entrance to* B_i
- *LiveOut(B_i) be the set of registers that are live on exit from* B_i
- *Use(B_i) be the set of registers used in* B_i
- *Def(B_i) be the set of registers defined in* B_i

Then

$$\begin{aligned} \mathit{LiveOut}(B_i) &= \begin{cases} \bigcup_{s \in \mathit{Succ}(B_i)} \mathit{LiveIn}(s) & \textit{if } B_i \textit{ is not a return node} \\ \emptyset & \textit{otherwise} \end{cases} \\ \mathit{LiveIn}(B_i) &= \mathit{Use}(B_i) \cup (\mathit{LiveOut}(B_i) - \mathit{Def}(B_i)) \end{aligned}$$

Available register analysis determines which registers are available along all paths of the graph, thus, the following forward-flow, all-paths equations are used:

Definition 37 *Let*

- B_i *be a basic block*
- *AvailIn(B_i) be the set of the registers that are available on entrance to* B_i
- *AvailOut(B_i) be the set of the registers that are available on exit from* B_i
- *Compute(B_i) be the set of the registers in* B_i *computed and not killed*
- *Kill(B_i) be the set of the registers in* B_i *that are killed due to an assignment*

Then

$$\begin{aligned}
AvailIn(B_i) &= \begin{cases} \bigcap_{p \in Pred(B_i)} AvailOut(p) & \text{if } B_i \text{ is not the header node} \\ \emptyset & \text{otherwise} \end{cases} \\
AvailOut(B_i) &= Compute(B_i) \cup (AvailIn(B_i) - Kill(B_i))
\end{aligned}$$

Busy register analysis determines which registers are busy along all paths of the graph, thus, the following backward-flow, all-paths equations are used:

Definition 38 *Let*

- *B_i be a basic block*
- *BusyIn(B_i) be the set of the registers that are busy on entrance to B_i*
- *BusyOut(B_i) be the set of the registers that are busy on exit from B_i*
- *Use(B_i) be the set of the registers that are used before killed in B_i*
- *Kill(B_i) be the set of the registers that are killed before used in B_i*

Then

$$\begin{aligned}
BusyOut(B_i) &= \begin{cases} \bigcap_{s \in Succ(B_i)} BusyIn(s) & \text{if } B_i \text{ is not a return node} \\ \emptyset & \text{otherwise} \end{cases} \\
BusyIn(B_i) &= Use(B_i) \cup (BusyOut(B_i) - Kill(B_i))
\end{aligned}$$

The problem of finding the uses of a register definition, i.e. a du-chain problem, is solved by a backward-flow, any-path problem. Similarly, the problem of finding all definitions for a use of a register, i.e. a ud-chain problem, is solved by a forward-flow, any-path problem. The previous data flow problems are summarized in the table in Figure 5-5.

	Forward-Flow	Backward-Flow
Any-path	Reach ud-chains	Live du-chains
All-path	Available Copy propagation	Busy Dead

Figure 5-5: Data Flow Problems - Summary

Recently, precise interprocedural live variable equations were presented as part of a code optimization at link-time system [SW93]. A two-phase approach is used in order to remove information propagation across unrelated subroutines that call the same other subroutine. The call graph has two nodes for each call node; the call node as such, which has an out-edge to the header node of the callee subroutine, and the ret_call node, which has an in-edge from the return node of the callee subroutine. In the first phase, information flows across normal nodes and call edges only; return edges are removed from the call graph. In the second phase, information flows across normal nodes and return edges only; call edges are removed from the call graph. This phase makes use of the summary information calculated in the first phase. Because the information flows from the caller to the callee, and viceversa, this method provides a more precise information than other methods presented in the literature.

Definition 39 presents the equations used for precise interprocedural register analysis. Live and dead register equations are solved for the first phase, and summarized for each subroutine of the call graph in the PUse() and PDef() sets. Since live register equations are also solved in the second phase, these equations have been associated with the phase number to differentiate them (e.g. LiveIn1() for the first phase, and LiveIn2() for the second phase). Separate equations are given for call, and ret_call basic blocks. The initial boundary conditions for both live and dead equations is the empty set.

Definition 39 *Let*

- *B_i be a basic block other than call and ret_call*
- *LiveIn1(B_j) be the set of registers that are live on entrace to B_j during phase one*
- *LiveOut1(B_j) be the set of registers that are live on exit from B_j during phase one*
- *DeadIn(B_j) be the set of registers that have been killed on entrance to B_j*
- *DeadOut(B_j) be the set of registers that have been killed on exit from B_j*
- *Use(B_j) be the set of registers used in B_j*
- *Def(B_j) be the set of registers defined in B_j*
- *LiveIn2(B_j) be the set of registers that are live on entrace to B_j during phase two*
- *LiveOut2(B_j) be the set of registers that are live on exit from B_j during phase two*

Then precise interprocedural live register analysis is calculated as follows:

- *Phase 1:*

$$\begin{array}{rcl}
\mathit{LiveOut1}(B_i) & = & \begin{cases} \bigcup_{s \in Succ(B_i)} \mathit{LiveIn1}(s) & \text{if } B_i \text{ is not a return node} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{LiveIn1}(B_i) & = & \mathit{Use}(B_i) \cup (\mathit{LiveOut1}(B_i) - \mathit{Def}(B_i)) \\
\mathit{DeadOut}(B_i) & = & \begin{cases} \bigcap_{s \in Succ(B_i)} \mathit{DeadIn}(s) & \text{if } B_i \text{ is not a return node} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{DeadIn}(B_i) & = & \mathit{Def}(B_i) \cup (\mathit{DeadOut}(B_i) - \mathit{Use}(B_i)) \\
\mathit{LiveOut1}(ret\_call) & = & \cup_{s \in Succ(ret\_call)} \mathit{LiveIn1}(s) \\
\mathit{LiveIn1}(ret\_call) & = & \mathit{LiveOut1}(ret\_call) \\
\mathit{LiveOut1}(call) & = & \mathit{LiveIn1}(entry) \cup (\mathit{LiveOut1}(ret\_call) - \mathit{DeadIn}(entry)) \\
\mathit{LiveIn1}(call) & = & \mathit{LiveOut1}(call) \\
\mathit{DeadOut}(ret\_call) & = & \cap_{s \in Succ(ret\_call)} \mathit{DeadIn}(s) \\
\mathit{DeadIn}(ret\_call) & = & \mathit{DeadOut}(ret\_call) \\
\mathit{DeadOut}(call) & = & \mathit{DeadIn}(entry) \cup (\mathit{DeadOut}(ret\_call) - \mathit{LiveIn1}(entry)) \\
\mathit{DeadIn}(call) & = & \mathit{DeadOut}(call)
\end{array}$$

- *Subroutine summary:* $\forall p$ *subroutine,*

$$\begin{aligned} PUse(p) &= LiveIn1(entry) \\ PDef(p) &= DeadIn1(entry) \end{aligned}$$

- *Phase 2:*

$$\begin{aligned} LiveOut2(B_i) &= \begin{cases} \bigcup_{s \in Succ(B_i)} LiveIn2(s) & \text{if } B_i \text{ is not the return node of main} \\ \emptyset & \text{otherwise} \end{cases} \\ LiveIn2(B_i) &= Use(B_i) \cup (LiveOut2(B_i) - Def(B_i)) \\ LiveOut2(ret\_call) &= \bigcup_{s \in Succ(ret\_call)} LiveIn2(s) \\ LiveIn2(ret\_call) &= LiveOut2(ret\_call) \\ LiveOut2(call) &= PUse(p) \cup (LiveOut2(ret\_call) - PDef(p)) \\ LiveIn2(call) &= LiveOut2(call) \end{aligned}$$

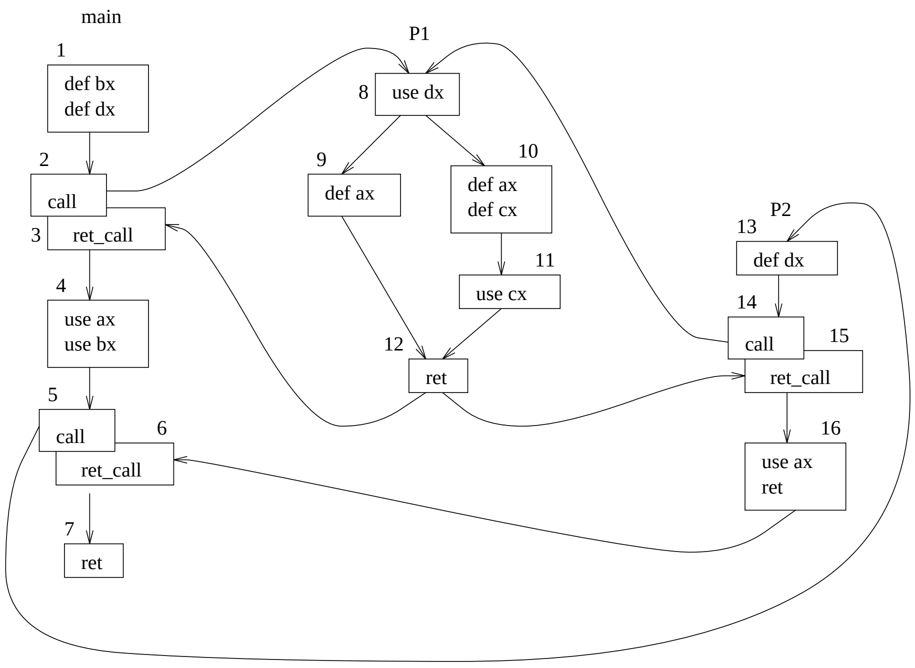


Figure 5-6: Live Register Example Graph

Example 8 *Consider the call graph of Figure 5-6. This program has a main procedure and two subroutines. Interprocedural live register analysis, as explained in Definition 39 provides the following summary information for its nodes:*

- *Phase 1:*

Subroutine	*Node*	*Def*	*Use*	*LiveIn1*	*LiveOut1*	*DeadIn*	*DeadOut*
P1	*12*	∅	∅	∅	∅	∅	∅
	11	∅	{cx}	{cx}	∅	∅	∅
	10	{ax,cx}	∅	∅	{cx}	{ax,cx}	∅
	9	{ax}	∅	∅	∅	{ax}	∅
	8	∅	{dx}	{dx}	∅	{ax}	{ax}
P2	*16*	∅	{ax}	{ax}	∅	∅	∅
	15	∅	∅	{ax}	{ax}	∅	∅
	14	∅	∅	{dx}	{dx}	{ax}	{ax}
	13	{dx}	∅	∅	{dx}	{ax,dx}	{ax}
main	*7*	∅	∅	∅	∅	∅	∅
	6	∅	∅	∅	∅	∅	∅
	5	∅	∅	∅	∅	{ax,dx}	{ax,dx}
	4	∅	{ax,bx}	{ax,bx}	∅	{dx}	{ax,dx}
	3	∅	∅	{ax,bx}	{ax,bx}	{dx}	{dx}
	2	∅	∅	{bx,dx}	{bx,dx}	{ax}	{ax}
	1	{bx,dx}	∅	∅	{bx,dx}	{ax,bx,dx}	{ax}

- *Subroutine summary:*

Subroutine	*PUse*	*PDef*
P1	{dx}	{ax}
P2	∅	{ax,dx}
main	∅	{ax,bx,dx}

- *Phase 2:*

Subroutine	*Node*	*Def*	*Use*	*LiveIn2*	*LiveOut2*
P1	*12*	∅	∅	{ax,bx}	{ax,bx}
	11	∅	{cx}	{ax,bx,cx}	{ax,bx}
	10	{ax,cx}	∅	{bx}	{ax,bx,cx}
	9	{ax}	∅	{bx}	{ax,bx}
	8	∅	{dx}	{bx,dx}	{bx}
P2	*16*	∅	{ax}	{ax}	∅
	15	∅	∅	{ax}	{ax}
	14	∅	∅	{dx}	{dx}
	13	{dx}	∅	∅	{dx}
main	*7*	∅	∅	∅	∅
	6	∅	∅	∅	∅
	5	∅	∅	∅	∅
	4	∅	{ax,bx}	{ax,bx}	∅
	3	∅	∅	{ax,bx}	{ax,bx}
	2	∅	∅	{bx,dx}	{bx,dx}
	1	{bx,dx}	∅	∅	{bx,dx}

Other types of data flow equations are also used to solve data flow problems. Consider the problem of finding all reaching register definitions to a basic block B_i according to Definition 22. In this definition, the reaching problem is defined in terms of the available problem; a register reaches a basic block if that register is available along some path from a predecessor node to the current node. This problem is equivalent to finding the set ReachIn(). The following equation is used to solve this problem:

Definition 40 *Let*

1. *B_i be a basic block*
2. *Reach(B_i) be the set of reaching registers to B_i*
3. *Avail(B_i) be the set of available registers from B_i*

Then

$$Reach(B_i) = \bigcup_{p \in Pred(B_i)} Avail(p)$$

The problem of finding available registers out of a basic block is defined in terms of locally available and reaching definitions (see Definition 25). This problem is equivalent to finding the set AvailOut(). The following equation is used:

Definition 41 *Let*

1. *B_i be a basic block*
2. *Avail(B_i) be the set of available registers from B_i*
3. *Reach(B_i) be the set of reaching registers to B_i*
4. *Propagate(B_i) be the set of the registers that are propagated across B_i*
5. *Def(B_i) be the set of locally available definitions in B_i*

Then

$$Avail(B_i) = Def(B_i) \cup (Reach(B_i) \cap Propagate(B_i))$$

Finally, Definition 27 defines the live register problem in terms of reaching definitions and upwards exposed uses. This problem is equivalent to solving the equation to the LiveIn() set. The following equation is used:

Definition 42 *Let*

1. *B_i be a basic block*
2. *Live(B_i) be the set of live registers on entrance to B_i*
3. *Reach(B_i) be the set of reaching registers to B_i*
4. *UpwardExp(B_i) be the set of the registers that are upwards exposed in B_i*

Then

$$Live(B_i) = Reach(B_i) \cap UpwardExp(B_i)$$

5.3.3 Solving Data Flow Equations

Given the control flow graph of a subroutine, data flow equations can be solved by two different methods: the iterative method, where a solution is recomputed until a fixed-point is met; and the interval method, where a solution is found for an interval and then propagated across the nodes in that interval. These equations do not have a unique solution, but the minimal solution is taken as the answer. Iterative algorithms are explained in [ASU86b], and interval algorithms are given in [All72, AC76].

5.4 Code-improving Optimizations

This section describes how data flow information is used to solve code-improving optimizations for a decompiler. The aim of these optimizations is to eliminate all references to condition codes and registers as they do not exist in high-level languages, and to regenerate the high-level expressions available in the decompiled program. This section makes references to the initial Figure 5-2, which is replicated here for convenience as Figure 5-7.

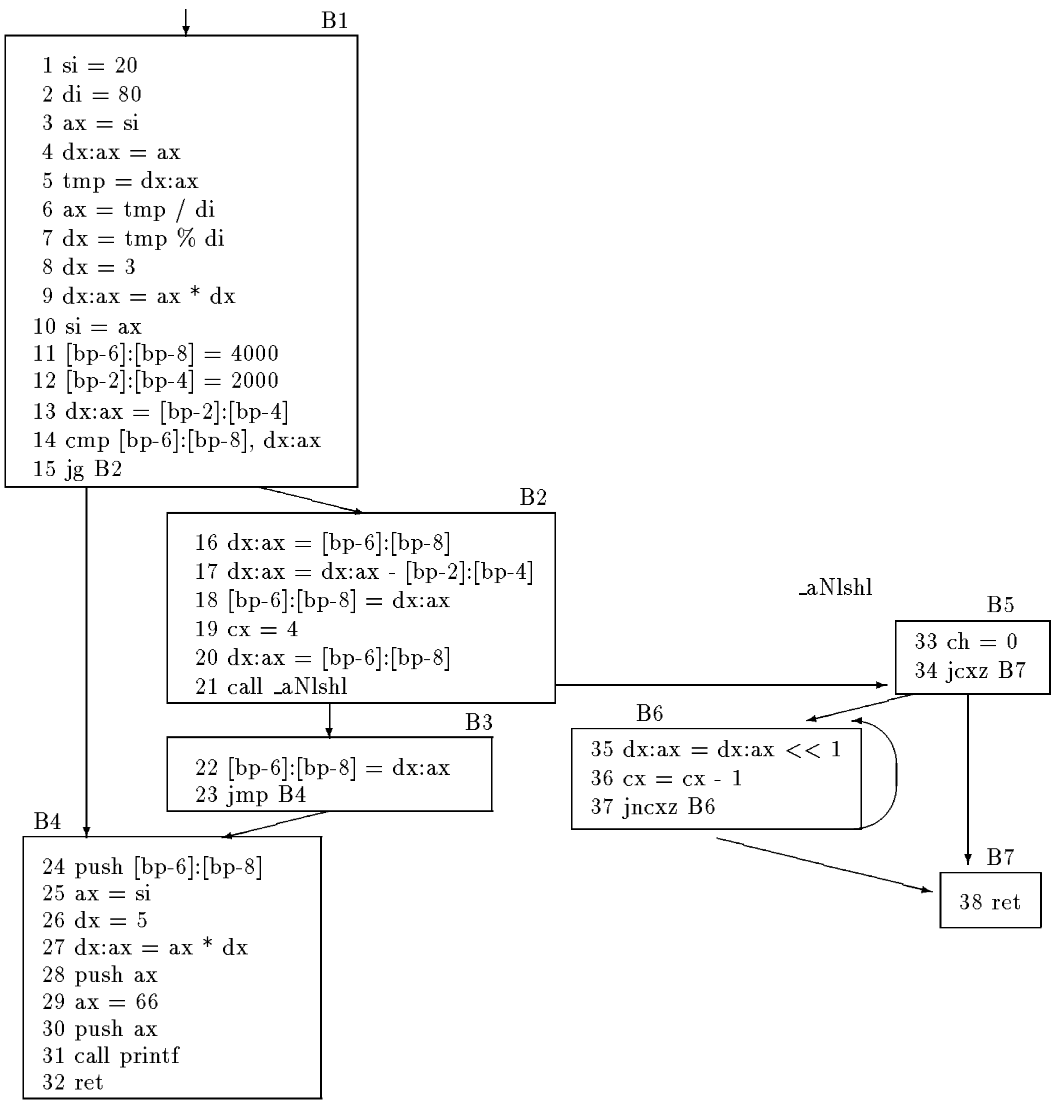
5.4.1 Dead-Register Elimination

A register is dead if it is defined by an instruction and it is not used before being redefined by a subsequent instruction. If the instruction that defines a dead register defines only this one register, it is said that the instruction is useless, and thus, is eliminated. On the other hand, if the instruction also defines other register(s), the instruction is still useful but should not define the dead register any more. In this case, the instruction is modified to reflect this fact. Dead register analysis is solved with the use of definition-use chains on registers, as the definition-use chain states which instructions use the defined register; if there are no instructions that use this register, the register is dead. Consider the following code from basic block B1, Figure 5-7 with definition-use (du) chains for all registers defined. Note that register variables do not have a du-chain as they represent local variables rather than temporary registers.

```
6  ax = tmp / di    ; du(ax) = {9}
7  dx = tmp % di    ; du(dx) = {}
8  dx = 3           ; du(dx) = {9}
9  dx:ax = ax * dx  ; du(ax) = {10}  du(dx) = {}
10 si = ax
```

From inspection, register dx at instruction 7 and 9 is defined but not subsequently used before redefinition, so it is dead in both instructions. Instruction 7 defines only this register, thus, it is redundant and can be eliminated. Instruction 9 also defines register ax, so the instruction is modified to reflect the fact that dx is not defined by the instruction any more. The resulting code looks like this:

```
6  ax = tmp / di    ; du(ax) = {9}
8  dx = 3           ; du(dx) = {9}
9  ax = ax * dx     ; du(ax) = {10}
10 si = ax
```



```
1 si = 20
2 di = 80
3 ax = si
4 dx:ax = ax
5 tmp = dx:ax
6 ax = tmp / di
7 dx = tmp % di
8 dx = 3
9 dx:ax = ax * dx
10 si = ax
11 [bp-6]:[bp-8] = 4000
12 [bp-2]:[bp-4] = 2000
13 dx:ax = [bp-2]:[bp-4]
14 cmp [bp-6]:[bp-8], dx:ax
15 jg B2
```

```
16 dx:ax = [bp-6]:[bp-8]
17 dx:ax = dx:ax - [bp-2]:[bp-4]
18 [bp-6]:[bp-8] = dx:ax
19 cx = 4
20 dx:ax = [bp-6]:[bp-8]
21 call _aNlshl
```

```
22 [bp-6]:[bp-8] = dx:ax
23 jmp B4
```

```
24 push [bp-6]:[bp-8]
25 ax = si
26 dx = 5
27 dx:ax = ax * dx
28 push ax
29 ax = 66
30 push ax
31 call printf
32 ret
```

```
33 ch = 0
34 jcxz B7
```

```
35 dx:ax = dx:ax << 1
36 cx = cx - 1
37 jncxz B6
```

```
38 ret
```

Figure 5-7: Flow Graph Before Optimization

The algorithm in Figure 5-8 finds all registers that are dead and removes them from the code.

For the purposes of decompilation optimization, du-chains are to be used again later on, so the du-chains needs to be updated to reflect the elimination of some instructions: if an instruction i is to be eliminated due to a dead register definition r defined in terms of other registers (i.e. $r = f(r_1, \ldots, r_n), n \geq 1$), the uses of these registers at instruction i no longer exist, and thus, the corresponding du-chains of the instructions that define the registers used at i are to be modified so that they no longer have a reference to i. This problem is solved by checking the use-definition chain of i, which states which instructions j define registers used in i. Consider again the piece of code from basic block B1 with du and ud (use-definition) chains on registers:

```
procedure DeadRegElim
/* Pre: du-chains on registers have been computed for all instructions.
 * Post: dead registers and instructions are eliminated */

  for (each basic block b) do
    for (each instruction i in b) do
      for (each register r defined in i) do
        if (du(r) = {}) then
           if (i defines only register r) then
              eliminate instruction i
           else
              modify instruction i not to define register r
              def(i) = def(i) - {r}
           end if
        end if
      end for
    end for
  end for
end procedure
```

Figure 5-8: Dead Register Elimination Algorithm

```
5  tmp = dx:ax     ; du(tmp) = {6,7}         ; ud(dx) = {4} ud(ax) = {4}
6  ax = tmp / di   ; du(ax) = {9}            ; ud(tmp) = {5}
7  dx = tmp % di   ; du(dx) = {}             ; ud(tmp) = {5}
8  dx = 3          ; du(dx) = {9}
9  dx:ax = ax * dx ; du(ax) = {10} du(dx)={} ; ud(ax) = {6} ud(dx) = {8}
10 si = ax         ;                         ; ud(ax) = {9}
```

When instruction 7 is detected to be redundant, its ud-chain is checked for any instruction(s) that defined the register(s) involved in the computation of the dead register `dx`. As seen, register `tmp` is used at instruction 7 and was defined in instruction 5 (ud(tmp) = {5}), which has a du-chain of instructions 6 and 7. Since instruction 7 is going to be eliminated, the du-chain of instruction 5 must be updated to reach only instruction 6, leading to the following code after dead register elimination and du-chain update:

```
5  tmp = dx:ax     ; du(tmp) = {6}  ; ud(dx) = {4}  ud(ax) = {4}
6  ax = tmp / di   ; du(ax) = {9}   ; ud(tmp) = {5}
8  dx = 3          ; du(dx) = {9}
9  ax = ax * dx    ; du(ax) = {10}  ; ud(ax) = {6}  ud(dx) = {8}
10 si = ax         ;                ; ud(ax) = {9}
```

The algorithm in Figure 5-9 solves the problem of updating du-chains while doing dead-register elimination. This algorithm should be invoked by the `deadRegElim` procedure once an instruction is detected to be redundant, and before it is removed. Note that the du-chain for a particular register might become empty, leading to further dead registers that are recursively eliminated from the code.

```
procedure UpdateDuChain (i: instructionNumber)
/* Pre: ud and du-chains on registers have been computed for all instructions.
 *      instruction i is to be eliminated.
 * Post: no du-chain references instruction i any more */

   for (each register r used in instruction i) do
     for (each instruction j in ud(r)) do
       if (i in du(r) at instruction j) then
          du(r) = du(r) - {i}
          if (du(r) = {}) then
             if (j defines only register r) then
                updateDuChain (j)
                eliminate instruction j
             else
                modify instruction j not to define register r
                def(j) = def(j) - {r}
             end if
         end if
       end if
     end for
   end for
end procedure
```

Figure 5-9: Update of du-chains

5.4.2 Dead-Condition Code Elimination

A condition code (or flag) is dead if it is defined by an instruction and is not used before redefinition. Since the definition of a condition code is a side effect of an instruction (i.e. the instruction has another function), eliminating dead-flags does not make an instruction redundant, therefore, instructions are not eliminated by dead-flag elimination. In this analysis, once a condition code has been determined to be dead, it is no longer necessary for it to be defined by an instruction, so this information is removed from the instruction. Information on condition codes is kept in an instruction in the form of sets: a set of defined conditions and a set of used conditions (i.e. bitsets). The analysis used to find which condition codes are dead is similar to dead-register analysis in that du-chains are used. In this case there is no need of ud-chains, since no instruction is eliminated. Consider the following code from basic block B1, Figure 5-7, with du-chains on condition codes:

```
14 cmp [bp-6]:[bp-8], dx:ax ; def={ZF,CF,SF} ; du(SF)={15} du(CF,ZF)={}
15 jg B2                    ; use={SF}
```

Instruction 14 defines condition codes `ZF` (zero), `CF` (carry), and `SF` (sign). Checking the du-chains of these conditions we find that only flag `SF` is used later on, thus, the other flags are not used after this definition, and are therefore dead. The definition of these flags is removed from the code associated with instruction 14, leading to the following code:

```
14  cmp [bp-6]:[bp-8], dx:ax ; def = {SF} ; du(SF)={15}
15  jg B2                    ; use = {SF}
```

The algorithm in Figure 5-10 finds all condition codes that are dead and eliminates them.

```
procedure DeadCCElim
/* Pre: du-chains on condition codes have been computed for all instructions.
 * Post: dead condition codes are eliminated */

   for (each basic block b) do
     for (each instruction i in b) do
       for (each condition code c in def(i)) do
         if (du(c) = {}) then
            def(i) = def(i) - {c}
         end if
       end for
     end for
   end for
end procedure
```

Figure 5-10: Dead Condition Code Elimination Algorithm

5.4.3 Condition Code Propagation

Dead-condition code elimination removes all definitions of condition codes that are not used in the program. All remaining condition code definitions have a use in a subsequent instruction, and are to be eliminated after capturing the essence of the condition. The problem can be solved by means of du-chains or ud-chains in condition codes; either way provides an equivalent solution. Consider the following code from basic block B1, Figure 5-7 with ud-chains on condition codes:

```
14  cmp [bp-6]:[bp-8], dx:ax  ; def = {SF}
15  jg B2                     ; use = {SF}  ; ud(SF) = {14}
```

For a particular flag(s) use, we find the instruction that defined the flag(s) and merge these two instructions according to the Boolean condition implicit in the instruction that uses the flag. Instruction 15 uses flag `SF`, and implicitly checks for a greater-than Boolean condition. Instruction 14 defines the flag used in instruction 15, and it compares the first identifier (`[bp-6]:[bp-8]`) against the second identifier (`dx:ax`). If the first identifier is greater than the second identifier, the `SF` is set. Other flags that were originally set by this instruction have been eliminated via dead-condition code elimination, so are not considered. It is obvious from the function of these two instructions that the propagation of the condition that sets the SF (i.e. comparing two identifiers) to the instruction that uses this condition will eliminate the instruction that defines the condition, and will generate a Boolean condition for the instruction that uses the condition. In our example, the propagation of the SF leads to the following code:

```
15  jcond ([bp-6]:[bp-8] > dx:ax) B2
```

thus, eliminating all flag references.

Condition Code Uses within Extended Basic Blocks

Definition 43 *An* extended basic block *is a sequence of basic blocks $B_1, \ldots, B_n$ such that for $1 \le i < n, B_i$ is the only predecessor of B_{i+1}, and for $1 < i \le n, B_i$ has only a conditional jump instruction.*

Flag definition and uses occur in the same basic block in most programs. In some standard cases, the flag definition is not within the same block of the flag use, but is within the same extended basic block, as in the following code:

```
1  cmp ax, dx     ; def = {SF,ZF}  ; du(SF) = {2}  du(ZF) = {3}
2  jg Bx          ; use = {SF}     ; ud(SF) = {1}
3  je By          ; use = {ZF}     ; ud(ZF) = {1}
```

In this case, instruction 1 defines two flags: SF and ZF. The sign flag is used by instruction 2 (within the same basic block), and the zero flag is used by instruction 3 (in a different basic block but within the same extended basic block). The sign condition from instruction 1 is propagated to instruction 2, which checks for a greater-than Boolean condition, and instruction 2 is replaced by:

```
1  cmp ax, dx             ; def = {ZF}  ; du(ZF) = {3}
2  jcond (ax > dx) Bx
3  je By                  ; use = {ZF}  ; ud(ZF) = {1}
```

Since instruction 1 also defines the zero flag, which is used at instruction 3, the instruction is not removed yet, as the identifiers that form part of the Boolean condition need to be known. Following the analysis, when instruction 3 is analyzed, the definition of the zero flag in instruction 1 is propagated to the use of this flag in instruction 3, and generates a Boolean condition that checks for the equality of the two registers. Since there are no other definitions of condition codes in instruction 1, this instruction is now safely eliminated, leading to the following code:

```
2  jcond (ax > dx) Bx
3  jcond (ax = dx) By
```

The algorithm can be extended to propagate condition codes that are defined in two or more basic blocks (i.e. by doing an and of the individual Boolean conditions), but it has not been required in practice, since it is almost unknown for even optimising compilers to attempt to track flag definitions across basic block boundaries[Gou93]. The algorithm in Figure 5-11 propagates the condition codes within an extended basic block.

The Boolean conditional expressions derived from this analysis generate expressions of the form described by the BNF in Figure 5-12. These expressions are saved as parse trees in the intermediate high-level representation.

```
procedure CondCodeProp
/* Pre: dead-condition code elimination has been performed.
 *      the sets of defined and used flags has been computed for all
 *          instructions.
 *      ud-chains on condition codes have been computed for all instructions.
 * Post: all references to condition codes have been eliminated */

   for (all basic blocks b in postorder)
     for (all instructions i in b in last to first order)
       if (use(i) <> {}) then   /* check for a flag use */
          for (all flags f in use(i)) do
            j = ud(f)
            def(j) = def(j) - {f}   /* remove it from the set */
            propagate identifiers from instruction j to the Boolean
               condition in instruction i (do not store repetitions).
            if (def(j) = {}) then
               eliminate instruction j.
            end if
          end for
       end if
     end for
   end for
end procedure
```

Figure 5-11: Condition Code Propagation Algorithm

Cond	::=	(Cond $\wedge$ RelTerm) | (Cond | RelTerm) | RelTerm
RelTerm	::=	Factor op Factor
Factor	::=	register | localVar | literal | parameter | global
op	::=	$\leq$ | $<$ | $=$ | $>$ | $\geq$ | $<>$

Figure 5-12: BNF for Conditional Expressions

5.4.4 Register Arguments

The register calling convention is used by compilers to speed up the invocation of a subroutine. It is an option available in most contemporary compilers, and is also used by the compiler runtime support routines. Given a subroutine, register arguments translate to registers that are used by the subroutine before being defined in the subroutine; i.e. upwards exposed uses of registers overall the whole subroutine. Consider the following code from basic blocks B5 and B6, Figure 5-7, subroutine `_aNlshl` after condition code elimination:

```
33  ch = 0
34  jcond (cx = 0) B7    ; ud(ch) = {33}  ud(cl) = {}
35  dx:ax = dx:ax << 1   ; ud(dx:ax) = {}
```

Instruction 34 uses register `cx`, which has not been completely defined in this subroutine: the high part, register `ch` is defined in instruction 33, but the low part is not defined at all. A similar problem is encountered in instruction 35: the registers `dx:ax` are not defined in the subroutine before being used. Information on registers used before being defined is summarized by an intraprocedural live register analysis: a register is live on entrance to the basic block that uses it. This analysis is done by solving the intraprocedural live register equations of Definition 36, or the equations for the first phase of precise interprocedural live register analysis (Definition 39). Performing live register analysis on subrotine `_aNlshl` leads to the following LiveIn and LiveOut sets:

Basic Block	LiveIn	LiveOut
B5	{dx,ax,cl}	{dx,ax}
B6	{dx,ax}	{}
B7	{}	{}

The set of LiveIn registers summarized for the header basic block B5 is the set of register arguments used by the subroutine; `dx`, `ax`, and `cl` in this example. The formal argument list of this subroutine is updated to reflect these two arguments:

```
formal_arguments(_aNlshl) = (arg1 = dx:ax, arg2 = cl)
```

It is said that the `_aNlshl` subroutine *uses* these registers. In general, any subroutine that makes use of register arguments uses those registers, thus, an invocation to one of these subroutines (i.e. a `call` instruction) is also said to use those registers, as in the following instruction:

```
21 call _aNlshl     ; use = {dx, ax, cl}
```

The algorithm in Figure 5-13 finds the set of register arguments (if any) to a subroutine.

5.4.5 Function Return Register(s)

Functions return results in registers, and there is no machine instruction that states which registers are being returned by the function. After function return, the caller uses the registers returned by the function before they are redefined (i.e. these registers are live on entrance to the basic block that follows the function call). This register information is propagated across subroutine boundaries, and is solved with a reaching and live register analysis. Consider the following code from basic blocks B2 and B3, Figure 5-7:

```
20  dx:ax = [bp-6]:[bp-8]  ; def = {dx, ax}  use = {}
21  call _aNlshl           ; def = {}        use = {dx, ax, cl}
22  [bp-6]:[bp-8] = dx:ax  ; def = {}        use = {dx, ax}
```

```
procedure FindRegArgs (s: subroutineRecord)
/* Pre: intraprocedural live register analysis has been performed on
 *        subroutine s.
 * Post: uses(s) is the set of register arguments of subroutine s. */

  if (LiveIn(headerNode(s)) <> {}) then
     uses(s) = LiveIn(headerNode(s))
  else
     uses(s) = {}
  end if
end procedure
```

Figure 5-13: Register Argument Algorithm

Instruction 22 uses registers dx:ax; these registers are defined in instruction 20, but between this definition and the use a subroutine call occurs. Since it is not known whether this subroutine is a procedure or a function, it is not safe to assume that the definition in instruction 20 is the one reaching the use in instruction 22. Summary information is needed to determine which definition reaches instruction 22. Performing an intraprocedural reaching register analysis on subroutine _aNlshl leads to the following ReachIn and ReachOut sets:

Basic Block	ReachIn	ReachOut
B5	{}	{ch}
B6	{ch}	{cx,dx,ax}
B7	{cx,dx,ax}	{cx,dx,ax}

This analysis states that the last definitions of registers cx, dx, and ax reach the end of the subroutine (i.e. ReachOut set of basic block B7). The caller subroutine uses only some of these reaching registers, thus it is necessary to determine which registers are upwards exposed in the successor basic block(s) to the subroutine invocation. This information is calculated by solving the interprocedural live register equations of Definition 36, or the second phase of precise interprocedural live register analysis (Definition 39). Since the information needs to be accurate, the live register analysis equations are solved in an optimistical way; i.e. a register is live if a use of that register is seen in a subsequent node. The following LiveIn and LiveOut sets are calculated for the example of Figure 5-7:

Basic Block	LiveIn	LiveOut
B1	{}	{}
B2	{}	{dx,ax}
B3	{dx,ax}	{}
B4	{}	{}
B5	{dx,ax,cl}	{dx,ax}
B6	{dx,ax}	{dx,ax}
B7	{dx,ax}	{dx,ax}

From the three registers that reach basic block B3, only two of these registers are used (i.e. belong to LiveIn of B3): `dx:ax`, thus, these registers are the only registers of interest once the called subroutine has been finished, and are the registers returned by the function. The condition that checks for returned registers is:

$$\text{ReachOut(B7)} \bigcap \text{LiveIn(B3)} = \{\text{dx,ax}\}$$

In general, a subroutine can have one or more return nodes, therefore, the ReachOut() set of the subroutine must have all registers that reach each single exit. The following equation summarizes the ReachOut information for a subroutine s:

$$\text{ReachOut}(s) = \cap_{B_i=\text{return}} \text{ReachOut}(B_i)$$

Once a subroutine has been determined to be a function, and the register(s) that the function returns has been determined, this information is propagated to two different places: the return instruction(s) from the function, and the instructions that `call` this function. In the former case, all return basic blocks have a `ret` instruction; and this instruction is modified to return the registers that the function returns. In our example, instruction 38 of basic block B7, Figure 5-7 is modified to the following code:

```
38  ret dx:ax
```

In the latter case, any function invocation instruction (i.e. `call` instruction) is replaced by an `asgn` instruction that takes as left-hand side the defined register(s), and takes the function call as the right-hand side of the instruction, as in the following code:

```
21  dx:ax = call _aNlshl   ; def = {dx,ax}   use = {dx, ax, cl}
```

The instruction is transformed into an `asgn` instruction, and defines the registers on the left-hand side (lhs).

The algorithm in Figure 5-14 determines which subroutines are functions (i.e. return a value in a register(s)). It is important to note that in the case of library functions whose return register(s) is not used, the call is not transformed into an `asgn` instruction but remains as a `call` instruction.

5.4.6 Register Copy Propagation

Register copy propagation is the method by which a defined register in an assignment instruction, say `ax = cx`, is replaced in a subsequent instruction(s) that references or uses this register, if neither register is modified (i.e. redefined) after the assignment (i.e. neither `ax` nor `cx` is modified). If this is the case, references to register `ax` are replaced by references to register `cx`, and, if all uses of `ax` are replaced by `cx` then `ax` becomes dead and the assignment instruction is eliminated. A use of `ax` can be replaced with a use of `cx` if `ax = cx` is the only definition of `ax` that reaches the use of `ax` and if no assignments to `cx` have occurred after the instruction `ax = cx`. The former condition is checked with ud-chains on registers. The latter condition is checked with an r-clear condition (i.e. a forward-flow, all-paths analysis). Consider the following code from basic block B2, Figure 5-7 with ud-chains and du-chains:

```
procedure FindRetRegs
/* Pre: interprocedural live register analysis has been performed.
 *     intraprocedural reaching register definition has been performed.
 * Post: def(f) is the set of registers returned by a function f.
 *       call instruction to functions are modified to asgn instructions.
 *       ret instructions of functions return the function return registers.*/

   for (all subroutines s) do
     for (all basic blocks b in postorder) do
       for (all instructions i in b) do
         if (i is a call instruction to subroutine f) then
            if (function(f) == False) then /* f is not a function so far */
               def(i) = LiveIn(succ(b)) intersect ReachOut(f)
               if (def(i) <> {}) then      /* it is a function */
                  def(f) = def(i)
                  function(f) = True
                  rhs(i) = i               /* convert i into an asgn inst */
                  lhs(i) = def(f)
                  opcode(i) = asgn
                  for (all ret instructions j of function f) do
                    exp(j) = def(f)        /* propagate return register(s) */
                  end for
               end if
            else                           /* f is a function */
               rhs(i) = i                  /* convert i into an asgn inst */
               lhs(i) = def(f)
               opcode(i) = asgn
               def(i) = def(f)             /* registers defined by i */
            end if
         end if
       end for
     end for
   end for
end procedure
```

Figure 5-14: Function Return Register(s)

```
16  dx:ax = [bp-6]:[bp-8]          ;                    du(dx:ax) = {17}
17  dx:ax = dx:ax - [bp-2]:[bp-4]  ; ud(dx:ax) = {16}  du(dx:ax) = {18}
18 [bp-6]:[bp-8] = dx:ax           ; ud(dx:ax) = {17}
```

Following the ud-chains of these instructions, instruction 17 uses registers dx:ax, which were defined in instruction 16. Since these registers have not been redefined between instructions 16 and 17, the right-hand side of the instruction is replaced in the use of the registers as follows:

```
17  dx:ax = [bp-6]:[bp-8] - [bp-2]:[bp-4] ; du(dx:ax)={18}
```

Since there is only one use of these registers at instruction 16 (i.e. du(dx:ax) = 17), the registers are now dead and thus, the instruction is eliminated. In a similar way, instruction 18 uses registers `dx:ax`, which are defined in instruction 17. Since these registers have not been redefined between those two instructions, the right-hand side of instruction 17 is replaced into the use of the registers in instruction 18, leading to:

```
18  [bp-6]:[bp-8] = [bp-6]:[bp-8] - [bp-2]:[bp-4]
```

Since there was only one use of the registers definition at instruction 17, these registers become dead and the instruction is eliminated. As noticed in this example, the right-hand side of an instruction *i* can be replaced into a further use of the left-hand side of instruction *i*, building expressions on the right-hand side of an assignment instruction.

Consider another example from basic block B1, Figure 5-7, after dead-register elimination, and with ud-chains and du-chains on registers (excluding register variables):

```
3   ax = si        ;                             ; du(ax) = {4}
4   dx:ax = ax     ; ud(ax) = {3}                ; du(dx:ax) = {5}
5   tmp = dx:ax    ; ud(dx:ax) = {4}             ; du(tmp) = {6}
6   ax = tmp / di  ; ud(tmp) = {5}               ; du(ax) = {9}
8   dx = 3         ;                             ; du(dx) = {8}
9   ax = ax * dx   ; ud(ax) = {6} ud(dx) = {8}   ; du(ax) = {10}
10  si = ax        ; ud(ax) = {9}
```

The use of register `ax` in instruction 4 is replaced with a use of the register variable `si`, making the definition of `ax` in 3 dead. The use of `dx:ax` in instruction 5 is replaced with a use of `si` (from instruction 4), making the definition of `dx:ax` dead. The use of `tmp` in instruction 6 is replaced with a use of `si` (from instruction 5), making the definition of `tmp` dead at 5. The use of `ax` at instruction 9 is replaced with a use of `(si / di)` from instruction 6, making the definition of `ax` dead. In the same instruction, the use of `dx` is replaced with a use of constant 3 from instruction 8, making the definition of `dx` at 8 dead. Finally, the use of `ax` at instruction 10 is replaced with a use of `(si / di) * 3` from instruction 9, making the definition of `ax` at 9 dead. Since the register(s) defined in instructions 3 $\rightarrow$ 9 were used only once, and all these registers became dead, the instructions are eliminated, leading to the final code:

```
10 si = (si / di) * 3
```

When propagating registers across assignment instructions, a register is bound to be defined in terms of an expression of other registers, local variables, arguments, and constants. Since any of these identifiers (besides constants) can be redefined, it is necessary to check that none of these identifiers is redefined across the path from the instruction that defines the register to the instruction that uses it. Thus, the following necessary conditions are checked for register copy propagation:

1. Uniqueness of register definition for a register use: registers that are used before being redefined translate to temporary registers that hold an intermediate result for the machine. This condition is checked by means of ud-chains on registers used in an instruction.

2. rhs-clear path: the identifiers x in an expression that defines a register r (i.e. the rhs of the instruction) that satisfies condition 1 are checked to have an x-clear path to the instruction that uses the register r. The rhs-clear condition for an instruction j that uses a register r which is uniquely defined at instruction i is formally defined as:

$$\text{rhs-clear}_{i \to j} = \bigcap_{x \in \text{rhs}(i)} x\text{-clear}_{i \to j}$$

$$\begin{array}{rll} \text{where} & \text{rhs}(i) & = \text{the right hand side of instruction } i \\ \text{and} & x & = \text{an identifier that belong to the rhs}(i) \\ \text{and} & x\text{-clear}_{i \to j} & = \begin{cases} \text{True} & \text{if there is no definition of } x \text{ along the path } i \to j \\ \text{False} & \text{otherwise} \end{cases} \end{array}$$

The algorithm in Figure 5-15 performs register copy propagation on assignment instructions. For this analysis, registers that can be used as both word and byte registers (e.g. ax, ah, al) are treated as different registers in the live register analysis. Whenever register ax is defined, it also defines registers ah and al, but, if register al is defined, it defines only registers al and ax, but not register ah. This is needed so that uses of part of a register (e.g. high or low part) can be detected and treated as a byte operand rather than an integer operand.

Extension to Non-Assignment Register Usage Instructions

The algorithm given in Figure 5-15 is general enough to propagate registers that are used in instructions other than assignments, such as push, call, and jcond instructions. Consider the following code from basic block B1, Figure 5-7 after condition code propagation:

```
13  dx:ax = [bp-2]:[bp-4]             ; du(dx:ax) = {15}
15  jcond ([bp-6]:[bp-8] > dx:ax) B2  ; ud(dx:ax) = {13}
```

Instruction 15 uses registers dx:ax, which are uniquely defined in instruction 13. The rhs of instruction 13 is propagated to the use of these registers, leading to the elimination of instruction 13. The final code looks as follows:

```
15  jcond ([bp-6]:[bp-8] > [bp-2]:[bp-4]) B2
```

In a similar way, a use of a register in a push instruction is replaced by a use of the rhs of the instruction that defines the register, as in the following code from basic block B4, Figure 5-7 after dead-register elimination:

```
25  ax = si       ;                 du(ax) = {27}
26  dx = 5        ;                 du(dx) = {27}
27  ax = ax * dx  ; ud(dx) = {26}  du(ax) = {28}
28  push ax       ; ud(ax) = {27}
```

Applying the register copy propagation algorithm we arrive at the following code:

```
28  push (si * 5)
```

and instruction 25, 26, and 27 are eliminated.

A call instruction that has been modified into an asgn instruction due to a function being invoked rather than a procedure is also a candidate for register copy propagation. Consider the following code after function return register determination:

```
procedure RegCopyProp
/* Pre: dead-register elimination has been performed.
 *      ud-chains and du-chains have been computed for all instructions.
 * Post: most references to registers have been eliminated.
 *       high-level language expression have been found.  */

  for (all basic blocks b in postorder) do
    for (all instructions j in basic block b) do
      for (all registers r used by instruction j) do
        if (ud(r) = {i}) then   /* r is uniquely defined at instruction i */
           prop = True
           for (all identifiers x in rhs(i)) do  /* compute rhs-clear */
             if (not x-clear(i, j)) then
                prop = False
             end if
           end for
           if (prop == True) then                /* propagate rhs(i) */
              replace the use of r in instruction j with rhs(i)
              du(r) = du(r) - {j}   /* at instruction i */
              if (du(r) = {}) then
                 if (i defines only register r) then
                    eliminate i
                 else
                    modify instruction i not to define register r
                    def(i) = def(i) - {r}
                 end if
              end if
           end if  /* end propagate */
        end if
      end for
    end for
  end for
end procedure
```

Figure 5-15: Register Copy Propagation Algorithm

```
21  dx:ax = call _aNlshl   ; ud(dx:ax) = {20}  ud(cl) = {19}
                           ; du(dx:ax) = {22}
22  [bp-6]:[bp-8] = dx:ax  ; ud(dx:ax) = {21}
```

The function `_aNlshl` returns a value in registers `dx:ax`. These registers are used in the first instruction of the basic block that follows the current one, and are copied to the final local long variable at offset `-6`. Performing copy propagation leads to the following code:

```
22  [bp-6]:[bp-8] = call _aNlshl
```

eliminating instruction 21 as `dx:ax` become dead.

5.4.7 Actual Parameters

Actual parameters to a subroutine are normally pushed on the stack before invocation to the subroutine. Since nested subroutine calls are allowed in most languages, the arguments pushed on the stack represent those arguments of two or more subroutines, thus, it is necessary to determine which arguments belong to which subroutine. To do this, an expression stack is used, which stores the expressions associated with `push` instructions. Whenever a `call` instruction is met, the necessary number of arguments are popped from the stack. Consider the following code from basic block B4, Figure 5-7 after dead-register elimination and register copy propagation:

```
24  push [bp-6]:[bp-8]
28  push (si * 5)
30  push 66
31  call printf
```

Instructions 24, 28, and 30 `push` the expressions associated with each instruction into a stack, as shown in Figure 5-16. When the call to `printf` is reached, information on this function is checked to determine how many bytes of arguments the function call takes; in this case it takes 8 bytes. Expressions from the stack are then popped, checking the type of the expressions to determine how many bytes are used by each. The first expression is an integer constant which takes 2 bytes, the second expression is an integer expression which takes 2 bytes, and the third expression is a long variable which takes 4 bytes; for a total of 8 bytes needed by this function call. The expressions are popped from the stack and placed on the actual parameter list of the invoked subroutine according to the calling convention used by the subroutine. In our example, the library function `printf` uses C calling convention, leading to the following code:

```
31  call printf (66, si * 5, [bp-6]:[bp-8])
```

Instructions 24, 28, and 30 are eliminated from the intermediate code when they are placed on the stack.

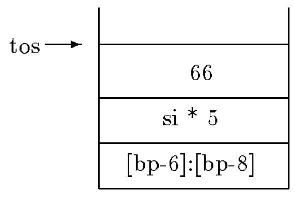


Figure 5-16: Expression Stack

Register arguments are not `push`ed on the stack, but have been defined in the use set of the subroutine that uses them. In this case, placing the actual arguments to a subroutine in the actual argument list is an extension of the register copy propagation algorithm. Consider the following code from basic blocks B2 and B3, Figure 5-7 after dead register elimination, and register argument detection:

```
19  cl = 4                  ; du(cl) = {21}
20  dx:ax = [bp-6]:[bp-8]   ; du(dx:ax) = {21}
21  dx:ax = call _aNlshl    ; ud(dx:ax) = {20}  ud(cl) = {19}
```

Instruction 21 uses registers `dx:ax`, defined in instruction 20, and register `cl`, defined in instruction 19. These uses are replaced with uses of the rhs of the corresponding instructions, and placed on the actual argument list of `_aNlshl` in the order defined by the formal argument list, leading to the following code:

```
21  dx:ax = call _aNlshl ([bp-6]:[bp-8], 4)
```

Instruction 19 and 20 are eliminated since they now define dead registers.

5.4.8 Data Type Propagation Across Procedure Calls

During the instantiation of actual arguments to formal arguments, data types for these arguments needs to be verified, as if they are different, one of the data types needs to be modified. Consider the following code from basic block B4, Figure 5-7 after all previous optimizations:

```
31  call printf (66, si * 5, [bp-6]:[bp-8])
```

where the actual argument list has the following data types: integer constant, integer, and long variable. The formal argument list of `printf` has a pointer to a character string as the first argument, and a variable number of unknown data type arguments following it. Since there is information on the first argument only, the first actual argument is checked, and it is found that it has a different data type. Given that the data types used by the library subroutines must be right (i.e. they are trusted), it is safe to say that the actual integer constant must be an offset into memory, pointing to a character string. By checking memory, it is found that at location `DS:0066` there is a string; thus, the integer constant is replaced by the string itself. The next two arguments have unknown formal type, so the type given by the caller is trusted, leading to the following code:

```
31  call printf ("c * 5 = %d, a = %ld\n", si * 5, [bp-6]:[bp-8])
```

Other cases of type propagation include the conversion of two integers into one long variable (i.e. the callee has determined that one of the arguments is a long variable, but the caller has so far used the actual argument as two separate integers).

5.4.9 Register Variable Elimination

Register variables translate to local variables in a high-level language program. These registers are replaced by new local variable names. This name replacement can be done during data flow analysis, or by the code generator. In our example, if registers `si` and `di` are replaced by the local names `loc1` and `loc2`, the following code fragment will be derived for part of basic block B1, Figure 5-7:

```
1  loc1 = 20
2  loc2 = 80
9  loc1 = (loc1 / loc2) * 3
```

5.4.10 An Extended Register Copy Propagation Algorithm

The optimizations of register copy propagation, actual parameter detection, and data type propagation across procedure calls can be performed during the one pass that propagates register information to other instructions, including arguments. Figure 5-17 lists the different high-level instructions that define and use registers. Only 3 instructions can define registers: an `asgn`, which is eliminated via register copy propagation as explained in Section 5.2.6, a function `call`, which is translated into an equivalent `asgn` instruction and eliminated by the register copy propagation method, and a `pop` instruction, which has not been addressed yet.

Define	Use
asgn (lhs)	asgn (rhs)
call (function)	call (register arguments)
pop	jcond
	ret (function return registers)
	push

Figure 5-17: Potential High-Level Instructions that Define and Use Registers

A `pop` instruction defines the associated register with whatever value is found on the top of stack. Given that `pop` instructions used to restore the stack after a subroutine call, or during subroutine return have already been eliminated from the intermediate code during idiom analysis (see Chapter 4, Sections 4.2.1 and 4.2.1), the only remaining use of a `pop` instruction is to get the last value pushed onto the stack by a previous `push` instruction (i.e. a spilled value). Since expressions associated with `push` instructions were being pushed onto an expression stack for the detection of actual arguments (see Section 5.4.7), whenever a `pop` instruction is reached, the expression on the top of stack is associated with the register of the `pop` instruction, converting the instruction into an `asgn` instruction. Consider the following code from a matrix addition procedure that spills the partially computed answer onto the stack at instructions 27 and 38, after dead-register elimination. In this example, three arrays have been passed as arguments to the procedure: the arrays pointed to by `bp+4` and `bp+6` are the two array operands, and the array pointed to by `bp+8` is the resultant array. The three arrays are arrays of integers (i.e. 2 bytes):

```
18  ax = si          ; ud(ax) = {20}
19  dx = 14h         ; ud(dx) = {20}
20  ax = ax * dx     ; ud(ax) = {21}
21  bx = ax          ; ud(bx) = {22}
22  bx = bx + [bp+4] ; ud(bx) = {25}
23  ax = di          ; ud(ax) = {24}
24  ax = ax << 1     ; ud(ax) = {25}
25  bx = bx + ax     ; ud(bx) = {26}
26  ax = [bx]        ; ud(ax) = {27}
27  push ax          ; spill ax
28  ax = si          ; ud(ax) = {30}
29  dx = 14h         ; ud(dx) = {30}
```

```
30  ax = ax * dx      ; ud(ax) = {31}
31  bx =  ax         ; ud(bx) = {32}
32  bx = bx + [bp+6] ; ud(bx) = {35}
33  ax = di          ; ud(ax) = {34}
34  ax = ax << 1     ; ud(ax) = {35}
35  bx = bx + ax     ; ud(bx) = {37}
36  pop  ax          ; ud(ax) = {37}
37  ax = ax + [bx]   ; ud(ax) = {38}
38  push ax          ; spill ax
39  ax = si          ; ud(ax) = {41}
40  dx = 14h         ; ud(dx) = {41}
41  ax = ax * dx     ; ud(ax) = {42}
42  bx = ax          ; ud(bx) = {43}
43  bx = bx + [bp+8] ; ud(bx) = {46}
44  ax = di          ; ud(ax) = {45}
45  ax = ax << 1     ; ud(ax) = {46}
46  bx = bx + ax     ; ud(bx) = {48}
47  pop  ax          ; ud(ax) = {48}
48  [bx] = ax
```

After register copy propagation on instructions 18 $\rightarrow$ 27, instruction 27 holds the contents of the array pointed to by `bp+4` offset by `si` and `di` (row and column offsets), represented by the following expression:

```
27  push [(si*20) + [bp+4] + (di*2)]
```

this expression is `push`ed on the stack, and register `ax` is redefined in the next instruction. Following extended register copy propagation, instruction 36 `pop`s the expression on the stack, and is modified to the following `asgn` instruction:

```
36  ax = [(si*20) + [bp+4] + (di*2)]   ; ud(ax) = {37}
```

this instruction is replaced into instruction 37, and register `ax` is spilled at instruction 38 holding the addition of the contents of the two arrays at offsets `si` and `di`, represented by the following expression:

```
38  push [(si*20) + [bp+4] + (di*2)] + [(si*20) + [bp+6] + (di*2)]
```

Finally, this expression is `pop`ped in instruction 47, replacing the `pop` by the following `asgn` instruction:

```
47  ax = [(si*20) + [bp+4] + (di*2)] + [(si*20) + [bp+6] + (di*2)]
```

and register `bx` holds the offset into the result array at offsets `si` and `di`. The registers in instruction 48 are replaced by the expressions calculated in instructions 46 and 47, leading to the following code:

```
48  [(si*20) + [bp+8] + (di*2)] = [(si*20) + [bp+4] + (di*2)] +
                                  [(si*20) + [bp+6] + (di*2)]
```

Note that this instruction does not *define* any registers, only uses them, therefore, this instruction is final in the sense that it cannot be replaced into any subsequent instruction. As seen, the rhs and lhs hold expressions that calculate an address of an array. These expressions can be further analyzed to determine that they calculate an array offset, and thus, the arguments passed to this subroutine are pointers to arrays; this information can then be propagated to the caller subroutine.

Figure 5-18 is a description of the final algorithm used for extended register copy propagation.

5.5 Further Data Type Propagation

Further data type determination can be done once all program expressions have been found, since data types such as arrays use address computation to reference an object in the array. This address computation is represented by an expression that needs to be simplified in order to arrive to a high-level language expression. Consider the array expression of Section 5.4.10:

```
48  [(si*20) + [bp+8] + (di*2)] = [(si*20) + [bp+4] + (di*2)] +
                                  [(si*20) + [bp+6] + (di*2)]
```

A heuristic method can be used to determine that the integer pointer at `bp+8` is a 2-dimensional array given that 2 offset expressions are used to compute an address. The offset `di*2` is adjusting the index `di` by the size of the array element type (2 in this case for an integer), and the offset `si*20` is adjusting the index `si` by the size of the row times the size of the array element (i.e. 20 / 2 = 10 elements in a row, or the number of columns in the array); therefore, the expression could be modified to the following code:

```
48  [bp+8][si][di] = [bp+4][si][di] + [bp+6][si][di]
```

and the type of the arguments are modified to array (i.e. a pointer to an integer array). In order to determine the bounds of the array, more heuristic intervention is needed. The number of elements in the one row was determined by the previous heuristic, the number of rows can be determined if the array is within a loop or any other structure that gives information regarding the number of rows. Consider the matrix addition subroutine in Figure 5-19.

This subroutine has two loops, one for the rows and one for the columns. By checking all conditional jumps for references to index `si`, the upper bound on the number of rows can be determined. In basic block B2, `si` is compared against `5`; if `si` is greater or equal to `5`, the loop is not executed (i.e. the array is not indexed into); therefore, we can assume that this is the upper bound on rows. The number of columns can also be checked by finding conditional jump instructions that use register `di`. In this case, basic block B5 compares this register against `10`; if the register is greater or equal to this constant, the inner loop is not executed (i.e. the array is not indexed into). Therefore, this constant can be used as the upper bound for the number of columns. Note that this number is the same as the one that was already known from the heuristics in determining an array address computation, therefore, we assume the number is right. This leads to the following formal argument declaration:

```
procedure ExtRegCopyProp (p: subroutineRecord)
/* Pre: dead-register analysis has been performed.
 *      dead-condition code analysis has been performed.
 *      register arguments have been detected.
 *      function return registers have been detected.
 * Post: temporary registers are removed from the intermediate code. */

 initExpStk().
 for (all basic blocks b of subroutine p in postorder) do
   for (all instructions j in b) do
     for (all registers r used by instruction j) do
       if (ud(r) = {i}) then  /* uniquely defined at instruction i */
          case (opcode(i))
            asgn: if (rhsClear (i, j))
                       case (opcode(j))
                         asgn:  propagate (r, rhs(i), rhs(j)).
                         jcond, push, ret:  propagate (r, rhs(i), exp(j)).
                         call:  newRegArg (r, actArgList(j)).
                       end case
                  end if
            pop:  exp = popExpStk().
                  case (opcode(j))
                    asgn:  propagate (r, exp, rhs(j)).
                    jcond, push, ret: propagate (r, exp, exp(j)).
                    call:  newRegArg (exp, actArgList(j)).
                  end case
            call: case (opcode(j))
                    asgn:  rhs(j) = i.
                    push, ret, jcond:  exp(j) = i.
                    call:  newRegArg (i, actArgList(j)).
                  end case
          end case
       end if
     end for

     if (opcode(i) == push) then
        pushExpStk (exp(i)).
     elsif (opcode(i) == call) and (invoked routine uses stack arguments) then
        pop arguments from the stack.
        place arguments on actual argument list.
        propagate argument type.
     end if
   end for
 end for
end procedure
```

Figure 5-18: Extended Register Copy Propagation Algorithm

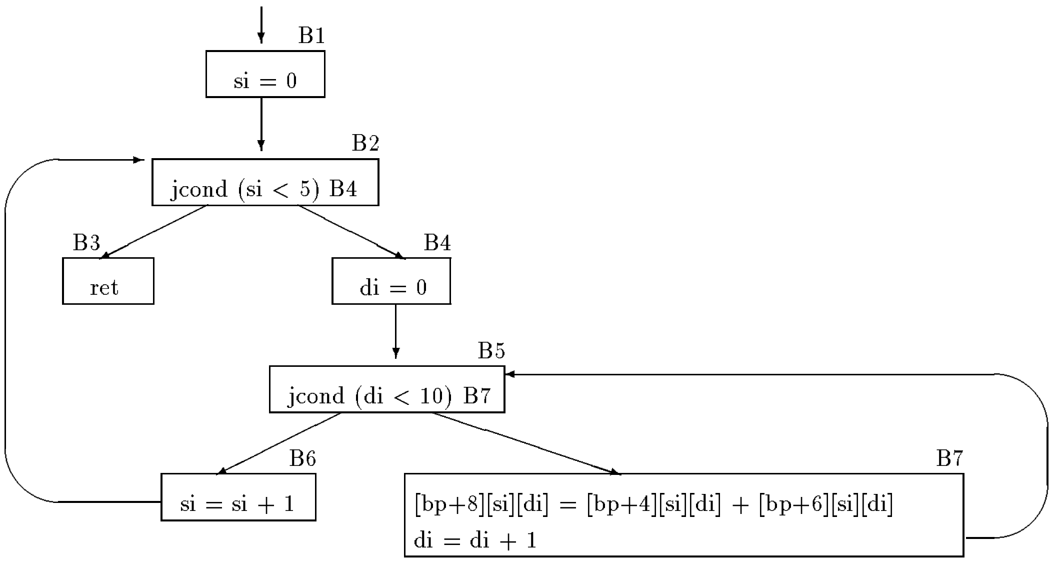


Figure 5-19: Matrix Addition Subroutine

```
formal_arguments (arg1: array[5][10] = [bp+4],
                  arg2: array[5][10] = [bp+6],
                  arg3: array[5][10] = [bp+8])
```

and the information is propagated to the caller subroutine.

It is in general hard to determine the bounds of an array if the code was optimised. For example, if strength reduction had been applied to the subscript calculation, or code motion had moved part of the subscript calculation out of the loop, or if induction variable elimination had replaced the loop indexes, then the previous heuristic method could not be applied. In this case, the decompiler would either leave the bounds of the array unknown, or ask the user for a solution via an interactive session.

Chapter 6

Control Flow Analysis

The control flow graph constructed by the front-end has no information on high-level language control structures, such as `if..then..else`s and `while()` loops. Such a graph can be converted into a structured high-level language graph by means of a *structuring* algorithm. High-level control structures are detected in the graph, and subgraphs of control structures are tagged in the graph. The relation of this phase with the data flow analysis phase and the back-end is shown in Figure 6-1.

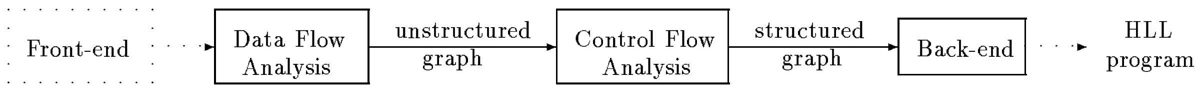


Figure 6-1: Context of the Control Flow Analysis Phase

A generic set of high-level control structures is used to structure the graph. This set should be general enough to cater for different control structures available in commonly used languages such as C, Pascal, Modula-2, and Fortran. Such structures should include different types of loops and conditionals. Since the underlying structure of the graph is not modified, functional and semantical equivalence is preserved by this method.

6.1 Previous Work

Most structuring algorithms have concentrated on the removal of `goto` statements from control flow graphs at the expense of introduction of new Boolean variables, code replication, the use of multilevel exit loops, or the use of a set of high-level structures not available in commonly used languages. A graph transformation system has also been presented, it aims at the recognition of the underlying control structures without the removal of all `goto` statements. The following sections summarize the work done in this area.

6.1.1 Introduction of Boolean Variables

Böhm and Jacopini[BJ66] proved that any program flowgraph can be represented by another flowgraph which is decomposable into π (sequence of nodes), ϕ (post-tested loop), and $\triangle$ (2-way conditional node) with the introduction of new Boolean variables and assignments to these variables. Cooper[Coo67] pointed out that if new variables may be introduced to the original program, any program can be represented in one node with at most one ϕ; therefore, from a practical point of view, the theorem is meaningless[Knu74].

Ashcroft and Manna[AM71] demonstrated that `goto` programs cannot be converted into `while()` programs without the introduction of new variables, and presented an algorithm for the conversion of these programs with the introduction of new Boolean variables. The conversion preserves the topology of the original flowchart program, but performs computations in different order.

Williams and Ossher[WO78] presented an iterative algorithm to convert a multiexit loop into a single exit loop, with the introduction of one Boolean variable and a counter integer variable for each loop.

Baker and Zweben[BZ80] reported on the structuring of multiexit loops with the introduction of new Boolean variables. The structuring of multiple exit loops is considered a control flow complexity issue, and is measured in this paper.

Williams and Chen[WG85] presented transformations to eliminate `goto` statements from Pascal programs. `Goto`s were classified according to the positioning of the target label: at the same level as the corresponding label, branch out of a structure, transferral of a label out of a structure, and abnormal exits from subroutines. All these transformations required the introduction of one or more Boolean variables, along with the necessary assignment and test statements to check on the value of a Boolean. The algorithm was implemented in Prolog on a PDP11/34.

Erosa and Hendren[EH93] present an algorithm to remove all `goto` statements from C programs. The method makes use of `goto`-elimination and `goto`-movement transformations, and introduces one new Boolean variable per `goto`. On average, three new instructions are introduced to test for each new Boolean, and different loop and if conditionals are modified to include the new Boolean. This method was implemented as part of the McCAT parallelizing decompiler.

The introduction of new (Boolean) variables modifies the semantics of the underlying program, as these variables do not form part of the original program. The resultant program is functionally equivalent to the original program, thus it produces the same results.

6.1.2 Code Replication

Knuth and Floyd[KF71] presented different methods to avoid the use of `goto` statements without the introductions of new variables. Four methods were given: the introduction of recursion, the introduction of new procedures, node splitting, and the use of a `repeat..until()` construct. The use of the node splitting technique replicates code in the final program. It is also proved that there exist programs whose `goto` statements cannot be eliminated without the introduction of new procedure calls.

Williams[Wil77] presents five subgraphs which lead to unstructured graphs: abnormal selection paths, multiple exit loops, multiple entry loops, overlapping loops, and parallel loops. In order to transform these subgraphs into structured graphs, code duplication is performed.

Williams and Ossher[WO78] presented an algorithm to replace multiple entry loops by single entry `while()` loop. The method made use of code duplication of all nodes that could be reached from abnormal entries into the loop.

Baker and Zweben[BZ80] reported on the use of the node splitting technique to generate executionally equivalent flowgraphs by replicating one or more nodes of the graph. Node splitting was considered a control flow complexity issue, and was measured.

Oulsnam[Oul82] presented transformations to convert six types of unstructured graphs to structured equivalent graphs. The methodology made use of node duplication, but no function duplication. It was demonstrated that the time overhead produced by the duplication of nodes was an increased time factor of 3 for at least one path.

Code replication modifies the original program/graph by replicating code/node one or more times, therefore, the final program/graph is functionally equivalent to the original program/graph, but its semantics and structure have been modified.

6.1.3 Multilevel Exit Loops and Other Structures

Baker[Bak77] presented an algorithm to structure flowgraphs into equivalent flowgraphs that made use of the following control structures: `if..then..else`, multilevel `break`, multilevel `next`, and endless loops. `Goto`s were used whenever the graph could not be structured using the previous structures. The algorithm was extended to irreducible graphs as well. It was demonstrated that the algorithm generated well-formed and properly nested programs, and that any `goto` statements in the final graph jumped forward. This algorithm was implemented in the `struct` program on a PDP11/54 running under Unix. It was used to rewrite Fortran programs into Ratfor, an extended Fortran language that made use of control structures. The `struct` program was later used by J.Reuter in the `decomp` decompiler to structure graphs built from object files with symbol information.

Sharir[Sha80] presented an algorithm to find the underlying control structures in a flow graph. This algorithm detected normal conditional and looping constructs, but also detected proper and improper strongly-connected intervals, and proper and improper outermost intervals. The final flow graph was represented by a hierarchical flow structure.

Ramshaw[Ram88] presented a method to eliminate all `goto` statements from programs, by means of forward and backward elimination rules. The resultant program was a structurally equivalent program that made use of multilevel exits from endless-type, named loops. This algorithm was used to port the Pascal version of Knuth's $\TeX$ compiler into the PARC/CSL, which uses Mesa. Both these languages allow the use of `goto` statements, but outward `goto`s are not allowed in Mesa.

The use of multilevel exits or high-level constructs not available in most languages restricts the generality of the structuring method and the number of languages in which the structured version of the program can be written. Currently, most 3rd generation languages (e.g. Pascal, Modula-2, C) do not make use of multilevel exits; only Ada allows them.

6.1.4 Graph Transformation System

Lichtblau[Lic85] presented a series of transformation rules to transform a control flow graph into a trivial graph by identifying subgraphs that represent high-level control structures; such as 2-way conditionals, sequence, loops, and multiexit loops. Whenever no rules were applicable to the graph, an edge was removed from the graph and a `goto` was generated in its place. This transformation system was proved to be finite Church-Rosser, thus the transformations could be applied in any order and the same final answer is reached.

Lichtblau formalized the transformation system by introducing context-free flowgraph grammars, which are context-free grammars defined by production rules that transform one graph into another[Lic91]. He proved that given a rooted context-free flowgraph grammar GG, it is possible to determine whether a flowgraph g can be derived from GG. He provided an algorithm to solve this problem in polynomial time complexity.

The detection of control structures by means of graph transformations does not modify the semantics or functionality of the underlying program, thus a transformation system provides a method to generate a semantically equivalent graph. Lichtblau's method uses a series of graph transformations on the graph to convert/transform the graph into an equivalent structured graph (if possible). These transformations do not take into account graphs generated from short-circuit evaluation languages, where the operands of a compound Boolean condition are not all necessarily evaluated, and thus generate unstructured graphs according to this methodology.

In contrast, the structuring algorithms presented in this thesis transform an arbitrary control flow graph into a functional and semantical equivalent flow graph that is structured under a set of generic control structures available in most commonly used high-level languages, and that makes use of `goto` *jumps whenever the graph cannot be structured with the generic structures. These algorithms take into account graphs generated by short-circuit evaluation, and thus do not generate unnecessary* `goto` *jumps for these graphs.*

6.2 Graph Structuring

The structuring of a sample control flow graph is presented in an informal way. The algorithms used to structure graphs are explained in Section 6.6. The control flow graph of Figure 6-2 is a sample program that contains several control structures. The intermediate code has been analyzed by the data flow analysis phase, and all variables have been given names.

The aim of a structuring algorithm for decompilation is to determine all underlying control structures of a control flow graph, based upon a predetermined set of high-level control structures. If the graph cannot be structured with the predefined set of structures, `goto` jumps are used. These conditions ensure functional and semantical equivalence between the original and final graph.

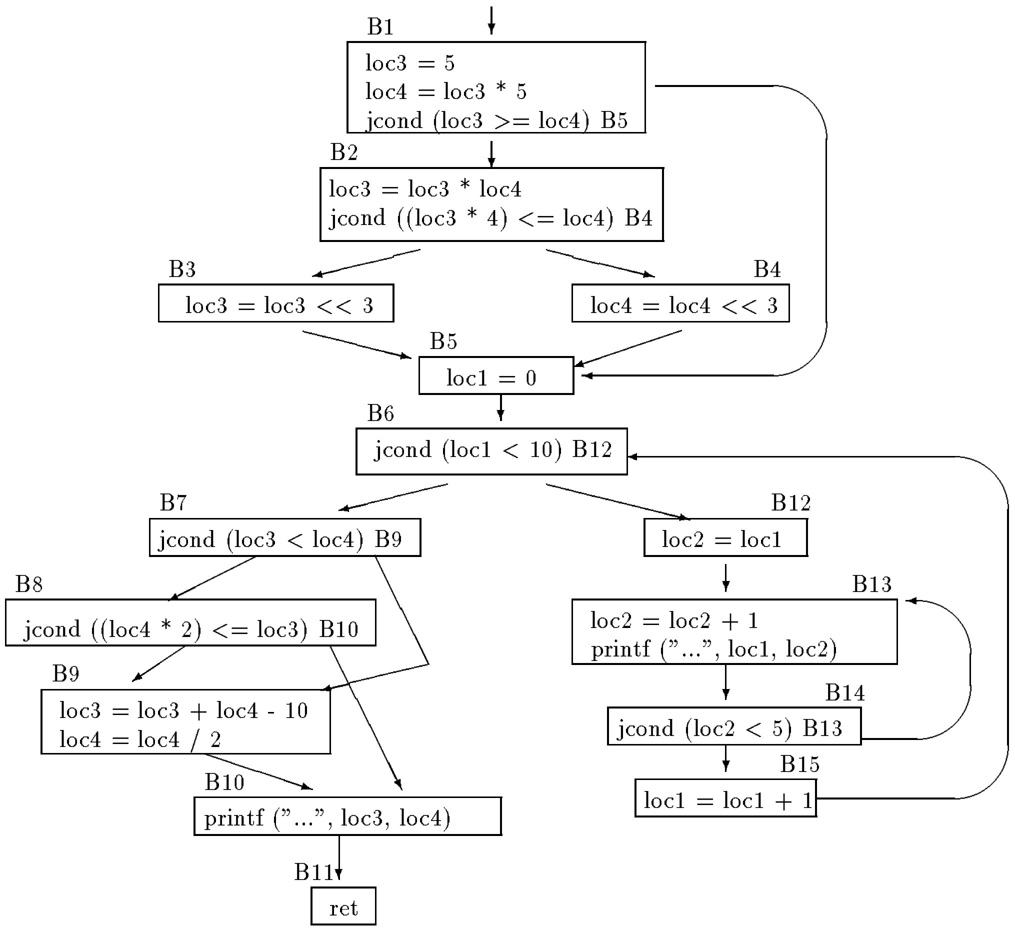


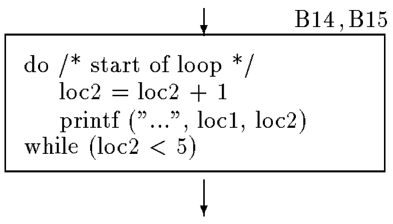
Figure 6-2: Sample Control Flow Graph

6.2.1 Structuring Loops

In graphs, loops are detected by the presence of a back-edge; that is, an edge from a "lower" node to a "higher" node. The notion of lower and higher are not formally defined yet, but can be thought as the nodes that are lower and higher up in the diagram (for a graph that is drawn starting at the top). In the graph of Figure 6-2 there are 2 back-edges: (B14,B13), and (B15,B6). These back-edges represent the extent of 2 different loops.

The type of the loop is detected by checking the header and the last node of the loop. The loop (B14,B13) has no conditional check on its header node, but the last node of the loop tests whether the loop should be executed again or not; thus, this is a post-tested loop, such as a `do..while()` in C, or a `repeat..until()` in Modula-2. The subgraph that represents this loop can be logically transformed into the subgraph of Figure 6-3, where the loop subgraph was replaced by one node that holds all the intermediate code instructions, as well as information on the type of loop.

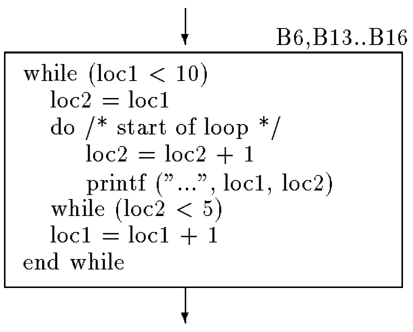
The loop (B15,B6) has a conditional header node that determines whether the loop is executed or not. The last node of this loop is a 1-way node that transfers control back



```
do /* start of loop */
    loc2 = loc2 + 1
    printf ("...", loc1, loc2)
while (loc2 < 5)
```

Figure 6-3: Post-tested Loop

to the header of the loop. This loop is clearly a pre-tested loop, such as a `while()` loop in a variety of languages. The subgraph of this loop can be logically transformed into the subgraph of Figure 6-4, where the loop subgraph has been replaced with the one node that holds all information on the loop and its instructions.



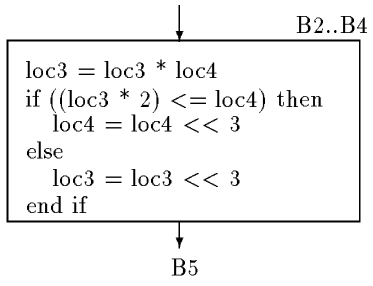
```
while (loc1 < 10)
   loc2 = loc1
   do /* start of loop */
       loc2 = loc2 + 1
       printf ("...", loc1, loc2)
   while (loc2 < 5)
   loc1 = loc1 + 1
end while
```

Figure 6-4: Pre-tested Loop

6.2.2 Structuring Conditionals

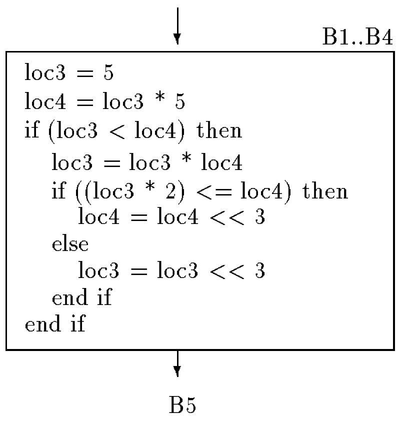
The 2-way conditional node B2 branches control to node B4 if the condition `(loc3 * 2) <= loc4` is true, otherwise it branches to node B3. Both these nodes are followed by the node B5, in other words, the conditional branch that started at node B2 is finished at node B5. This graph is clearly an `if()then..else` structure, and can be logically transformed into the subgraph of Figure 6-5, where the node represents basic blocks B2, B3, and B4. Note that all instructions before the conditional jump that belong to the same basic block are not modified.

The 2-way conditional node B1 transfers control to node B5 if the condition `loc3 >= loc4` is true, otherwise it transfers control to node B2. From out previous example, node B2 has been merged with nodes B3 and B4, and transformed into an equivalent node with an out-edge to node B5; thus, there is a path from node B2 $\rightarrow$ B5. Since B5 is one of the target branch nodes of the conditional at node B1, and it is reached by the other branch of the conditional, this 2-way node represents a single branch conditional (i.e. an `if()then`). This subgraph can be transformed into the node of Figure 6-6, where the condition at node B1 has been negated since the false branch is the single branch that forms part of the `if`.



```
loc3 = loc3 * loc4
if ((loc3 * 2) <= loc4) then
   loc4 = loc4 << 3
else
   loc3 = loc3 << 3
end if
```

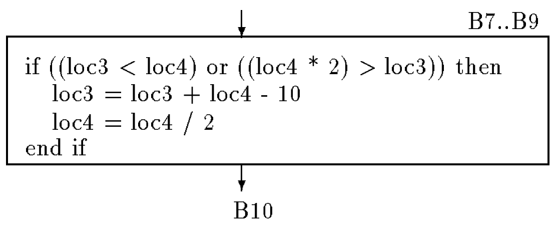
Figure 6-5: 2-way Conditional Branching



```
loc3 = 5
loc4 = loc3 * 5
if (loc3 < loc4) then
   loc3 = loc3 * loc4
   if ((loc3 * 2) <= loc4) then
      loc4 = loc4 << 3
   else
      loc3 = loc3 << 3
   end if
end if
```

Figure 6-6: Single Branch Conditional

The 2-way conditional nodes B7 and B8 are not trivially structured, since, if node B8 is considered the head of an `if..then..else` finishing at node B10, and node B7 is considered head of an `if..then`, we do not enter the subgraph headed by B8 at its entry point, but in one of the clauses of the conditional branch. If we structure node B7 first as an `if..then`, then node B8 branches out of the subgraph headed at B7 through another node other than the exit node B9; thus, the graph cannot be structured with the `if..then`, and `if..then..else` structures. But since both B7 and B8 only have a conditional branch instruction, these two conditions could be merged into a compound conditional in the following way: node B9 is reached whenever the condition in node B7 is true, or when the condition at B7 is false and the condition at B8 is false as well. Node B10 is reached whenever the condition at node B7 is false and the one at B8 is true, or by a path from node B9. This means that node B9 is reached whenever the condition at node B7 is true or the condition at node B8 is false, and the final end node is basic block B10. The final compound condition is shown in Figure 6-7, along with the transformed subgraph.



```
if ((loc3 < loc4) or ((loc4 * 2) > loc3)) then
   loc3 = loc3 + loc4 - 10
   loc4 = loc4 / 2
end if
```

Figure 6-7: Compound Conditional Branch

6.3 Control Flow Analysis

Information on the control structures of a program is available through *control flow analysis* of the program's graph. Information is collected in the different nodes of the graph, whether they belong to a loop and/or conditional, or are not part of any structure. This section defines control flow terminology available in the literature; for more information refer to [All72, Tar72, Tar74, HU75, Hec77, ASU86b].

6.3.1 Control Flow Analysis Definitions

The following definitions define basic concepts used in control flow analysis. These definitions make use of a directed graph $G = (N, E, h)$.

Definition 44 *A* **path** *from n_1 to n_v; $n_1, n_v \in N$, represented $n_1 \rightarrow n_v$, is a sequence of edges $(n_1, n_2), (n_2, n_3), \ldots, (n_{v-1}, n_v)$ such that $(n_i, n_{i+1}) \in E, \forall 1 \leq i < v, v \geq 1$.*

Definition 45 *A* **closed path** *or* **cycle** *is a path $n_1 \rightarrow n_v$ where $n_1 = n_v$.*

Definition 46 *The* **successors** *of $n_i \in N$ are $\{n_j \in N \mid n_i \rightarrow n_j\}$ (i.e. all nodes reachable from n_i).*
The **immediate successors** *of $n_i \in N$ are $\{n_j \in N \mid (n_i, n_j) \in E\}$.*

Definition 47 *The* **predecessors** *of $n_j \in N$ are $\{n_i \in N \mid n_i \rightarrow n_j\}$ (i.e. all nodes that reach n_j).*
The **immediate predecessors** *of $n_j \in N$ are $\{n_i \in N \mid (n_i, n_j) \in E\}$.*

Definition 48 *A node $n_i \in N$* **back dominates** *or* **predominates** *a node $n_k \in N$ if n_i is on every path $h \rightarrow n_k$. It is said that n_i* **dominates** *n_k.*

Definition 49 *A node $n_i \in N$* **immediately back dominates** *$n_k \in N$ if $\nexists n_j \bullet n_j$ back dominates $n_k \wedge n_i$ back dominates n_j (i.e. n_i is the closest back dominator to n_k). It is said that n_i is the* **immediate dominator** *of n_k.*

Definition 50 *A* **strongly connected region** *(SCR) is a subgraph $S = (N_S, E_S, h_S)$ such that $\forall n_i, n_j \in N_S \bullet \exists\, n_i \rightarrow n_j \wedge n_j \rightarrow n_i$.*

Definition 51 *A* **strongly connected component** *of G is a subgraph $S = (N_S, E_S, h_S)$ such that*

- *S is a strongly connected region.*
- *$\nexists S_2$ strongly connected region of $G \bullet S \subset S_2$.*

Definition 52 Depth first search *(DFS) is a traversal method that selects edges to traverse emanating from the most recently visited node which still has unvisited edges.*

A DFS algorithm defines a partial ordering of the nodes of G. The *reverse postorder* is the numbering of nodes during their last visit; the numbering starts with the maximum number of nodes in the graph, and finishes at 1. Throughout this chapter, all numbered graphs use the reverse postorder numbering scheme.

Definition 53 *A* **depth first spanning tree** *(DFST) of a flow graph G is a directed, rooted, ordered spanning tree of G grown by a DFS algorithm. A DFST T can partition the edges in G into three sets:*

1. *Back edges $= \{(v, w) : w \rightarrow v \in T\}$.*
2. *Forward edges $= \{(v, w) : v \rightarrow w \in T\}$.*
3. *Cross edges $= \{(v, w) : \nexists \ (v \rightarrow w$ or $w \rightarrow v)$ and $w \leq v$ in preorder$\}$.*

6.3.2 Relations

Definition 54 *Let R be a relation on a set S,*
Then xRy denotes $(x, y) \in R$.

Definition 55 *Let R be a relation on a set S,*
Then

- *the* **reflexive closure** *of R is $R^\alpha = R \cup \{(x,x) | x \in S\}$*
- *the* **transitive closure** *of R is $R^\beta = R^1 \cup R^2 \cup \ldots$, where $R^1 = R$ and $R^i = RR^{i-1}$ for $i \geq 2$*
- *the* **reflexive transitive closure** *of R is $R^* = R^\alpha \cup \ R^\beta$*
- *the* **completion** *of R is $\hat{R} = \{(x, y) \in S \times S \,|\, xR^*y \wedge \ \nexists \ z \in S \bullet yRz\}$.*

Definition 56 *Let R be a relation on a set S,*
Then (S,R) is **finite Church-Rosser (fcr)** *if and only if:*

1. *R is finite, i.e. $\forall \, p \in S \bullet \exists \, k_p \bullet \ pR^iq \Rightarrow i \leq k_p$.*
2. *$\hat{R}$ is a function, i.e. $p\hat{R}q \wedge p\hat{R}r \Rightarrow q = r$.*

6.3.3 Interval Theory

An *interval* is a graph theoretic construct defined by J.Cocke in [Coc70], and widely used by F.Allen for control flow analysis[All70, AC72] and data flow analysis[All72, All74, AC76]. The following sections summarize interval theory concepts.

Intervals

Definition 57 *Given a node h, an* **interval** *$I(h)$ is the maximal, single-entry subgraph in which h is the only entry node and in which all closed paths contain h. The unique interval node h is called the* **interval head** *or simply the header node.*

By selecting the correct set of header nodes, G can be partitioned into a unique set of disjoint intervals $\mathcal{I} = \{I(h_1), I(h_2), \ldots, I(h_n)\}$, for some $n \geq 1$. The algorithm to find the unique set of intervals of a graph is described in Figure 6-8. This algorithm makes use of the following variables: H (set of header nodes), $I(i)$ (set of nodes of interval i), and $\mathcal{I}$ (list of intervals of the graph G), as well as the function immedPred(n) which returns the next immediate predecessor of n.

```
procedure intervals ($G = (N, E, h)$)
/* Pre: $G$ is a graph.
 * Post: the intervals of $G$ are contained in the list $\mathcal{I}$. */

 $\mathcal{I} := \{\}$.
 $H := \{h\}$.
 for (all unprocessed $n \in H$) do
     $I(n) := \{n\}$.
     repeat
         $I(n) := I(n) + \{m \in N \mid \forall p \in \text{immedPred}(m) \bullet p \in I(n)\}$.
     until
         no more nodes can be added to $I(n)$.
     $H := H + \{m \in N \mid m \notin H \wedge m \notin I(n) \wedge (\exists\, p \in \text{immedPred}(m) \bullet p \in I(n))\}$.
     $\mathcal{I} := \mathcal{I} + I(n)$.
 end for
end procedure
```

Figure 6-8: Interval Algorithm

The example in Figure 6-9 shows a graph G with its intervals in dotted boxes. This graph has two intervals, I(1) and I(2). Interval I(2) contains a loop, the extent of this loop is given by the back-edge (4,2).

Definition 58 *The* **interval order** *is defined as the order of nodes in an interval list, given by the intervals algorithm of Figure 6-8.*

Some interval properties:

1. The header node back dominates each node in the interval.

2. Each strongly connected region in the interval must contain the header node.

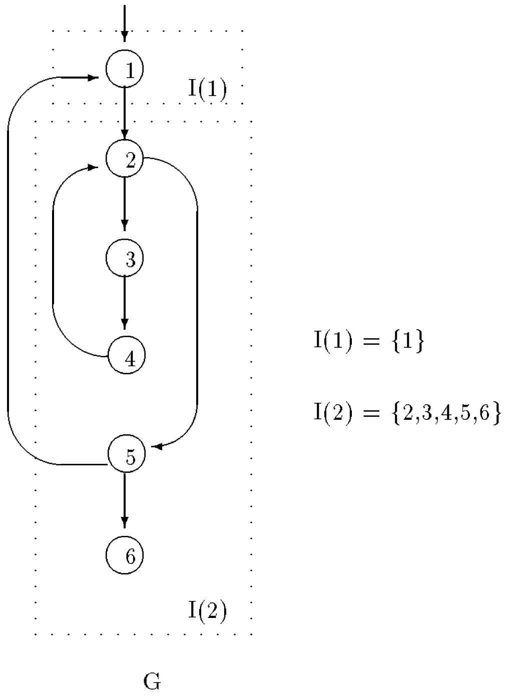


Figure 6-9: Intervals of a Graph

3. The interval order is such that if all nodes are processed in the order given, then all interval predecessors of a node reachable along loop free paths from the header will have been processed before the given node.

Definition 59 *A* **latching node** *is any node in the interval which has the header node as an immediate successor.*

Derived Sequence Construction

The **derived sequence of graphs**, $G^1 \dots G^n$, was described by F.Allen[All70, All72] based on the intervals of graph G. The construction of graphs is an iterative method that collapses intervals into nodes. G is the first order graph, represented G^1. The second order graph, G^2, is derived from G^1 by collapsing each interval in G^1 into a node. The immediate predecessors of the collapsed node are the immediate predecessors of the original header node which are not part of the interval. The immediate successors are all the immediate, non-interval successors of the original exit nodes. Intervals for G^2 are computed with the interval algorithm, and the graph construction process is repeated until a limit flow graph G^n is reached. G^n has the property of being a trivial graph (i.e. single node) or an irreducible graph. Figure 6-10 describes this algorithm.

Definition 60 *The* **n-th order graph** *or* **limit flow graph**, *G^n, of a graph G is defined as the graph G^{i-1}, $i \geq 1$, constructed by the derivedSequence algorithm of Figure 6-10, such that $G^{i-1} = G^i$.*

Definition 61 *A graph G is* **reducible** *if its n-th order graph G^n is trivial.*

```
procedure derivedSequence (G = (N,E,h))
/* Pre: $G$ is a graph.
 * Post: the derived sequence of $G$, $G^1 \ldots G^n, n \geq 1$ has been constructed. */

 $G^1 = G$.
 $\mathcal{I}^1$ = intervals($G^1$).
 $i = 2$.
 repeat /* Construction of $G^i$ */
     $N^i = \{n^i \mid I^{i-1}(n^{i-1}) \in \mathcal{I}^{i-1}\}$
     $\forall n \in N^i \bullet p \in \text{immedPred}(n) \Leftrightarrow (\exists m \in N^{i-1} \bullet m \in I^{i-1}(m) \wedge$
         $p \in immedPred(m) \wedge p \notin I^{i-1}(m))$.
     $(h^i_j, h^i_k) \in E^i \Leftrightarrow (\exists\, n, m, h^{i-1}_j, h^{i-1}_k \in N^{i-1} \bullet h^{i-1}_j = I^{i-1}(h^{i-1}_j) \wedge$
         $h^{i-1}_k = I^{i-1}(h^{i-1}_k) \wedge m \in I^{i-1}(h^{i-1}_j) \wedge n \in I^{i-1}(h^{i-1}_k) \wedge (m,n) \in E^{i-1}$.
     $i = i + 1$.
 until
     $G^i == G^{i-1}$.
end procedure
```

Figure 6-10: Derived Sequence Algorithm

The construction of the derived sequence is illustrated in Figure 6-11. The graph G^1 is the initial control flow graph G. G^1 has 2 intervals, previously described in Figure 6-9. Graph G^2 represents the intervals of G^1 as nodes. G^2 has a loop in its unique interval. This loop represents the loop extended by the back-edge (5,1). Finally, G^3 has no loops and is a trivial graph.

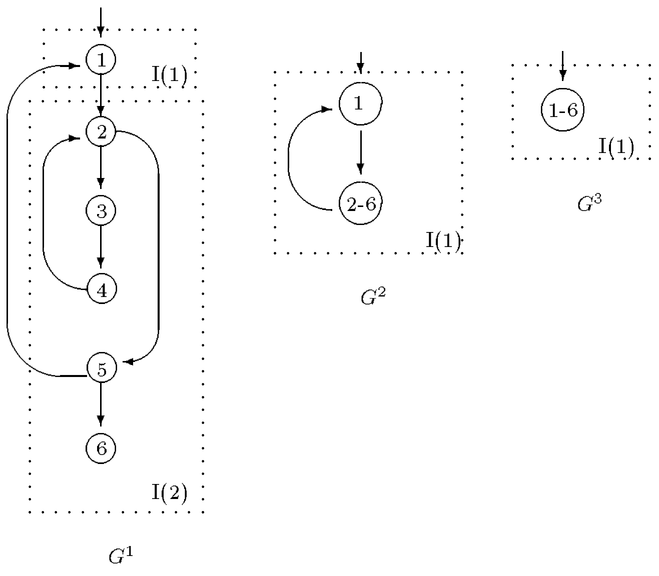


Figure 6-11: Derived Sequence of a Graph

Implementation Considerations

To compute the intervals of a graph G, G needs to be defined in terms of its predecessors and successors (i.e. an adjacency-type graph representation). With the aid of extra data structures, Hecht presented an optimized algorithm to find intervals[Hec77], of complexity $O(e)$, $\mid E \mid = e$.

6.3.4 Irreducible Flow Graphs

An irreducible flow graph is a graph such that its n-th order graph is not a trivial graph (by interval reduction). Irreducible flow graphs are characterized by the existence of a forbidden canonical irreducible graph [HU72, HU74, Hec77]. The absence of this graph in a flow graph is enough for the graph to be reducible. The canonical irreducible graph is shown in Figure 6-12.

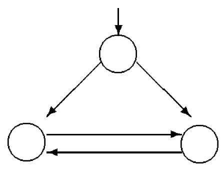


Figure 6-12: Canonical Irreducible Graph

Theorem 1 *A flow graph is irreducible if and only if it has a subgraph of the form canonical irreducible graph.*

6.4 High-Level Language Control Structures

Different high-level languages use different control structures, but in general, no high-level language uses all different available control structures. This section illustrates different control structures, gives a classification, and analyses the structures available in commonly used high-level languages such as C, Pascal, and Modula-2.

6.4.1 Control Structures - Classification

Control structures have been classified into different classes according to the complexity of the class. An initial classification was provided by Kosaraju in [Kos74], and was used to determine which classes were reducible to which other classes. This classification was expanded by Ledgard and Marcotty in [LM75], and was used to present a hierarchy of classes of control structures under semantical reducibility.

Figure 6-13 shows all the different control structures that are under consideration in this classification; these structures are:

1. Action: a single basic block node is an action.

2. Composition: a sequence of 2 structures is a composition.

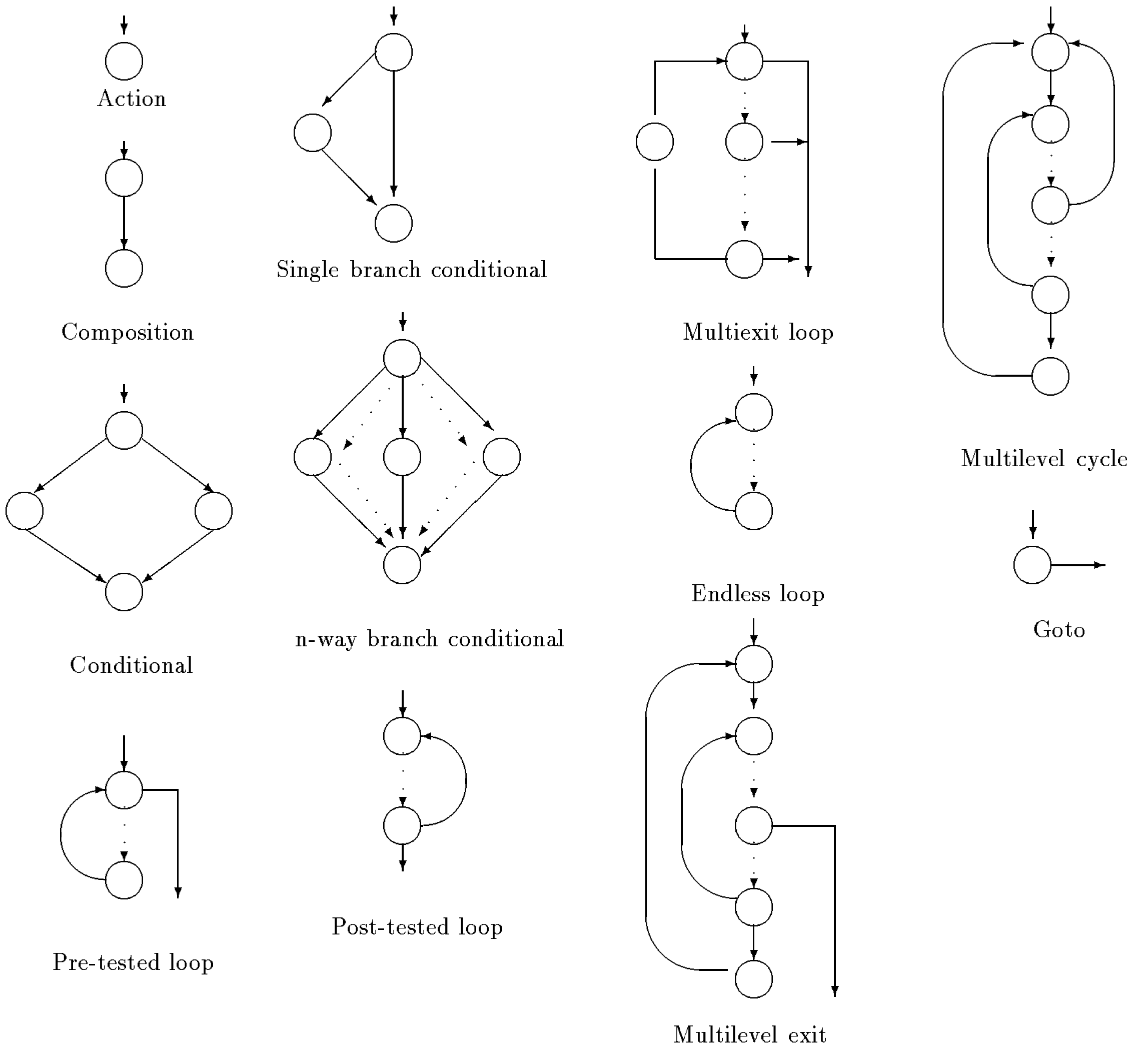


Figure 6-13: High-level Control Structures

3. Conditional: a structure of the form `if p then s1 else s2`, where `p` is a predicate and `s1,s2` are structures is a conditional structure.

4. Pre-tested loop: a loop of the form `while p do s`, where `p` is a predicate and `s` is a structure, is a pre-tested loop structure.

5. Single branch conditional: a conditional of the form `if p then s`, where `p` is a predicate and `s` is a structure, is a single branch conditional structure.

6. n-way conditional: a conditional of the form

```
case p of
 1 : s1
 2 : s2
 ...
 n : sn
```

```
end case
```

where `p` is a predicate and `s1..sn` are structures, is an n-way conditional structure.

7. Post-tested loop: a loop of the form `repeat s until p`, where `s` is a structure and `p` is a predicate, is a post-tested loop structure.

8. Multiexit loop: a loop of the form

```
while p1 do
  s1
  if p2 then exit
  s2
  if p3 then exit
  ...
  if pn then exit
  sn
end while
```

where `s1..sn` are structures and `p1..pn` are predicates, is a multiexit loop structure. Each `exit` statement branches out of the loop to the first statement/basic block after the loop.

9. Endless loop: a loop of the form `loop s end`, where `s` is a structure, is an endless loop.

10. Multilevel exit: an `exit(i)` statement causes the termination of `i` enclosing endless loops.

11. Multilevel cycle: a `cycle(i)` statement causes the i-th enclosing endless loop to be re-executed.

12. Goto: a `goto` statement transfers control to any other basic block, regardless of unique entrance conditions.

Based on these 12 different structures, control structures are classified into the following classes:

- D structures: D for Dijkstra. D = {1,2,3,4}
- D' structures: extension of D structures. D' = {1,2,3,4,5,6,7}
- BJn structures: BJ for Böhm and Jacopini, n for the maximum number of predicates in a multiexit loop. BJn = {1,2,3,8}
- REn structures: RE for Repeat-End, n for the maximum number of exit levels. REn = {1,2,3,9,10}
- RECn structures: REC for Repeat-End with `cycle(i)` structures, n for the number of levels. RECn = {1,2,3,9,10,11}
- DREn structures: DRE for Repeat-End and Do-while loops, n for the maximum number of enclosing levels to exit. DREn = {1,2,3,4,9,10}

- DRECn structures: DREC for Repeat-End, Do-while, and cycle(i) structures, n for the maximum number of enclosing endless loops. DRECn = {1,2,3,4,9,10,11}
- GPn structures: any structure that has one-in, one-out substructures that have at most n different predicates. GPn = {1..7,9}
- L structures: any well-formed structure. There are no restrictions on the number of predicates, actions, and transfers of control; therefore, `goto` statements are allowed. L = {1..12}

Definition 62 *Let s1 and s2 be two structures, then s1 is a* **semantical conversion** *of s2 if and only if*

- *For every input, s2 computes the same function as s1.*
- *The primitive actions and predicates of s2 are precisely those of s1.*

In other words, no new semantics such as variables, actions, or predicates, are allowed by this conversion.

Based on semantical conversion, the classes of control structures form a hierarchy, as shown in Figure 6-14. The classes higher up in the hierarchy are a semantical conversion of the lower classes.

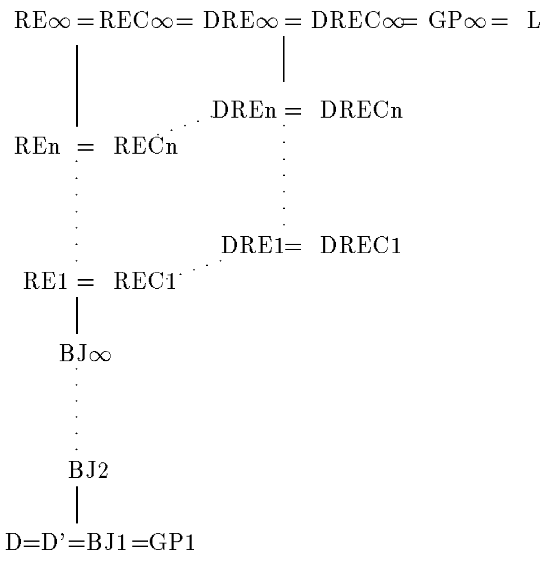


Figure 6-14: Control Structures Classes Hierarchy

6.4.2 Control Structures in 3rd Generation Languages

In this section, different high-level languages are analysed and classified in terms of their control structures. The selected languages are used in a variety of applications, including systems programming, numerical or scientifical applications, and multipurpose applications; these languages are: Modula-2, Pascal, C, Fortran, and Ada.

Modula-2 [Wir85, PLA91] does not allow for the use of `goto` statements, therefore, the control flow graphs generated by this language are structured and reducible. Modula-2 has all D'-type structures: 2-way conditionals (`IF p THEN s1 {ELSE s2}`), n-way conditional (`CASE p OF ... END`, pre-tested loop (`WHILE p DO`), post-tested loop (`REPEAT s UNTIL p`), and infinite loop (`LOOP s END`). An endless loop can be terminated by one or more `EXIT` statements within the statement sequence body of the loop. This construct can be used to simulate other loop structures, such as a multiexit loop with n predicates (BJn structure). An `EXIT` statement terminates the execution of the immediately enclosing endless loop statement, and the execution resumes at the statement following the end of the loop. If an `EXIT` occurs within a pre-tested or post-tested loop nested within an endless loop, both the inner loop and enclosing endless loop are terminated; therefore, an `EXIT` statement is equivalent to an `exit(1)` statement, and belongs to the RE1 class of structures.

Pascal [Coo83] is not as strict as Modula-2, in the sense that it allows `goto` statements to be used. All D'-type structures are allowed: 2-way conditionals (`if p then s1 [else s2]`), n-way conditional (`case p of ... end`), pre-tested loop (`while p do`), post-tested loop (`repeat s until p`), and the endless loop is simulated by a `while()` with a true condition (`while (True) do`). `Goto`s can be used to simulate multiexit and multilevel loops, but can also be used in an unstructured way, to enter in the middle of a structure; therefore, L class structures are permitted in this language.

C [KR88] allows for structured and unstructured transfer of control. D' structures are represented by the following statements: 2-way conditional (`if (p) s1 [else s2]`), n-way conditional (`switch (p) {...}`), pre-tested loop (`while (p) {s}`), post-tested loop (`do s while (p)`), and endless loop (`for (;;;)` or `while (1) {s}`). 1 level exit of control is allowed by the use of `break` statements, and 1 level cycle transfer of control is allowed by the use of the `continue` statement; therefore, C contains structures of the RE1 and REC1 classes. The use of `goto` statements can model any structure from the DRECn class, but can also produce unstructured graphs; therefore C allows for L class structures.

Fortran [Col81] has different types of conditionals which include: 2-way conditional (`IF (p) s1,s2`), arithmetic if or 3-way conditional (`IF (p) s1,s2,s3`), and computed goto statements or n-way conditionals (`GOTO (s1,s2,...,sn) p`). Pre-tested, post-tested and endless loops are all simulated by use of the `DO` statement; therefore, all D'-type structures are allowed in Fortran. Finally, `goto` statements are allowed, producing structured or unstructured transfers of control, allowing for L type structures.

Ada [DoD83] allows most D'-type structures, including: 2-way conditionals (`if p then s1 [else s2]`), n-way conditional (`case p is ... end`), pre-tested loop (`while p` and `for` loops), and endless loop (`loop s end loop`). Ada also allows the use of the `exit` statements to exit from within named endless loops; therefore, several nested loops can be terminated with this instruction (i.e. REn class type structure). `Goto` statements are allowed in a restricted way; they can transfer control only to a statement of an enclosing sequence of statements, but not the reverse. Also, it is prohibited to transfer control into the alternatives of a `case` statement, or an `if..then..else` statement. These restrictions on the use of `goto`s makes them simulate multilevel exits and multilevel continues, but do not permit unstructured transfers of control; therefore, up to DRECn-type structures can be built in this language.

Figure 6-15 summarizes the different types of classes of structures available in the set of distinguished languages. It must be pointed out that all of these languages make use of D'-type structures, plus one or more structures that belong to different types of classes. Unstructured languages allow for the unstructured use of `goto`, which is the case of Pascal and Fortran. Structured uses of `goto`, such as in Ada, permit the construction of structured control flow graphs, since up to DRECn-type structures can be simulated by these `goto`s.

Language	Control Structure Classification
Modula-2	D' + BJn + RE1
Pascal	D' + L
C	D' + BJn + DREC1 + L
Fortran	D' + L
Ada	D' + DRECn

Figure 6-15: Classes of Control Structures in High-Level Languages

6.4.3 Generic Set of Control Structures

In order to structure a graph, a set of generic control structures needs to be selected. This set must be general enough to cater for commonly used structures in a variety of languages. From the review of some 3rd generation languages in the previous section, it is clear that most languages have D' class structures, plus some type of structured or unstructured transfer of control (i.e. multilevel exits or `goto`s). Structures from the REn, RECn, DREn, and DRECn classes can all be simulated by the use of structured transfers of control via a `goto` statement. Since most of the languages allow the use of `goto`, and not all languages have the same multilevel exit or multilevel continue structures, `goto` is a better choice of a generic construct than `exit(i)` or `cycle(i)`. It is therefore desirable to structure a control flow graph using the following set of generic structures:

- Action
- Composition
- Conditional
- Pre-tested loop
- Single branching conditional
- n-way conditional
- Post-tested loop
- Endless loop
- Goto

In other words, the generic set of control structures has all D' and L class structures.

6.5 Structured and Unstructured Graphs

A structured control flow graph is a graph generated from programs that use structures of up to the DRECn class; i.e. a graph that is decomposable into subgraphs with one entry and one or more exits. Languages that allow the use of `goto` can still generate structured graphs, if the `goto`s are used to transfer control in a structured way (i.e. to transfer control to the start or the end of a structure). Unstructured graphs are generated by the unstructured transfer of control of `goto` statements, that is, a transfer of control in the middle of a structured graph, which breaks the previously structured graph into an unstructured one since there is more than one entry into this subgraph. Unstructuredness can also be introduced by the optimization phase of the compiler when code motion is performed (i.e. code is moved).

6.5.1 Loops

A loop is a strongly connected region in which there is a path between any two nodes of the directed subgraph. This means that there must be at least one back-edge to the loop's header node.

A **structured loop** is a subgraph that has one entry point, one back-edge, and possibly one or more exit points that transfer control to the same node. Structured loops include all natural loops (pre-tested and post-tested loops), endless loops, and multiexit loops. These loops are shown in Figure 6-16.

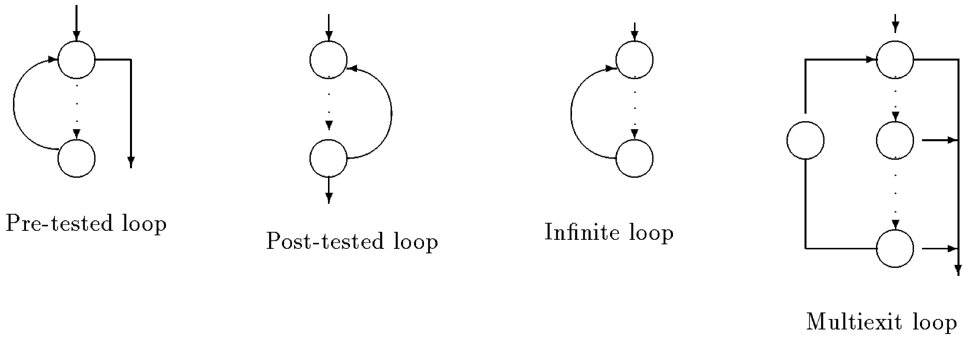


Figure 6-16: Structured Loops

An **unstructured loop** is a subgraph that has one or more back-edges, one or more entry points, and one or more exit points to different nodes. Figure 6-17 illustrates four different types of unstructured loops:

- Multientry loop: a loop with two or more entry points.
- Parallel loop: a loop with two or more back-edges to the same header node.
- Overlapping loops: two loops that overlap in the same strongly connected region.
- Multiexit loop: a loop with two or more exits to different nodes.

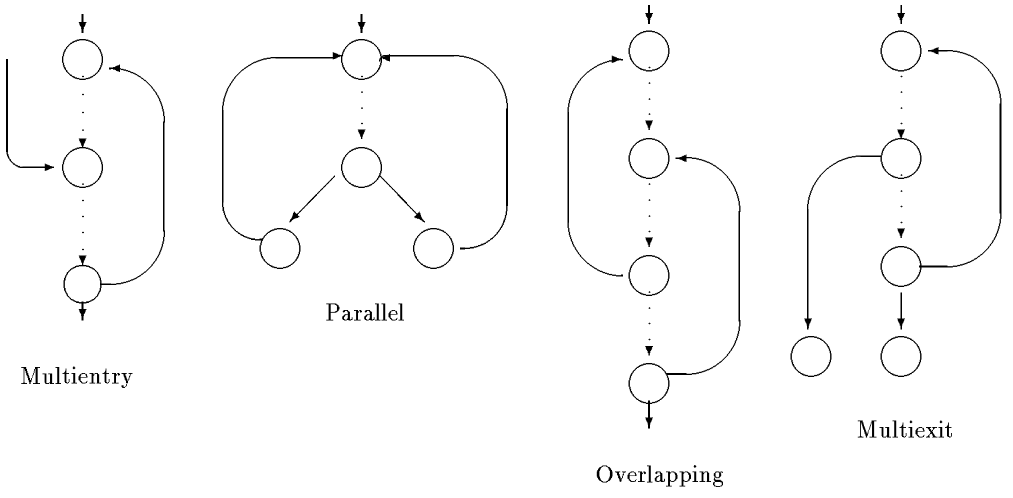


Figure 6-17: Sample Unstructured Loops

The **follow** node of a structured or unstructured loop is the first node that is reached from the exit of the loop. In the case of unstructured loops, one node is considered the loop exit node, and the first node that follows it is the follow node of the loop.

6.5.2 Conditionals

A **structured 2-way conditional** is a directed subgraph with a 2-way conditional header node, one entry point, two or more branch nodes, and a common end node that is reached by both branch nodes. This final common end node is referred to as the follow node, and has the property of being immediately dominated by the header node.

In an `if..then` conditional, one of the two branch nodes of the header node is the follow node of the subgraph. In an `if..then..else` conditional, neither branch node is the follow node, but they both converge to a common end node. Figure 6-18 shows these two generic constructs, with the values of the out-edges of the header node; true or false. In the case of an `if..then`, either the true or the false edge leads to the follow node, thus, there are two different graphs to represent such a structure; whereas in the case of an `if..then..else`, the graph representation is unique.

In a similar way, a **structured n-way conditional** is a directed subgraph with one n-way entry header node (i.e. n successor nodes from the header node), and a common end node that is reached by the n successor nodes. This common end node is referred to as the follow node, and has the property of being dominated by the header node of the structure. A sample 4-way conditional is shown in Figure 6-19.

Unstructured 2-way conditionals are 2-way node header subgraphs, with two or more entries into the branches of the header node, or two or more exits from branches of the header node. These graphs are represented in the abnormal selection path graph, shown in Figure 6-20 (a). It is known from the graph structure that an `if..then..else` subgraph can start at nodes 1 and 2, generating the two subgraphs in Figure 6-20 (b) and (c). The

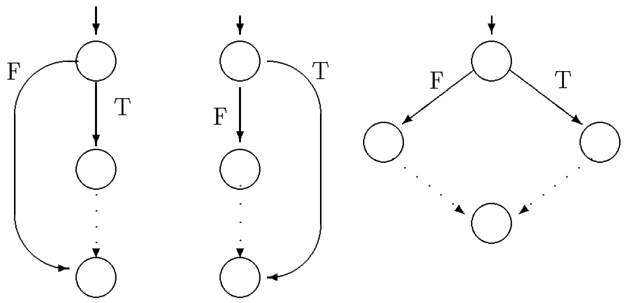


Figure 6-18: Structured 2-way Conditionals

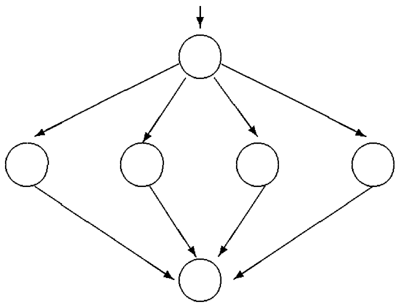


Figure 6-19: Structured 4-way Conditional

graph (b) assumes a 2-way conditional starting at node 2, with an abnormal entry at node 5. The graph (c) assumes a 2-way conditional starting at node 1, with an abnormal exit from node 2.

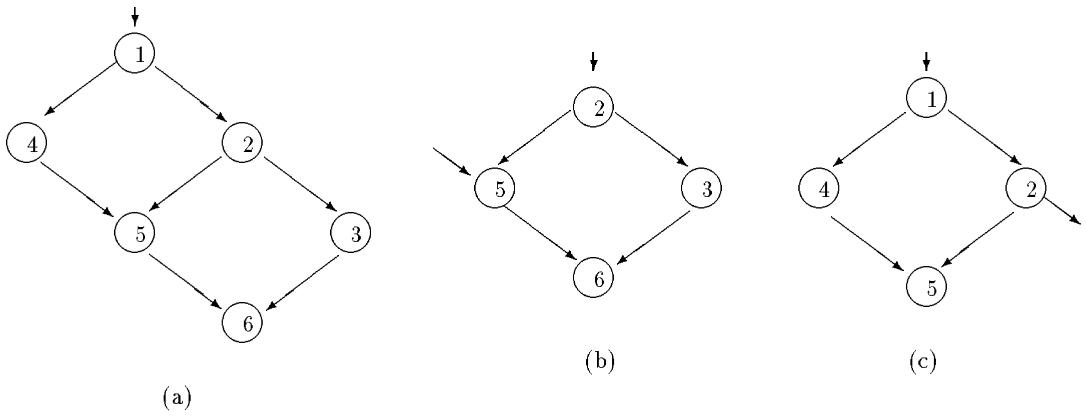


Figure 6-20: Abnormal Selection Path

In a similar way, **unstructured n-way conditionals** allow for two or more entries or exits to/from one or more branches of the n-way header node. Figure 6-21 shows four different cases of unstructured 3-way graphs: graph (a) has an abnormal forward out-edge from one of the branches, graph (b) has an abnormal backward out-edge from one of the branches, graph (c) has an abnormal forward in-edge into one of the branches, and graph (d) has an abnormal backward in-edge into one of the branches.

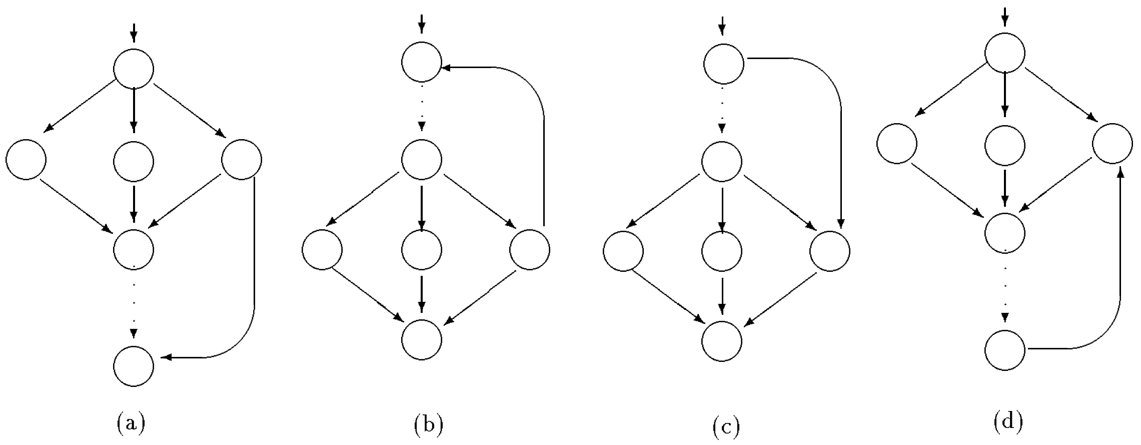


Figure 6-21: Unstructured 3-way Conditionals

6.5.3 Structured Graphs and Reducibility

A structured graph is one that is composed of structured subgraphs that belong to the class of graphs generated by DRECn structures. An informal demonstration is given to prove that all structured graphs of the class DRECn are reducible. Consider the informal graph grammar given in Figure 6-22. There are 11 production rules, each defining a different structured subgraph S. Each production rule indicates that a structured subgraph can be generated by replacing a node S with the associated right-hand side subgraph of the production.

Theorem 2 *The class DRECn of graphs is reducible.*
Demonstration: *The class DRECn of graphs is defined by the informal graph grammar of Figure 6-22. All the subgraphs in the right-hand side of the productions have the common property of having one entry point, and one or more exit points to a common target end point; in this way, transfers of control are done in a structured way. By theorem 1, it is known that a graph is irreducible if and only if it has a subgraph of the form of the canonical irreducible graph (see Figure 6-12). This canonical irreducible graph is composed of two subgraphs: a conditional branching graph, and a loop. The former subgraph has one entry point, and the latter subgraph has two (or more) entry points; which is what makes the graph irreducible. Since none of the productions of the graph grammar generate subgraphs that have more than one entry point, this graph grammar cannot generate an irreducible graph; thus, the graphs that belong to the DRECn class are reducible.*

6.6 Structuring Algorithms

In decompilation, the aim of a structuring algorithm is to determine the underlying control structures of an arbitrary graph, thus converting it into a functional and semantical

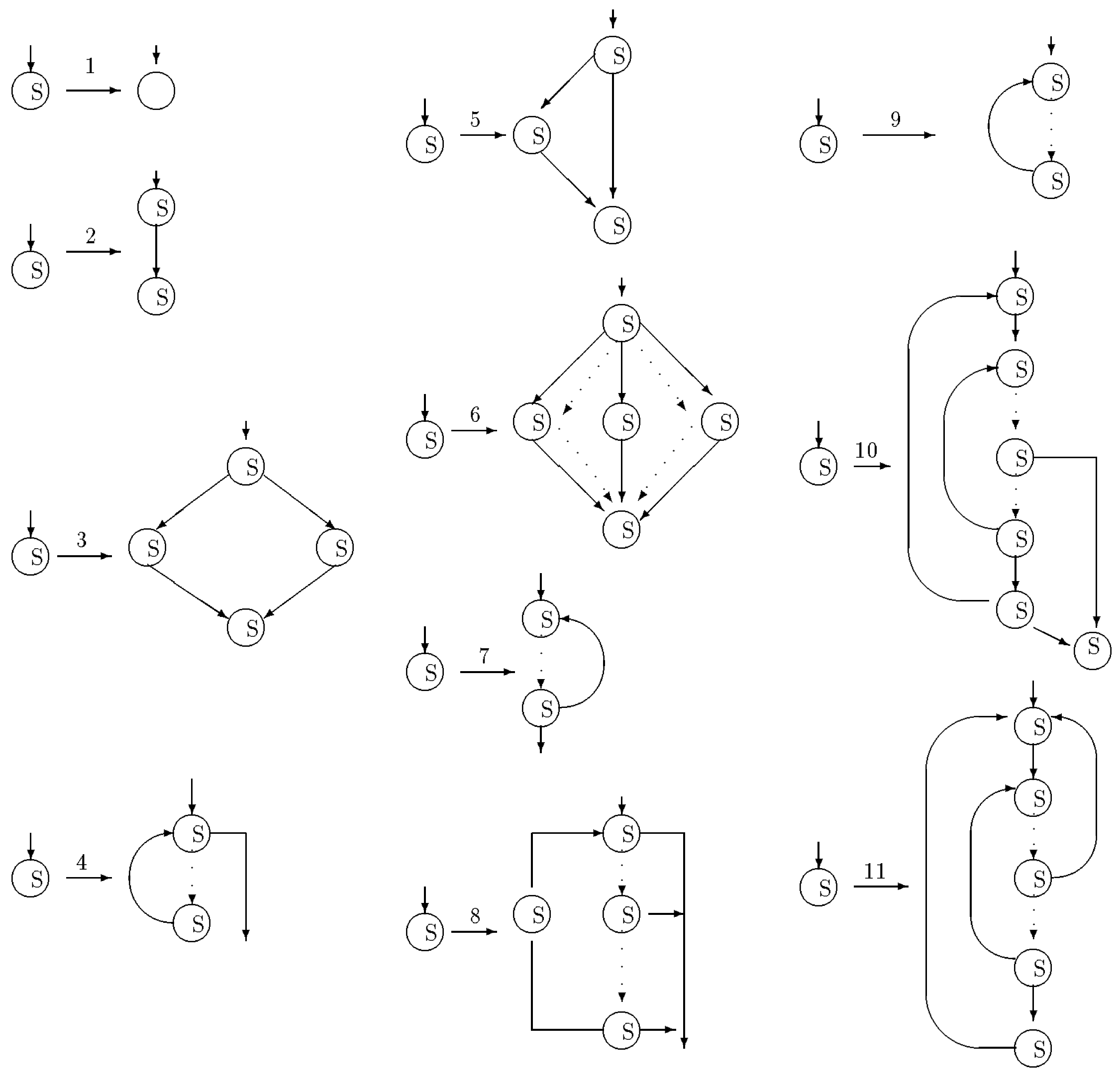


Figure 6-22: Graph Grammar for the Class of Structures DRECn

equivalent graph. Arbitrary graph stands for any control flow graph; reducible or irreducible, from a structured or unstructured language. Since it is not known what language the initial program was written in, and what compiler was used (e.g. what optimizations were turned on), the use of `goto` jumps must be allowed in case the graph cannot be structured into a set of generic high-level structures. The set of generic control structures of Section 6.4.3 is the one chosen for the structuring algorithm presented in this section.

6.6.1 Structuring Loops

In order to structure loops, a loop in terms of a graph representation needs to be defined. This representation must be able to not only determine the extent of a loop, but also provide a nesting order for the loops. As pointed out by Hecht in [Hec77], the representation of a loop by means of cycles is too fine a representation since loops are not necessarily properly nested or disjoint. The use of strongly connected components as loops is too coarse a representation as there is no nesting order. The use of strongly connected regions does not provide a unique cover of the graph, and does not cover the entire graph. Finally, the use of intervals does provide a representation that satisfies the abovementioned conditions: one loop per interval, and a nesting order provided by the derived sequence of graphs.

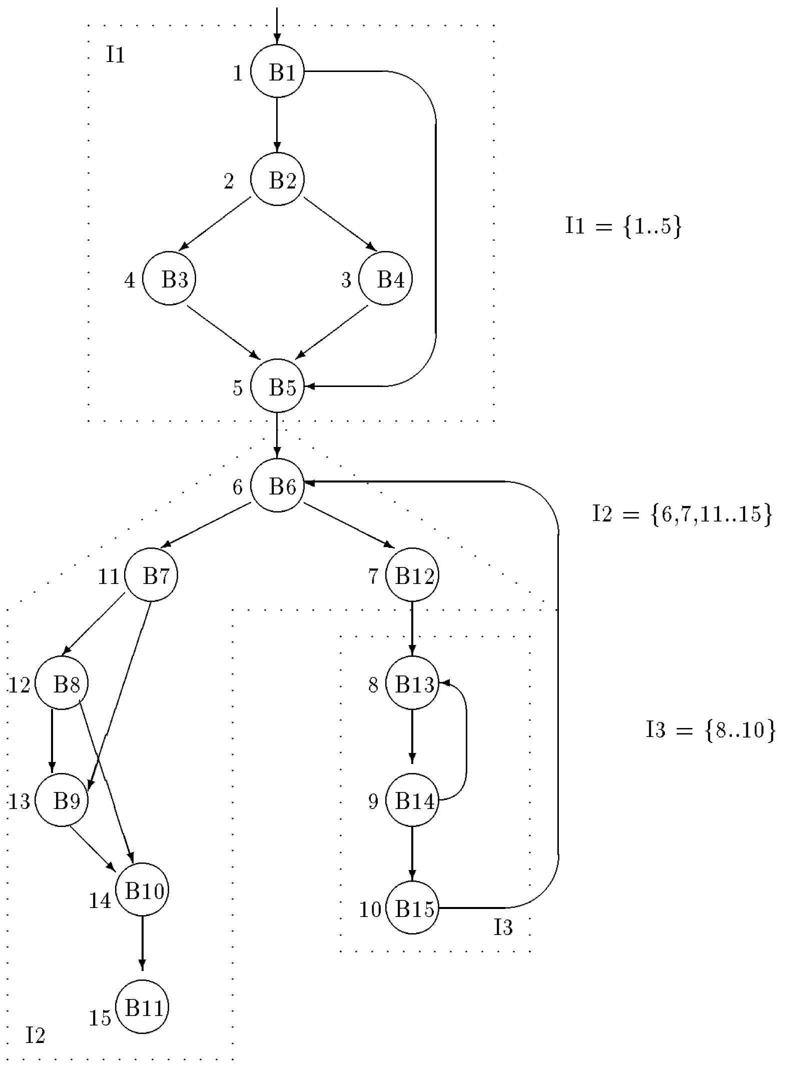


Figure 6-23: Intervals of the Control Flow Graph of Figure 6-2

Given an interval $I(h_j)$ with header h_j, there is a loop rooted at h_j if there is a back-edge to the header node h_j from a latching node $n_k \in I(h_j)$. Consider the graph in Figure 6-23, which is the same graph from Figure 6-2 without intermediate instruction information, and with intervals delimitered by dotted lines. There are 3 intervals: I_1 rooted at basic block B1, I_2 rooted at node B6, and I_3 rooted at node B13.

In this graph, interval I_3 contains the loop (B14,B13) in its entirety, and interval I_2 contains the header of the loop (B15,B6), but its latching node is in interval I_3. If each of the intervals are collapsed into individual nodes, and the intervals of that new graph are found, the loop that was between intervals I_3 and I_2 must now belong to the same interval. Consider the derived sequence of graphs $G^2 \ldots G^4$ in Figure 6-24. In graph G^2, the loop between nodes I_3 and I_2 is in interval I_5 in its entirety. This loop represents the corresponding loop of nodes (B15,B6) in the initial graph. It is noted that there are no more loops in these graphs, and that the initial graph is reducible since the trivial graph G^4 was derived by this process. It is noted that the length of the derived sequence is proportional to the maximum depth of nested loops in the initial graph.

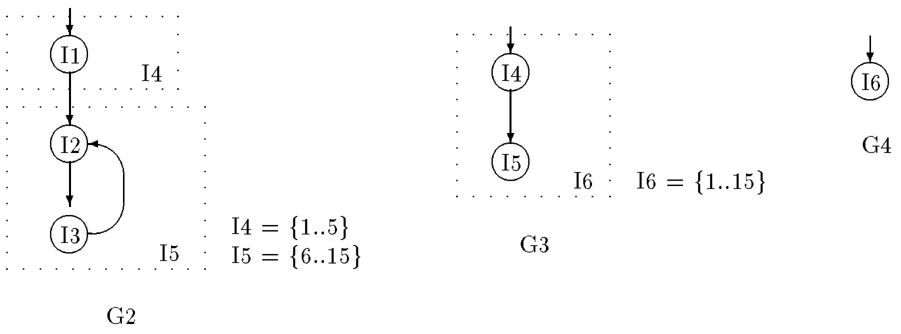


Figure 6-24: Derived Sequence of Graphs $G^2 \ldots G^4$

Once a loop has been found, the type of loop (e.g. pre-tested, post-tested, endless) is determined according to the type of header and latching nodes. Also, the nodes that belong to the loop are flagged as being so, in order to prevent nodes from belonging to two different loops, such as in overlapping, or multientry loops. These methods are explained in the following sections, for now we assume there are two procedures that determine the type of the loop, and mark the nodes that belong to that loop.

Given a control flow graph $G = G^1$ with interval information, the derived sequence of graphs $G^1, \ldots, G^n$ of G, and the set of intervals of these graphs, $\mathcal{I}^1 \ldots \mathcal{I}^n$, an algorithm to find loops is as follows: each header of an interval in G^1 is checked for having a back-edge from a latching node that belong to the same interval. If this happens, a loop has been found, so its type is determined, and the nodes that belong to it are marked. Next, the intervals of G^2, $\mathcal{I}^2$ are checked for loops, and the process is repeated until intervals in $\mathcal{I}^n$ have been checked. Whenever there is a potential loop (i.e. a header of an interval that has a predecessor with a back-edge) that has its header or latching node marked as belonging to another loop, the loop is disregarded as it belongs to an unstructured loop. These loops always generate `goto` jumps during code generation. In this algorithm no `goto` jumps and target labels are determined. The complete algorithm is given in Figure 6-25. This algorithm finds the loops in the appropriate nesting level, from innermost to outermost loop.

Finding the Nodes that Belong to a Loop

Given a loop induced by $(y, x), y \in I(x)$, it is noted that the two different loops that are part of the sample program in Figure 6-23 satisfy the following condition:

$$\forall n \in \text{loop}(y, x) \bullet \; n \in \{x \ldots y\}$$

```
procedure loopStruct ($G = (N, E, h)$)
/* Pre: $G^1 \ldots G^n$ has been constructed.
 *   $\mathcal{I}^1 \ldots \mathcal{I}^n$ has been determined.
 * Post: all nodes of $G$ that belong to a loop are marked.
 *   all loop header nodes have information on the type of loop and the latching node. */

 for ($G^i := G^1 \ldots G^n$)
     for ($I^i(h_j) := I^1(h_1) \ldots I^m(h_m)$)
       if ($(\exists x \in N^i \bullet (x, h_j) \in E^i) \wedge$ (inLoop($x$) == False))
         for (all $n \in$ loop $(x, h_j)$)
           inLoop($n$) = True
         end for
         loopType($h_j$) = findLoopType ($(x, h_j)$).
         loopFollow($h_j$) = findLoopFollow ($(x, h_j)$).
       end if
     end for
 end for
end procedure
```

Figure 6-25: Loop Structuring Algorithm

In other words, the loop is formed of all nodes that are between x and y in terms of node numbering. Unfortunately, it is not that simple to determine the nodes that belong to a loop. Consider the multiexit graphs in Figure 6-26, where each loop has one abnormal exit, and each different graph has a different type of edge being used in the underlying DFST. As can be seen, loops with forward edges, back edges, or cross edges satisfy the above mentioned condition. The graph with the tree edge includes more nodes though, as nodes 4 and 5 are not really part of the loop, but have a number between nodes 2 and 6 (the bound of the loop). In this case, an extra condition is needed to be satisfied, and that is, that the nodes belong to the same interval, since the interval header (i.e. x) dominates all nodes of the interval, and in a loop, the loop header node dominates all nodes of the loop. If a node belongs to a different interval, it is not dominated by the loop header node, thus it cannot belong to the same loop. In other words, the following condition needs to be satisfied:

$$\forall n \in \text{loop}(y, x) \bullet n \in I(x)$$

Given an interval $I(x)$ with a loop induced by $(y, x), y \in I(x)$, the nodes that belong to this loop satisfy two conditions: In other words, a node n belongs to the loop induced by (y, x) if it belongs to the same interval (i.e. it is dominated by x), and its order (i.e. reverse postorder number) is greater than the header node and lesser than the latching node (i.e. it is a node from the "middle" of the loop). These conditions can be simplified in the following expression:

$$n \in \text{loop}(y, x) \Leftrightarrow n \in (I(x) \cap \{x \ldots y\})$$

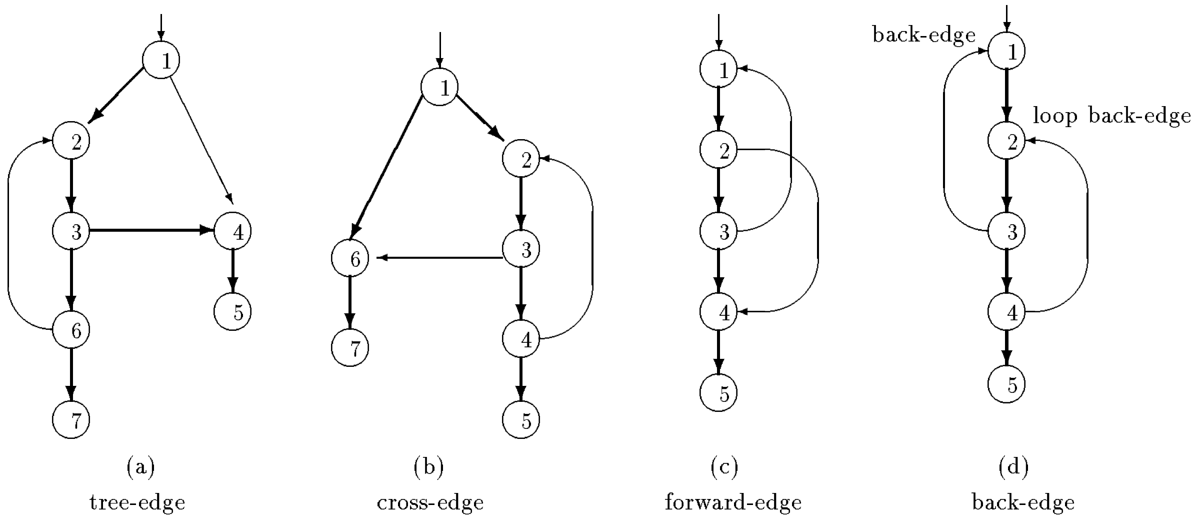


Figure 6-26: Multiexit Loops - 4 Cases

The loops from Figure 6-23 have the following nodes: loop (9,8) has only those two nodes, and loop (10,6) has all nodes between 6 and 10 that belong to the interval I5 (Figure 6-24) in G^2. These nodes are as follows:

- Loop (9,8) = {8,9}
- Loop (10,6) = {6..10}

The algorithm in Figure 6-27 finds all nodes that belong to a loop induced by a back-edge. These nodes are marked by setting their loop head to the header of the loop. Note that if an inner loop node has already been marked, it means that the node also belongs to a nested loop, and thus, its loopHead field is not modified. In this way, all nodes that belong to a loop(s) are marked by the header node of the most nested loop they belong to.

Determining the Type of Loop

The type of a loop is determined by the header and latching nodes of the loop. In a pre-tested loop, the 2-way header node determines whether the loop is executed or not, and the 1-way latching node transfers control back to the header node. A post-tested loop is characterized by a 2-way latching node that branches back to the header of the loop or out of the loop, and any type of header node. Finally, an endless loop has a 1-way latching node that transfers control back to the header node, and any type of header node.

The types of the two loops of Figure 6-23 are as follows: the loop (9,8) has a 2-way latching node and a call header node, thus, the loop is a post-tested loop (i.e. a `repeat..until()` loop). The loop (10,6) has a 1-way latching node and a 2-way header node, thus, the loop is a pre-tested loop (i.e. a `while()` loop).

In this example, the `repeat..until()` loop had a call header node, so there were no problems in saying that this loop really is a post-tested loop. A problem arises when both

```
procedure markNodesInLoop ($G = (N, E, h)$, $(y, x)$)
/* Pre: $(y, x)$ is a back-edge.
 * Post: the nodes that belong to the loop $(y, x)$ are marked. */

    nodesInLoop = $\{x\}$
    loopHead($x$) = $x$
    for (all nodes $n \in \{x + 1 \ldots y\}$)
        if ($n \in I(x)$)
            nodesInLoop = nodesInLoop $\cup\{n\}$
            if (loopHead($n$) == No_Node)
                loopHead($n$) = $x$.
            end if
        end if
    end for
end procedure
```

Figure 6-27: Algorithm to Mark all Nodes that belong to a Loop induced by (y, x)

the header and latching nodes are 2-way conditional nodes, since it is not known whether one or both branches of the header 2-way node branch into the loop or out of the loop; i.e. the loop would be an abnormal loop in the former case, and a post-tested loop in the latter case. It is therefore necessary to check whether the nodes of the branches of the header node belong to the loop or not, if they do not, the loop can be coded as a `while()` loop with an abnormal exit from the latching node. Figure 6-28 gives an algorithm to determine the type of loop based on the nodesInLoop set constructed in the algorithm of Figure 6-27.

Finding the Loop Follow Node

The loop follow node is the first node that is reached after the loop is terminated. In the case of natural loops, there is only one node that is reached after loop termination, but in the case of multiexit and multilevel exit loops, there can be more than one exit, thus, more than one node can be reached after the loop. Since the structuring algorithm only structured natural loops, all multiexit loops are structured with one "real" exit, and one or more abnormal exits. In the case of endless loops that have exits in the middle of the loop, several nodes can be reached after the different exits. It is the purpose of this algorithm to find only *one* follow node.

In a pre-tested loop, the follow node is the successor of the loop header that does not belong to the loop. In a similar way, the follow node of a post-tested loop is the successor of the loop latching node that does not belong to the loop. In endless loops there are no follow nodes initially, as neither the header nor the latching node jump out of the loop. But since an endless loop can have a jump out of the loop in the middle of the loop (e.g. a `break` in C), it can too have a follow node. Since the follow node is the first node that is reached after the loop is ended, it is desirable to find the *closest* node that is reached from the loop

```
procedure loopType ($G = (N, E, h)$, $(y, x)$, nodesInLoop)
/* Pre: $(y, x)$ induces a loop.
 *       nodesInLoop is the set of all nodes that belong to the loop $(y, x)$.
 * Post: loopType($x$) has the type of loop induced by $(y, x)$. */

    if (nodeType($y$) == 2-way)
        if (nodeType($x$) == 2w)
            if (outEdge($x$,1) $\in$ nodesInLoop $\wedge$ outEdge($x$,2) $\in$ nodesInLoop)
                loopType($x$) = Post_Tested.
            else
                loopType($x$) = Pre_Tested.
            end if
        else
            loopType($x$) = Post_Tested.
        end if
    else /* 1-way latching node */
        if (nodeType($x$) == 2-way)
            loopType($x$) = Pre_Tested.
        else
            loopType($x$) = Endless.
        end if
    end if
end procedure
```

Figure 6-28: Algorithm to Determine the Type of Loop

after an exit is performed. The closest node is the one with the smallest reverse postorder numbering; i.e. the one that is closest to the loop (in numbering order). Any other node that is also reached from the loop can be reached from the closest node (because it must have a greater reverse postorder numbering), thus, the closest node is considered the follow node of an endless loop.

Example 9 *The loops of Figure 6-23 have the next follow nodes:*

- *Follow (loop (9,8)) = 10*
- *Follow (loop (10,6)) = 11*

Figure 6-29 gives an algorithm to determine the follow node of a loop induced by (y, x), based on the nodesInLoop set determined in the algorithm of Figure 6-27.

6.6.2 Structuring 2-way Conditionals

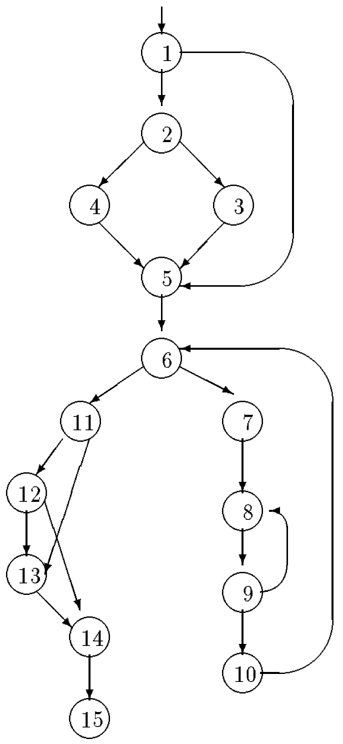
Both a single branch conditional (i.e. `if..then`) and a conditional (i.e. `if..then..else`) subgraph have a common end node, from here onwards referred to as the follow node, that has the property of being immediately dominated by the 2-way header node. Whenever

```
procedure loopFollow ($G = (N, E, h)$, $(y, x)$, nodesInLoop)
/* Pre: $(y, x)$ induces a loop.
 *      nodesInLoop is the set of all nodes that belong to the loop $(y, x)$.
 * Post: loopFollow($x$) is the follow node to the loop induced by $(y, x)$. */

    if (loopType($x$) == Pre_Tested)
        if (outEdges($x$,1) $\in$ nodesInLoop)
            loopFollow($x$) = outEdges($x$,2).
        else
            loopFollow($x$) = outEdges($x$,1).
        end if
    else if (loopType($x$) == Post_Tested)
        if (outEdges($y$,1) $\in$ nodesInLoop)
            loopFollow($x$) = outEdges($y$,2).
        else
            loopFollow($x$) = outEdges($y$,1).
        end if
    else /* endless loop */
        fol = Max /* a large constant */
        for (all 2-way nodes $n \in$ nodesInLoop)
            if ((outEdges($x$,1) $\notin$ nodesInLoop) $\wedge$ (outEdges($x$,1) $<$ fol))
                fol = outEdges($x$,1).
            else if ((outEdges($x$,2) $\notin$ nodesInLoop) $\wedge$ (outEdges($x$,2) $<$ fol))
                fol = outEdges($x$,2).
            end if
        end for
        if (fol $\neq$ Max)
            loopFollow($x$) = fol.
        end if
    end if
end procedure
```

Figure 6-29: Algorithm to Determine the Follow of a Loop

these subgraphs are nested, they can have different follow nodes or share the same common follow node. Consider the graph in Figure 6-30, which is the same graph from Figure 6-2 without intermediate instruction information, and with immediate dominator information. The nodes are numbered in reverse postorder.

In this graph there are six 2-way nodes, namely, nodes 1, 2, 6, 9, 11, and 12. As seen during loop structuring (Section 6.6.1), a 2-way node that belongs to either the header or the latching node of a loop is marked as being so, and must not be processed during 2-way conditional structuring given that it already belongs to another structure. Hence, the nodes 6 and 9 in Figure 6-30 are not considered in this analysis. Whenever two or more conditionals are nested, it is always desirable to analyze the most nested conditional first, and then the outer ones. In the case of the conditionals at nodes 1 and 2, node 2 must be



Node	Immediate Dominator
1	-
2	1
3	2
4	2
5	1
6	5
7	6
8	7
9	8
10	9
11	6
12	11
13	11
14	11
15	14

Figure 6-30: Control Flow Graph with Immediate Dominator Information

analyzed first than node 1 since it is nested in the subgraph headed by 1; in other words, the node that has a greater reverse postorder numbering needs to be analyzed first since it was last visited first in the depth first search traversal. In this example, both subgraphs share the common follow node 5; therefore, there is no node that is immediately dominated by node 2 (i.e. the inner conditional), but 5 is immediately dominated by 1 (i.e. the outer conditional), and this node is the follow node for both conditionals. Once the follow node has been determined, the type of the conditional can be known by checking whether one of the branches of the 2-way header node is the follow node, in which case, the subgraph is a single branching conditional, otherwise it is an `if..then..else`. In the case of nodes 11 and 12, node 12 is analyzed first and no follow node is determined since no node takes it as immediate dominator. This node is left in a list of unresolved nodes, because it can be nested in another conditional structure. When node 11 is analyzed, nodes 12, 13, and 14 are possible candidates for follow node, since nodes 12 and 13 reach node 14, this last node is taken as the follow (i.e. the node that encloses the most number of nodes in a subgraph, the largest node). Node 12, that is in the list of unresolved follow nodes, is also marked as having a follow node of 14. It is seen from the graph that these two conditionals are not properly nested, and a `goto` jump can be used during code generation.

A generalization of this example provides the algorithm to structure conditionals. The idea of the algorithm is to determine which nodes are header nodes of conditionals, and which nodes are the follow of such conditionals. The type of the conditional can be determined after finding the follow node by checking whether one of the branches of the header node is equivalent to the follow node. Inner conditionals are traversed first, then outer ones, so a descending reverse postorder traversal is performed (i.e. from greater to smaller node number). A set of unresolved conditional follow nodes is kept throughout the process. This set holds all 2-way header nodes for which a follow has not been found. For each 2-way node that is not part of the header or latching node of a loop, the follow node is calculated as the node that takes it as an immediate dominator and has two or more in-edges (since it must be reached by at least two different paths from the header). If there is more than one such node, the one that encloses the maximum number of nodes is selected (i.e. the one with the largest number). If such a node is not found, the 2-way header node is placed on the unresolved set. Whenever a follow node is found, all nodes that belong to the set of unresolved nodes are set to have the same follow node as the one just found (i.e. they are nested conditionals or unstructured conditionals that reach this node). The complete algorithm is shown in Figure 6-31.

```
procedure struct2Way (G=(N,E,h))
/* Pre: G is a graph.
 * Post: 2-way conditionals are marked in G.
 *       the follow node for all 2-way conditionals is determined. */

    unresolved = {}
    for (all nodes $m$ in descending order)
        if ((nodeType($m$) == 2-way) $\wedge$ (inHeadLatch($m$) == False))
            if ($\exists\, n \bullet n = \max\{i \mid \text{immedDom}(i) = m \wedge \#\text{inEdges}(i) \geq 2\}$)
                follow($m$) = $n$
                for (all $x \in$ unresolved)
                    follow($x$) = $n$
                    unresolved = unresolved - $\{x\}$
                end for
            else
                unresolved = unresolved $\cup$ $\{m\}$
            end if
        end if
    end for
end procedure
```

Figure 6-31: 2-way Conditional Structuring Algorithm

Compound Conditions

When structuring graphs in decompilation, not only the structure of the underlying constructs is to be considered, but also the underlying intermediate instructions information. Most high-level languages allow for short-circuit evaluation of compound Boolean conditions (i.e. conditions that include `and` and `or`). In these languages, the generated control flow graphs for these conditional expressions become unstructured since an exit can be performed as soon as enough conditions have been checked and determined the expression is true or false as a whole. For example, if the expression `x and y` is compiled with short-circuit evaluation, if expression `x` is false, the whole expression becomes false and therefore the expression `y` is not evaluated. In a similar way, an `x or y` expression is partially evaluated if the expression `x` is true. Figure 6-32 shows the four different subgraph sets that arise from compound conditions. The top graphs represent the logical condition that is under consideration, and the bottom graphs represent the short-circuit evaluated graphs for each compound condition.

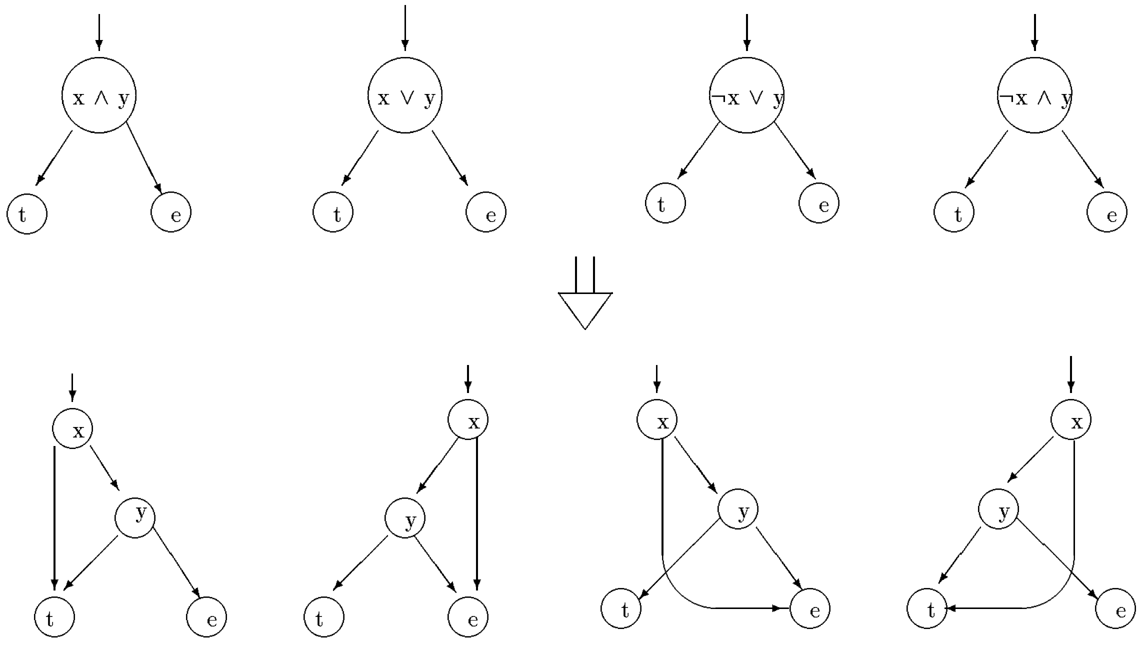


Figure 6-32: Compound Conditional Graphs

During decompilation, whenever a subgraph of the form of the short-circuit evaluated graphs is found, it is checked for the following properties:

1. Nodes `x` and `y` are 2-way nodes.
2. Node `y` has 1 in-edge.
3. Node `y` has a unique instruction, a conditional jump (`jcond`) high-level instruction.
4. Nodes `x` and `y` must branch to a common `t` or `e` node.

The first, second, and fourth properties are required in order to have an isomorphic subgraph to the bottom graphs given in Figure 6-32, and the third property is required to determine that the graph represents a compound condition, rather than an abnormal conditional graph. Consider the subgraph of Figure 6-2, in Figure 6-33 with intermediate instruction information. Nodes 11 and 12 are 2-way nodes, node 12 has 1 in-edge, node 12 has a unique instruction (a `jcond`), and both the true branch of node 11 and the false branch of node 12 reach node 13; i.e. this subgraph is of the form $\neg x \wedge y$ in Figure 6-32.

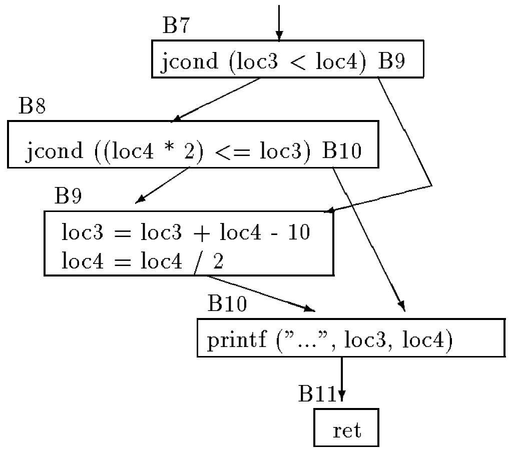


Figure 6-33: Subgraph of Figure 6-2 with Intermediate Instruction Information

The algorithm to structure compound conditionals makes use of a traversal from top to bottom of the graph, as the first condition in a compound conditional expression is higher up in the graph (i.e. it is tested first). For all 2-way nodes, the `then` and `else` nodes are checked for a 2-way condition. If either of these nodes represents one high-level conditional instruction (`jcond`), and the node has no other entries (i.e. the only in-edge to this node comes from the header 2-way node), and the node forms one of the 4 subgraphs illustrated in Figure 6-32, these two nodes are merged into a unique node that has the equivalent semantic meaning of the compound condition (i.e. depends on the structure of the subgraph), and the node is removed from the graph. This process is repeated until no more compound conditions are found (i.e. there could be 3 or more compound `and`s and `or`s, so the process is repeated with the same header node until no more conditionals are found). The final algorithm is shown in Figure 6-34.

6.6.3 Structuring n-way Conditionals

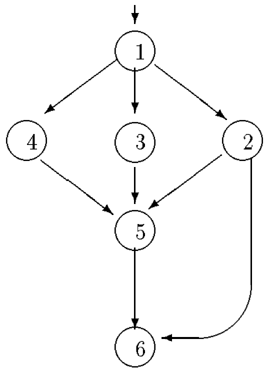
N-way conditionals are structured in a similar way to 2-way conditionals. Nodes are traversed from bottom to top of the graph in order to find nested n-way conditionals first, followed by the outer ones. For each n-way node, a follow node is determined. This node will optimally have n in-edges coming from the n successor nodes of the n-way header node, and be immediately dominated by such header node.

The determination of the follow node in an unstructured n-way conditional subgraph makes use of modified properties of the abovementioned follow node. Consider the unstructured graph in Figure 6-35, which has an abnormal exit from the n-way conditional subgraph. Candidate follow nodes are all nodes that have the header node 1 as immediate dominator, and that are not successors of this node, thus, nodes 5 and 6 are candidate follow nodes. Node 5 has 3 in-edges that come from paths from the header node, and node 6 has 2 in-edges from paths from the header node. Since node 5 has more paths from the header node that reach it, this node is considered the follow of the complete subgraph.

```
procedure structCompConds (G=(N,E,h))
/* Pre: G is a graph.
 * 2-way, n-way, and loops have been structured in G.
 * Post: compound conditionals are structured in G. */

 change = True
 while (change)
   change = False
   for (all nodes $n$ in postorder)
     if (nodeType($n$) = 2-way)
       $t$ = succ[$n$, 1]
       $e$ = succ[$n$, 2]
       if ((nodeType($t$) = 2-way) $\wedge$ (numInst($t$) = 1) $\wedge$ (numInEdges($t$) = 1))
         if (succ[$t$, 1] = $e$)
           modifyGraph ($\neg n \wedge t$)
           change = True
         else if (succ[$t$, 2] = $e$)
           modifyGraph ($n \vee t$)
           change = True
         end if
       else if ((nodeType($e$) = 2-way) $\wedge$ (numInst($e$) = 1) $\wedge$ (numInEdges($e$) = 1))
         if (succ[$e$, 1] = $t$)
           modifyGraph ($n \wedge e$)
           change = True
         else if (succ[$e$, 2] = $t$)
           modifyGraph ($\neg n \vee e$)
           change = True
         end if
       end if
     end if
   end for
 end while
end procedure
```

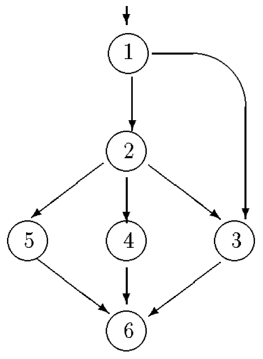
Figure 6-34: Compound Condition Structuring Algorithm

Unfortunately, abnormal entries into an n-way subgraph are not covered by the above method. Consider the graph in Figure 6-36, which has an abnormal entry into one of the branches of the header n-way node. In this case, node 6 takes node 1 as immediate dominator, due to the abnormal entry (1,2), instead of 2 (the n-way header node). In other words, the follow node takes as immediate dominator the common dominator of all in-edges to node 3; i.e. node 1. In this case, the node that performs an abnormal entry into the subgraph needs to be determined, in order to find a follow node that takes it as immediate dominator. The complete algorithm is shown in Figure 6-37.



node	immediate Dominator
1	-
2	1
3	1
4	1
5	1
6	1

Figure 6-35: Unstructured n-way Subgraph with Abnormal Exit



node	immediate Dominator
1	-
2	1
3	1
4	2
5	2
6	1

Figure 6-36: Unstructured n-way Subgraph with Abnormal Entry

6.6.4 Application Order

The structuring algorithms presented in the previous three sections determine the entry and exit (i.e. header and follow) nodes of subgraphs that represent high-level loops, n-way, and 2-way structures. These algorithms cannot be applied in a random order since they do not form a finite Church-Rosser system. Consider the graphs in Figure 6-38, which due to the abnormal entries and exits have loop subgraphs. Graph (a) has an abnormal exit from an n-way subgraph, and the complete graph belongs to the same loop. If this graph ought to be structured by loops first, the back-edge (3,1) would be found, leading to the loop {1,2,3}. By then structuring n-way conditionals, it is found that node 2 is a header node for an n-way subgraph, but since only 2 nodes of the subgraph rooted at 2 belong to the loop, it is determined that the subgraph cannot be structured as an n-way subgraph, but has several abnormal exits from the loop. On the other hand, if the graph ought to be structured by n-way subgraphs first, the subgraph {2,3,4,5,6} would be structured as an n-way subgraph with follow node 6. By then applying the loop algorithm, the nodes from the back-edge (3,1) are found to belong to different structures (i.e. node 3 belongs to a structure headed by node 2, and node 1 does not belong to any structure so far), therefore, an abnormal exit from one structure to the other exists, and the loop is not structured as such. In the case of graph (b), this graph is an irreducible graph, therefore, by first structuring it by loops,

```
procedure structNWay (G = (N,E,h))
/* Pre: G is a graph.
 * Post: n-way conditionals are structured in G.
        the follow node is determined for all n-way subgraphs. */

    unresolved = {}
    for (all nodes $m \in N$ in postorder)
        if (nodeType($m$) == n-way)
            if ($\exists$ $s$: succ($m$)$\bullet$ immedDom($s$) $\neq$ $m$)
                $n$ = commonImmedDom({$s$ | $s$ = succ($m$)})
            else
                $n = m$
            end if
            if ($\exists$ $j\bullet$ #inEdges($j$) =
                max{$i$ | immedDom($i$) = $n \wedge$ #inEdges($i$) $\geq$ 2$\bullet$ #inEdges($i$)})
                follow($m$) = $j$
                for (all $i \in$ unresolved)
                    follow($i$) = $j$
                    unresolved = unresolved - {$i$}
                end for
            else
                unresolved = unresolved $\cup$ {$m$}
            end if
        end if
    end for
end procedure
```

Figure 6-37: n-way Conditional Structuring Algorithm

a multiexit loop will be found, with abnormal exits coming from the nodes of the n-way subgraph (which is not structured as such due to the abnormal exits). On the other hand, if this graph was structured as an n-way subgraph first, the loop would not be structured as such, but as a `goto` jump.

These examples illustrate that the series of structuring algorithms presented in the previous sections is not finite Church-Rosser. This implies that an ordering is to be followed, and it is: structure n-way conditionals, followed by loop structuring, and 2-way conditional structuring last. Loops are structured first than 2-way conditionals to ensure the Boolean condition that form part of pre-tested or post-tested loops is part of the loop, rather than the header of a 2-way conditional subgraph. Once a 2-way conditional has been marked as being in the header or latching node of a loop, it is not considered for further structuring.

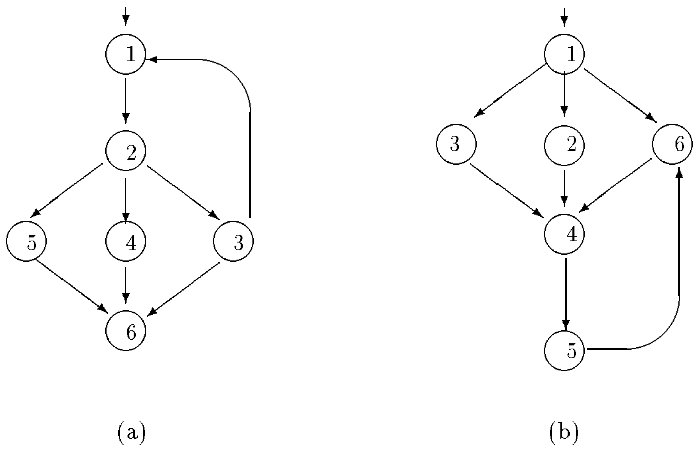


Figure 6-38: Unstructured Graph

The Case of Irreducible Graphs

The examples presented so far in this Chapter deal with reducible graphs. Recall from Section 6.3.4 that a graph is irreducible if it contains a subgraph of the form of the canonical irreducible flowgraph. In essence, a graph is irreducible if it has 2 or more entries (i.e. a multientry loop), at least 2 entries are dominated by the same common node, and this common node dominates the entrance nodes to the loop. Consider the multientry graphs in Figure 6-39. These graphs represent different classes of multientry graphs according to the underlying edges in a depth-first tree of the graph. As can be seen, graphs that have a tree-edge, cross-edge, and forward-edge are irreducible, but the graph with the back-edge coming into the loop is not irreducible since there is no common node that dominates all entries into the loop. This later loop is equivalent to an overlapping loop, much in the same way as a multiexit loop with a back-edge out of the loop (Figure 6-26, graph (d)).

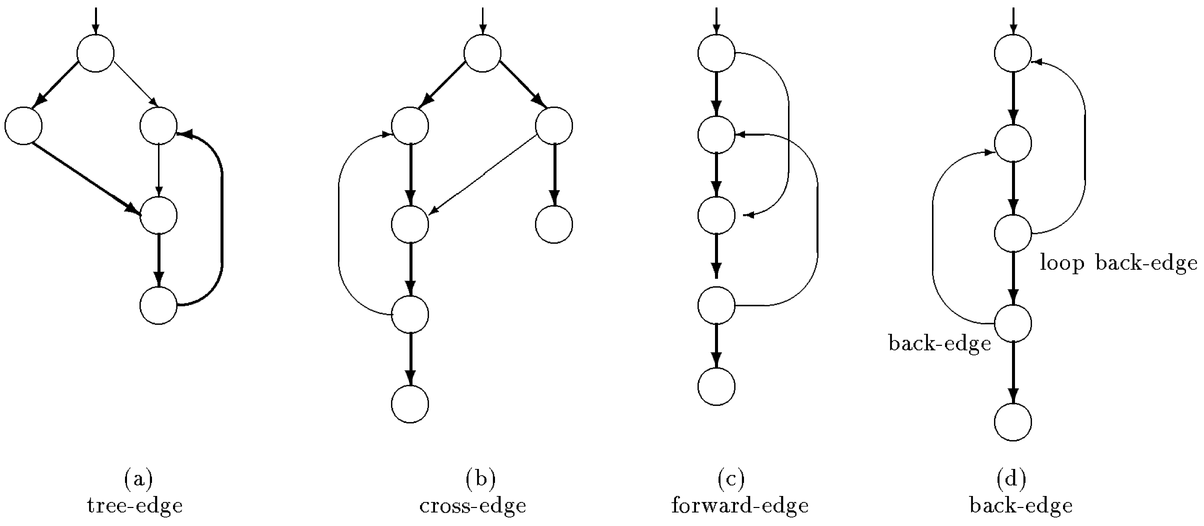
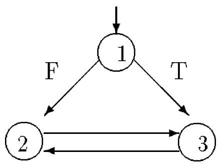


Figure 6-39: Multientry Loops - 4 Cases

Since it is the purpose of a decompiler structuring algorithm not to modify the semantics and functionality of the control flow graph, node splitting is not used to structure irreducible graphs, since the addition of new nodes modifies the semantics of the program. It is therefore desired to structure the graph without node replication, i.e. leave the graph as an irreducible graph that has `goto` jumps. Consider the graph in Figure 7-14 with immediate dominator information. Since the graph is irreducible, there is no loop that is contained entirely in an interval, therefore, the loop structuring algorithm determines that there are no natural loops as such. When structuring 2-way conditionals, the conditional at node 1 is determined to have the follow node 3, since this node is reached from both paths from the header node and has a greater numbering than node 2. This means that the graph is structured as a 2-way subgraph with follow node 3, and no natural loop. During code generation, `goto` jumps are used to simulate the loop, and the multientries (see Chapter 7, Section 7.1.3).



node	immediate dominator
1	-
2	1
3	1

Figure 6-40: Canonical Irreducible Graph with Immediate Dominator Information

Chapter 7

The Back-end

The high-level intermediate code generated by the data flow analyzer, and the structured control flow graph generated by the control flow analyzer, are the input to the back-end. This module is composed in its entirety by the code generator, which generates code for the target high-level language. This relationship is shown in Figure 7-1.

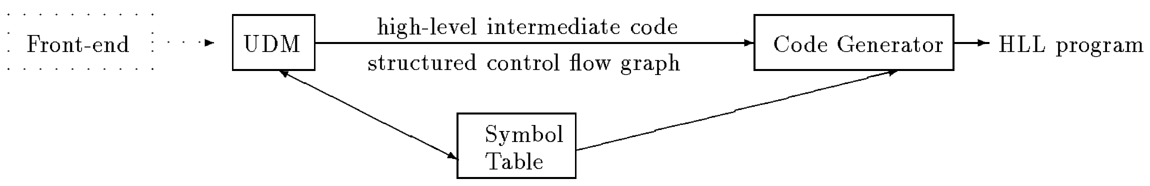


Figure 7-1: Relation of the Code Generator with the UDM

7.1 Code Generation

The code generator generates code for a predefined target high-level language. The following examples make use of the C language as target language, and the examples are based on the sample control flow graph of Chapter 6, Figure 6-2 after structuring information has been summarized on the graph.

7.1.1 Generating Code for a Basic Block

After data flow analysis, the intermediate instructions in a basic block are all high-level instructions; pseudo high-level instructions must have been eliminated from the code before this point. Consider the control flow graph in Figure 7-2 after data and control flow analyses. For each basic block, the instructions in the basic block are mapped to an equivalent instruction of the target language. Transfer of control instructions (i.e. `jcond` and `jmp` instructions) are dependent on the structure of the graph (i.e. they belong to a loop or a conditional jump (2 and n ways), or be equivalent to a `goto`), and hence, code is generated for them according to the control flow information, described in the next Section (Section 7.1.2). This section illustrates how code is generated for all other instructions of a basic block.

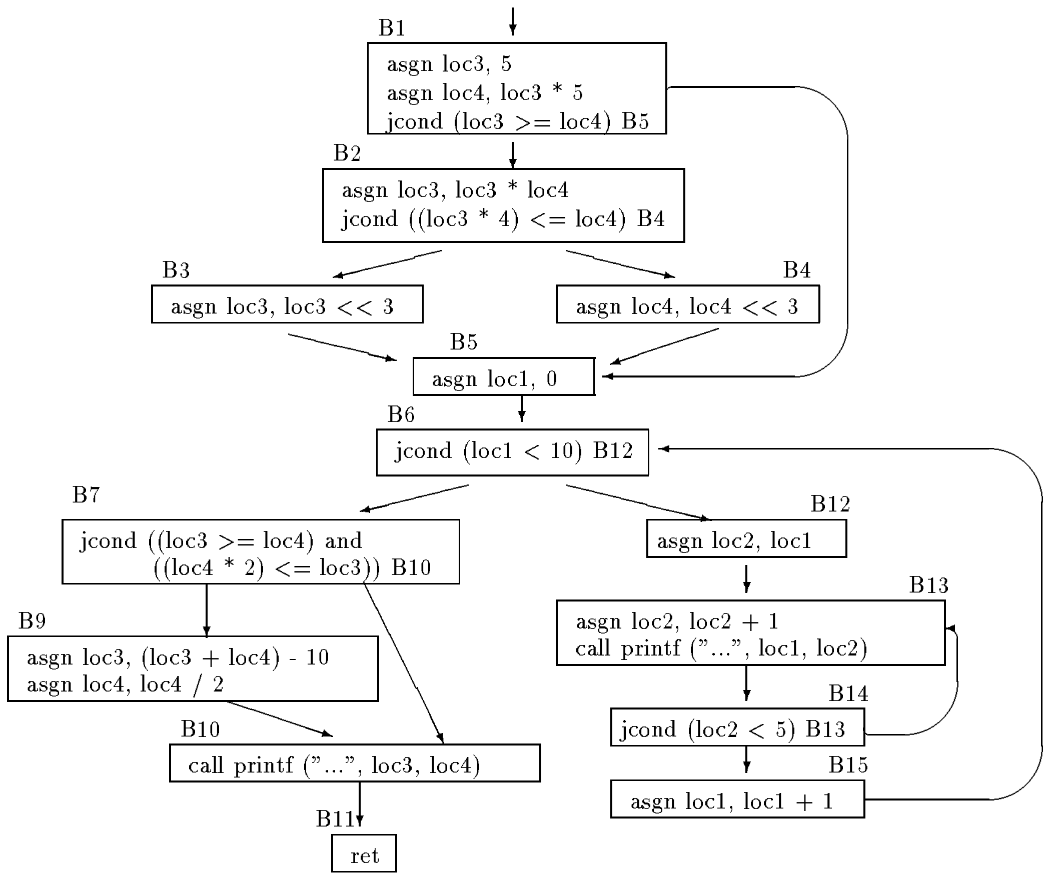


Figure 7-2: Sample Control Flow Graph After Data and Control Flow Analyses

Generating Code for asgn Instructions

The `asgn` instruction assigns to an identifier an arithmetic expression or another identifier. Expressions are stored by the decompiler in abstract syntax trees, therefore, a tree walker is used to generate code for them. Consider the first instruction of basic block B1, Figure 7-2:

```
asgn loc3, 5
```

The left hand side is the local identifier `loc3` and the right hand side is the constant identifier `5`. Since both expressions are identifiers, the code is trivially translated to:

```
loc3 = 5;
```

The first instruction of basic block B9, Figure 7-2 uses an expression in its right hand side:

```
asgn loc3, (loc3 + loc4) - 10
```

This instruction is represented by the abstract syntax tree of Figure 7-3; only the right hand side of the instruction is stored in the abstract syntax tree format (field `arg` of the intermediate triplet (see Figure 4-32, Chapter 4)). From the tree, the right hand side is equivalent to the expression `(loc3 + loc4) - 10`, and the C code for this instruction is:

```
loc3 = (loc3 + loc4) - 10;
```

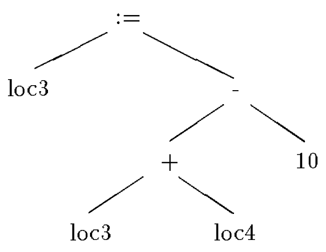


Figure 7-3: Abstract Syntax Tree for First Instruction of B9

Generating code from an abstract syntax tree is solved in a recursive way according to the type of operator; binary or unary. For binary operators, the left branch of the tree is traversed, followed by the operator, and the traversal of the right branch. For unary operators, the operator is first displayed, followed by its subtree expression. In both cases, the recursion ends when an identifier is met (i.e. the leaves of the tree).

Example 10 *Expressions are defined in an intermediate language using the following types of expressions:*

- *Binary expressions: all expressions that use a binary operator. The binary operators and their C counterparts are:*
  - *Less or equal to (*`<=`*).*
  - *Less than (*`<`*).*
  - *Equal (*`==`*).*
  - *Not equal (*`!=`*).*
  - *Greater (*`>`*).*
  - *Greater or equal to (*`>=`*).*
  - *Bitwise and (*`&`*).*
  - *Bitwise or (*`|`*).*
  - *Bitwise xor (*`^`*).*
  - *Not (1's complement) (*`~`*).*
  - *Add (*`+`*).*
  - *Subtract (*`-`*).*
  - *Multiply (*`*`*).*
  - *Divide (*`/`*).*
  - *Modulus (*`%`*).*
  - *Shift right (*`>>`*).*
  - *Shift left (*`<<`*).*
  - *Compound and (*`&&`*).*
  - *Compound or (*`||`*).*

- *Unary expressions: all expression that use a unary operator. The unary operators and their C counterparts are:*
  - *Expression negation (*`!`*).*
  - *Address of (*`&`*).*
  - *Dereference (*`*`*).*
  - *Post and pre increment (*`++`*).*
  - *Post and pre decrement (*`--`*).*
- *Identifiers: an identifier is the minimum type of expression. Identifiers are classified according to their location in memory and/or in registers, in the following way:*
  - *Global variable.*
  - *Local variable (negative offsets from the stack frame).*
  - *Formal parameter (positive offset from the stack frame).*
  - *Constant.*
  - *Register.*
  - *Function (function name and actual argument list).*

The algorithm of Figure 7-4 generates code for an expression that uses the above operator types, by walking the tree recursively.

```
procedure walkCondExp (e: expression)
/* Pre: e points to an expression tree (abstract syntax tree).
 * Post: the code for the expression tree pointed to by e is written. */

    case (expressionType(e))
        Boolean: write ("(%s %s %s)", walkCondExp (lhs(e)), operatorType(e),
                                walkCondExp (rhs(e))).
        Unary: write ("%s (%s)", operatorType(e), walkCondExp (exp(e))).
        Identifier: write ("%s", identifierName(e)).
    end case
end procedure
```

Figure 7-4: Algorithm to Generate Code from an Expression Tree

The identifierName(e) function returns the name of the identifier in the identifier node `e`; this name is taken from the appropriate symbol table (i.e. global, local or argument). Whenever the identifier is a register, the register is uniquely named by generating a new local variable; the next in the sequence of local variables. The new variable is placed at the end of the subroutine's local variables definition.

Generating Code for call Instructions

The `call` instruction invokes a procedure with the list of actual arguments. This list is stored in the `arg` field and is a sequential list of expressions (i.e. arithmetic expressions and/or identifiers). The name of the procedure is displayed followed by the actual arguments, which are displayed using the tree walker algorithm of Figure 7-4.

Generating Code for ret Instructions

The `ret` instruction returns an expression/identifier in a function. If the return instruction does not take any arguments, the procedure is finished at that statement. The return of an expression is optional.

The complete algorithm to generate code for a basic block (excluding transfer instructions) is shown in Figure 7-5. In this algorithm the function indent() is used; this function returns one or more spaces depending on the indentation level (3 spaces per indentation level).

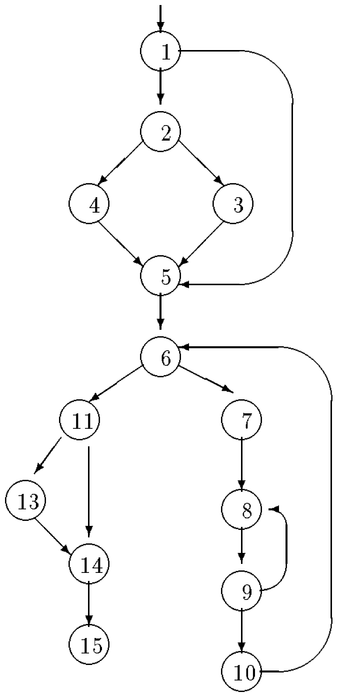
```
procedure writeBB (BB: basicBlock, indLevel: integer)
/* Pre: BB is a basic block.
 *      indLevel is the indentation level to be used in this basic block.
 * Post: the code for all instructions, except transfer instructions, is displayed. */

    for (all high-level instructions i of BB) do
        case (instType(i))
            asgn: write ("%s%s = %s;\n", indent(indLevel), walkCondExp (lhs(i)),
                        walkCondExp (rhs(i))).
            call: fa = "".
                    for (all actual arguments f ∈ formalArgList(i)) do
                        append (fa, "%s,", walkCondExp (f)).
                    end for
                    write ("%s%s (%s);\n", indent (indLevel), invokedProc (i), fa).
            ret: write ("%sreturn (%s);\n", indent(indLevel), exp(i)).
        end case
    end for
end procedure
```

Figure 7-5: Algorithm to Generate Code from a Basic Block

7.1.2 Generating Code from Control Flow Graphs

The information collected during control flow analysis of the graph is used in code generation to determine the order in which code should be generated for the graph. Consider the graph in Figure 7-6 with structuring information. This graph is the same graph of Figure 7-2 without intermediate instruction information; nodes are numbered in reverse postorder.

The generation of code from a graph can be viewed as the problem of generating code for the root node, recursing on the successor nodes that belong the structure rooted at the root



node	loopType	loopFollow	ifFollow
1			5
2			5
3			
4			
5			
6	pre_test	11	
7			
8	post_test	10	
9			
10			
11			14
13			
14			
15			

Figure 7-6: Control Flow Graph with Structuring Information

node (if any), and continue code generation with the follow node of the structure. Recall from Chapter 6 that the follow node is the first node that is reached from a structure (i.e. the first node that is executed once the structure is finished). Follow nodes for loops, 2-way and n-way conditionals are calculated during the control flow analysis phase. Other transfer of control nodes (i.e. 1-way, fall-through, call) transfer control to the unique successor node; hence the follow is the successor, and termination nodes (i.e. return) are leaves in the underlying depth-first search tree of the graph, and hence terminate the generation of code along that path.

This section describes the component algorithms of the algorithm to generate code for a procedure, `writeCode()`. To make the explanation easier, we will assume that this routine exists; therefore, we concentrate only on the generation of code for a particular structure and let the `writeCode()` routine generate code for the components of the structure. After enough algorithms have been explained, the algorithm for `writeCode()` is given.

Generating Code for Loops

Given a subgraph rooted at a loop header node, code for this loop is generated based on the type of loop. Regardless of type of loop, all loops have the same structure: loop header, loop body, and loop trailer. Both the loop header and trailer are generated depending on the type of loop, and the loop body is generated by generating code for the subgraph rooted at the first node of the loop body. Consider the loops in the graph of Figure 7-6. The loop rooted at node 6 is a pre-tested loop, and the loop rooted at node 8 is a post-tested loop.

In the case of the pre-tested loop, when the loop condition is True (i.e. the `jcond` Boolean conditional in node 6), the loop body is executed. If the branch into the loop was the False branch, the loop condition has to be negated since the loop is executed when the condition is False. The loop body is generated by the `writeCode()` routine, and the loop trailer consists only of an end of loop bracket (in C). Once this code has been generated, code for the loop follow node is generated by invoking the `writeCode()` routine. The following skeleton is used:

```
write ("%s while (loc1 < 10) {\n", indent(indLevel))
writeCode (7, indLevel + 1, 10, ifFollow, nFollow)
write ("%s }\n", indent(indLevel))
writeCode (11, indLevel, latchNode, ifFollow, nFollow)
```

where the first instruction generates code for the loop header, the second instruction generates code for the loop body; rooted at node 7 and having a latching node 10, the third instruction generates code for the loop trailer, and the fourth instruction generates code for the rest of the graph rooted at node 11.

In the post-tested loop, the loop condition is true when the branch is made to the loop header node. The following skeleton is used:

```
write ("%s do {\n", indent(indLevel))
writeBB (8, indLevel + 1)
writeCode (9, indLevel + 1, 9, ifFollow, nFollow)
write ("%s } while (loc2 < 5); \n", indent(indLevel))
writeCode (10, indLevel, latchNode, ifFollow, nFollow)
```

where the first instruction generates code for the loop header, the second instruction generates code for the instruction in the root node, the third instruction generates code for the loop body rooted at node 9 and ended at the loop latching node 9, the fourth instruction generates the loop trailer, and the fifth instruction generates code for the remainder of the graph rooted at node 10. Code is generated in a similar way for endless loops, with the distinction that there may or may not be a loop follow node.

Normally pre-tested loop header nodes have only one instruction associated with them, but in languages that allow for several logical instructions to be coded in the one physical instruction, such as in C, these instructions will be in the header node but not all of them would form part of the loop condition. For example, in the following C loop:

```
while ((a += 11) > 50)
{
   printf ("greater than 50\n");
   a = a - b;
}
```

the `while()` statement has two purposes: to add `11` to variable `a`, and to check that after this assignment `a` is greater than `50`. Since our choice of intermediate code allows for only one instruction to be stored in an intermediate instruction, the assignment and the comparison form part of two different instructions, as shown in the following intermediate code:

```
B3:
  asgn a, a + 11
  jcond (a <= 50) B5
B4:
  call printf ("greater than 50\n")
  asgn a, a - b
  jmp B3
B5:
  /* other code */
```

Two solutions are considered for this case: preserve the `while()` loop structure by repeating the extra instructions in the header basic block at the end of the loop, or transform the `while()` loop into an endless `for (;;)` loop that `break`s out of the loop whenever the Boolean condition associated with the `while()` is False. In our example, the former case leads to the following code in C:

```
  a = a + 11;
  while (a > 50) {
     printf ("greater than 50\n");
     a = a - b;
     a = a + 11;
  }
```

and the latter case leads to the following C code:

```
  for (;;) {
     a = a + 11;
     if (a <= 50)
        break;
     printf ("greater than 50\n");
     a = a - b;
  }
```

Either approach generates correct code for the graph; the former method replicates code (normally a few instructions, if any) and preserves the `while()` structure, the latter method does not replicate code but modifies the structure of the original loop. In this thesis the former method is used in preference to the latter, since this solution provides code that is easier to understand than the latter solution.

When generating code for the loop body or the loop follow node, if the target node has already been traversed by the code generator, it means that the node has already been reached along another path, therefore, a `goto` label needs to be generated to transfer control to the target code. The algorithm in Figure 7-7 generates code for a graph rooted at a loop header node. This algorithm generates code in C, and assumes the existence of the function `invExp()` which returns the inverse of an expression (i.e. negates the expression), and the procedure `emitGotoLabel()` which generates a unique label, generates a `goto` to that label, and places the label at the appropriate position in the final C code.

```
procedure writeLoop (BB: basicBlock; i, latchNode, ifFollow, nFollow: Integer)
/* Pre: BB is a pointer to the header basic block of a loop.
 *      i is the indentation level used for this basic block.
 *      latchNode is the number of the latching node of the enclosing loop (if any).
 *      ifFollow is the number of the follow node of the enclosing if structure (if any).
 *      nFollow is the number of the follow node of the enclosing n-way structure (if any).
 * Post: code for the graph rooted at BB is generated. */

  traversedNode(BB) = True.
  case (loopType(BB))                              /* Write loop header */
    Pre_Tested:
      writeBB (BB, i).
      if (succ (BB, Else) == loopFollow(BB)) then
        write ("%s while (%s) {\n", indent(i), walkCondExp (loopExp(BB))).
      else
        write ("%s while (%s) { \n", indent(i), walkCondExp (invExp(loopExp(BB)))).
      end if
    Post_Tested: write ("%s do\n {", indent(i)).
        writeBB (BB, i+1).
    Endless: write ("%s for (;;) { \n", indent(i)).
        writeBB (BB, i+1).
  end case
  if ((nodeType(BB) == Return) ∨ (revPostorder(BB) == latchNode)) then return.
  if (latchNode(BB) ≠ BB) then                     /* Loop is several basic blocks */
    for (all successors s of BB) do
      if (loopType(BB) ≠ Pre_Tested) ∨ (s ≠ loopFollow(BB)) then
        if (traversedNode(BB) == False) then
          writeCode (s, i+1, latchNode (BB), ifFollow, nFollow).
        else /* has been traversed */
          emitGotoLabel (firstInst(s)).
        end if
      end if
    end for
  end if
  case (loopType(BB))                              /* Write loop trailer */
    Pre_Tested: writeBB (BB, i+1).
        write ("%s }\n", indent(i)).
    Post_Tested: write ("%s } while (%s); \n", indent(i), walkCondExp (loopExp(BB))).
    Endless: write ("%s } \n", indent(i)).
  end case
  if (traversedNode(loopFollow(BB)) == False) then /* Continue with follow */
    writeCode (loopFollow(BB), i, latchNode, ifFollow, nFollow).
  else
    emitGotoLabel (firstInst(loopFollow(BB))).
  end if
end procedure
```

Figure 7-7: Algorithm to Generate Code for a Loop Header Rooted Graph

Generating Code for 2-way Rooted Graphs

Given a graph rooted at a 2-way node that does not form part of a loop conditional expression, code for this graph is generated by determining whether the node is the header of an `if..then` or an `if..then..else` condition. In the former case, code is generated for the condition of the `if`, followed by the code for the `then` clause, and finalized with the code for the if follow subgraph. In the latter case, code is generated for the `if` condition, followed by the `then` and `else` clauses, and finalized with the code for the follow node. Consider the two 2-way nodes in Figure 7-6 which do not form part of loop expressions; nodes 1, 2 and 11.

Node 1 is the root of an `if..then` structure since the follow node (node 5) is one of the immediate successors of node 1. The other immediate successor, node 2, is the body of the `then` clause, which is reached when the condition in node 1 is False; i.e. the condition needs to be negated, as in the following code:

```
write ("%s if (loc3 < loc4) {\n", indent(indLevel))
writeCode (2, indLevel+1, latchNode, 5, nFollow)
write ("%s }\n", indent(indLevel))
writeCode (5, indLevel, latchNode, ifFollow, nFollow)
```

where the first instruction generates code for the negated condition of the `if`, the second instruction generates code for the `then` clause subgraph which is rooted at node 2 and has 5 as a follow node, the third instruction generates the trailer of the `if`, and the last instruction generates code for the follow subgraph rooted at node 5.

Node 2 is the root of an `if..then..else` structure. In this case, neither immediate successors of the header node are equivalent to the follow node. The True branch is reached when the condition is True, and the False branch is reached when the condition is False, leading to the following code:

```
write ("%s if ((loc3 * 4) <= loc4) {\n", indent(indLevel))
writeCode (3, indLevel+1, latchNode, 5, nFollow)
write ("%s }\n else {\n", indent(indLevel))
writeCode (4, indLevel+1, latchNode, 5, nFollow)
write ("%s }\n", indent(indLevel))
```

where the first instruction generates code for the `if` condition, the second instruction generates code for the `then` clause, the third instruction generates the `else`, the fourth instruction generates code for the `else` clause, and the last instruction generates the `if` trailer. Code for the follow node is not generated in this case because this conditional is nested in another conditional that also takes 5 as the follow node. This is easily checked with the `ifFollow` parameter, which specifies the follow of the enclosing `if`, if it is the same, code for this node is not yet generated.

In a similar way, code is generated for the subgraph rooted at node 11. In this case, the True branch leads to the follow node, hence, the Boolean condition associated with this `if` has to be negated, and the False branch becomes the `then` clause. The following skeletal code is used:

```
write ("%s if ((loc3<loc4) || ((loc4*2)>loc3)) {\n", indent(indLevel))
writeCode (13, indLevel+1, latchNode, 14, nFollow)
write ("%s }\n", indent(indLevel))
writeCode (14, indLevel, latchNode, ifFollow, nFollow)
```

As with loops, `goto` jumps are generated when certain nodes in the graph have been visited before the current subgraph visits them. In this case, whenever the branches of a 2-way node have already been visited, a `goto` to such branch(es) is generated. Also, whenever a 2-way rooted subgraph does not have a follow node it means that the two branches of the graph do not lead to a common node because the branches are ended (i.e. a return node is met) before met. In this case, code is generated for both branches, and the end of the path will ensure that the recursion is ended. The algorithm in Figure 7-8 generates code for a graph rooted at a 2-way node that does not form part of a loop Boolean expression.

Generating Code for n-way Rooted Graphs

Given a graph rooted at an n-way node, code for this graph is generated in the following way: the n-way header code is emitted (a `switch()` is used in C), and for each successor of the header node the n-way option is emitted (a `case` is used in C), followed by the generation of code of the subgraph rooted at that successor and ended at the n-way follow node. Once the code for all successors has been generated, the n-way trailer is generated, and code is generated for the rest of the graph by generating code for the graph rooted at the follow node of the n-way header node. Whenever generating code for one of the branches or the follow node of the n-way structure, if the target node has already been traversed, a `goto` jump is generated to transfer control to the code associated with that node.

The algorithm in Figure 7-9 generates code for a graph rooted at an n-way node.

Generating Code for 1-way, Fall, and Call Rooted Graphs

Given a graph rooted at a 1-way, fall-through, or call node, the code for the basic block is generated, followed by the unique successor of such node. Even though call nodes have 2 successors, one of the successor edges points to the subroutine invoked by this instruction; since code is generated on a subroutine at a time basis, this branch is disregarded for code generation purposes, and the node is thought of as having a unique successor.

The algorithm in Figure 7-10 generates code for nodes that have a unique successor node. If code has already been generated for the unique follow node, it means that the graph was reached along another path and hence a `goto` jump is generated to transfer control to the code associated with that subgraph.

A Complete Algorithm

The final algorithm to generate C code from a subroutine's graph is shown in Figure 7-11. The `writeCode()` procedure takes as arguments a pointer to a basic block, the indentation level to be used, the latching node of an enclosing loop (if any), and the follow nodes of enclosing 2-way and n-way conditionals (if any). Initially, the basic block pointer points to the start of the subroutine's graph, the indentation level is 1, and there are no latching or

```
procedure write2way (BB: basicBlock; i, latchNode, ifFollow, nFollow: Integer)
/* Pre: BB is a 2-way basic block.
 *      i is the indentation level.
 *      latchNode is the latching node of the enclosing loop (if any).
 *      ifFollow is the follow node of the enclosing 2-way structure (if any).
 *      nFollow is the number of the follow node of the enclosing n-way structure (if any).
 * Post: the code for the tree rooted at BB is generated. */

  if (ifFollow(BB) ≠ MAX) then
    emptyThen = False.
    if (traversedNode(succ(BB,Then)) == False) then        /* Process then clause */
      if (succ(BB,Then) ≠ ifFollow(BB)) then
        write ("\n %s if (%s) \n {", indent(i+1), walkCondExp (ifExp(BB)).
        writeCode (succ(BB,Then), i+1, latchNode, ifFollow(BB), nFollow).
      else /* empty then clause; negate else clause */
        write ("\n %s if (%s) \n {", indent(i+1), walkCondExp (invExp(ifExp(BB))).
        writeCode (succ(BB,Else), i+1, latchNode, ifFollow(BB), nFollow).
        emptyThen = True.
      end if
    else
      emitGotoLabel (firstInst(succ(BB,Then))).
    end if
    if (traversedNode(succ(BB,Else)) == False) then        /* Process else clause */
      if (succ(BB,Else) ≠ ifFollow(BB)) then
        write ("%s } \n %s else \n {", indent(i), indent(i)).
        writeCode (succ(BB,Else), i+1, latchNode, ifFollow(BB), nFollow).
      end if
    else if (emptyThen == False)
      write ("%s } \n %s else \n {", indent(i), indent(i)).
      emitGotoLabel (firstInst(succ(BB,Else))).
    end if
    write ("%s } \n", indent(i)).
    if (traversedNode(ifFollow(BB)) == False)) then
      writeCode (ifFollow(BB), i, latchNode, ifFollow, nFollow).
    end if
  else    /* No follow, emit if..then..else */
      write ("\n %s if (%s) { \n", indent(i), walkCondExp(ifExp(BB)).
    writeCode (succ(BB,Then), i, latchNode, ifFollow, nFollow).
    write ("%s } \n %s else \n {", indent(i), indent(i)).
    writeCode (succ(BB,Else), i, latchNode, ifFollow, nFollow).
    write ("%s } \n", indent(i)).
  end if
end procedure
```

Figure 7-8: Algorithm to Generate Code for a 2-way Rooted Graph

```
procedure writeNway (BB: basicBlock; i, latchNode, ifFollow, nFollow: Integer)
/* Pre: BB is an n-way basic block.
 *      i is the indentation level.
 *      latchNode is the number of the enclosing loop latching node (if any).
 *      ifFollow is the number of the enclosing if terminating node (if any).
 *      nFollow is the number of the enclosing n-way terminating node (if any).
 * Post: code is generated for the graph rooted at BB. */

    write ("%s switch (%s) { \n", indent(i), nwayExp(BB)).
    for (all successors s of BB) do                /* Generate Code for each Branch */
        if (traversedNode(s) == False) then
            write ("%s case %s: \n", indent(i+1), index(s)).
            writeCode (s, i+2, latchNode, ifFollow, nwayFollow(BB)).
            write ("%s break; \n", indent(i+2)).
        else
            emitGotoLabel (firstInst(s)).
        end if
    end for
    if (traversedNode(nwayFollow(BB)) == False) /* Generate code for the follow node */
        writeCode (nwayFollow(BB), i, latchNode, ifFollow, nFollow).
    else
        emitGotoLabel (firstInst(nwayFollow(BB))).
    end if
end procedure
```

Figure 7-9: Algorithm to Generate Code for an n-way Rooted Graph

follow nodes to check upon (these values are set to a predetermined value). Whenever a follow node is met, no more code is generated along that path, and the procedure returns to the invoking procedure which is able to handle the code generation of the follow node. This is done so that the trailer of a conditional is generated before the code that follows the conditional. In the case of loops, the latching node is the last node for which code is generated along a path, after which recursion is ended and the invoked procedure handles the loop trailer code generation and the continuation of the follow of the loop.

The procedure order in which code is generated is determined by the call graph of the program. We like to generate code for the nested procedures first, followed by the ones that invoke them; hence, a depth-first search ordering on the call graph is followed, marking each subroutine graph as being traversed once it has been considered for code generation. C code for each subroutine is written by generating code for the header of the subroutine, followed by the local variables definition, and the body of the subroutine. The algorithm in Figure 7-12 shows the ordering used, and the generation of code for the subroutine. The isLib() function is used in this algorithm to determine whether a subroutine is a library or not; code is not generated for library routines that were detected by the signature method of Chapter 8. The writeComments() procedure writes information collected from the analysis

```
procedure write1way (BB: basicBlock; i, latchNode, ifFollow, nFollow: Integer)
/* Pre: BB is a pointer to a 1-way, call, or fall-through basic block.
 *      i is the indentation level used for this basic block.
 *      latchNode is the number of the latching node of the enclosing loop (if any).
 *      ifFollow is the number of the follow node of the enclosing 2-way structure (if any).
 *      nFollow is the number of the follow node of the enclosing n-way structure (if any).
 * Post: code for the graph rooted at BB is generated. */

  writeBB (BB, i).
  if (traversedNode(succ(BB,1)) == False) then
    writeCode (succ(BB,1), i, latchNode, ifFollow, nFollow).
  else
    emitGotoLabel (firstInst(succ(BB,1))).
  end if
end procedure
```

Figure 7-10: Algorithm to Generate Code for 1-way, Call, and Fall Rooted Graphs

```
procedure writeCode (BB: basicBlock; i, latchNode, ifFollow, nFollow: Integer)
/* Pre: BB is a pointer to a basic block. Initially it points to the head of the graph.
 *      i is the indentation level used for this basic block.
 *      latchNode is the number of the latching node of the enclosing loop (if any).
 *      ifFollow is the number of the follow node of the enclosing 2-way structure (if any).
 *      nFollow is the number of the follow node of the enclosing n-way structure (if any).
 * Post: code for the graph rooted at BB is generated. */

  if ((revPostorder(BB) == (ifFollow ∨ nFollow)) ∨ (traversedNode(BB) == True)) then
    return.
  end if
  traversedNode(BB) = True.
  if (isLoopHeader(BB)) then                       /* ... for loops */
    writeLoop (BB, i, latchNode, ifFollow).
  else                                             /* ... for other nodes */
    case (nodeType(BB))
      2-way: write2way (BB, i, latchNode, ifFollow, nFollow).
      n-way: writeNway (BB, i, latchNode, ifFollow, nFollow).
      default: write1way (BB, i, latchNode, ifFollow, nFollow).
    end case
  end if
end procedure
```

Figure 7-11: Algorithm to Generate Code from a Control Flow Graph

of the subroutine, such as the type of arguments that were used (stack arguments or register arguments), whether a high-level prologue was detected in the subroutine, the number of arguments the subroutine takes, whether the subroutine generates an irreducible graph or not, and many more.

```
procedure writeProc (p: procedure)
/* Pre: p is a procedure pointer; initially the start node of the call graph.
 * Post: C code is written for the program rooted at p in a depth-first fashion. */

    if (traversedProc(p) $\vee$ isLib(p)) then
        return.
    end if
    traversedProc(p) = True.
    for (all successors s $\in$ succ(p)) do               /* Dfs on Successors */
        writeProc (s).
    end for

    /* Generate code for this procedure */
    if (isFunction(p)) then                               /* Generate Subroutine Header */
        write ("%s %s (%s) \n {", returnType(p), funcName(p), formalArgList(p)).
    else
        write ("void %s (%s) \n {", procName(p), formalArgList(p)).
    end if
    writeComments(p).                                     /* Generate Subroutine Comments */
    for (all local variables v $\in$ localStkFrame(p)) do  /* Local Variable Definitions */
        write ("%s %s;\n", varType(v), genUniqueName(v)).
    end for
    if (isHighLevel(p)) then                              /* Generate Code for Subroutine */
        writeCode (controlFlowGraph(p), 1, Max, Max, Max).
    else     /* low-level subroutine, generate assembler */
        disassemble(p).
    end if
    write ("}\n").
end procedure
```

Figure 7-12: Algorithm to Generate Code from a Call Graph

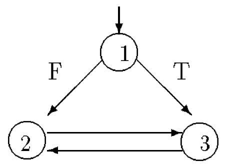
Using the algorithms described in this section, the C code in Figure 7-13 is generated for the graph of Figure 7-2. Local variables are uniquely named in a sequential order starting from one, and making use of the prefix `loc` (for local).

```
void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
int loc3;
int loc4;

    loc3 = 5;
    loc4 = (loc3 * 5);
    if (loc3 < loc4) {
        loc3 = (loc3 * loc4);
        if ((loc3 << 2) > loc4) {
            loc3 = (loc3 << 3);
        }
        else {
            loc4 = (loc4 << 3);
        }
    }
    loc1 = 0;
    while ((loc1 < 10)) {
        loc2 = loc1;
        do {
            loc2 = (loc2 + 1);
            printf ("i = %d, j = %d\n", loc1, loc2);
        } while ((loc2 < 5));
        loc1 = (loc1 + 1);
    }
    if ((loc3 >= loc4) && ((loc4 << 1) <= loc3)) {
        loc3 = ((loc3 + loc4) - 10);
        loc4 = (loc4 / 2);
    }
    printf ("a = %d, b = %d\n", loc3, loc4);
}
```

Figure 7-13: Final Code for the Graph of Figure 7.2

7.1.3 The Case of Irreducible Graphs

As pointed out in Chapter 6, Section 6.6.4, loops of irreducible graphs are not structured as natural loops since the nodes of the loop do not form part of a complete interval. Consider the canonical irreducible flow graph of Figure 7-14 with structuring information.



node	ifFollow
1	3
2	
3	

Figure 7-14: Canonical Irreducible Graph with Structuring Information

During code generation, code for the `if` in node 1 is generated first, since the True branch leads to the follow node of the `if` (node 3), the False branch of node 1 is the `then` clause (negating the Boolean condition associated with node 1). Code for the `then` clause is generated, and the `if..then` is followed by the code generated for node 3. Since node 3 transfers control to node 2, which has already been visited during code generation, a `goto` jump is generated to the associated code of node 2. This `goto` simulates the loop and provides an unstructured `if..then` structure by transferring control to the `then` clause of the `if`. The skeletal code for this graph in C is as follows:

```
  if (! 1) {
L:   2;
  }
  3;
  goto L;
```

where the numbers represent the basic blocks and code for each basic block is generated by the `writeBB()` procedure.

Chapter 8

Decompilation Tools

The decompiler tools are a series of programs that help the decompiler generate target high-level language programs. Given a binary file, the **loader** determines where the binary image starts, and whether there is any relocation information for the file. Once the binary image has been loaded into memory (and possibly relocated), a **disassembler** can be used to parse the binary image and produce an assembler form of the program. The parsing process can benefit from the use of a compiler and library signature recognizer, which determines whether a subroutine is a library subroutine or not, according to a set of predefined signatures generated by another program. In this way, only the original code written by the user is disassembled and decompiled. The disassembler can be considered part of the decompiler, as it parses only the binary image (i.e. it is a phase of the front-end module). Once the program has been parsed, it can be decompiled using the methods of Chapters 4,5, 6, and 7, generating the target high-level program. Finally, a **postprocessor** program can improve the quality of the high-level code. Figure 8-1 shows the different stages involved in a decompilation system.

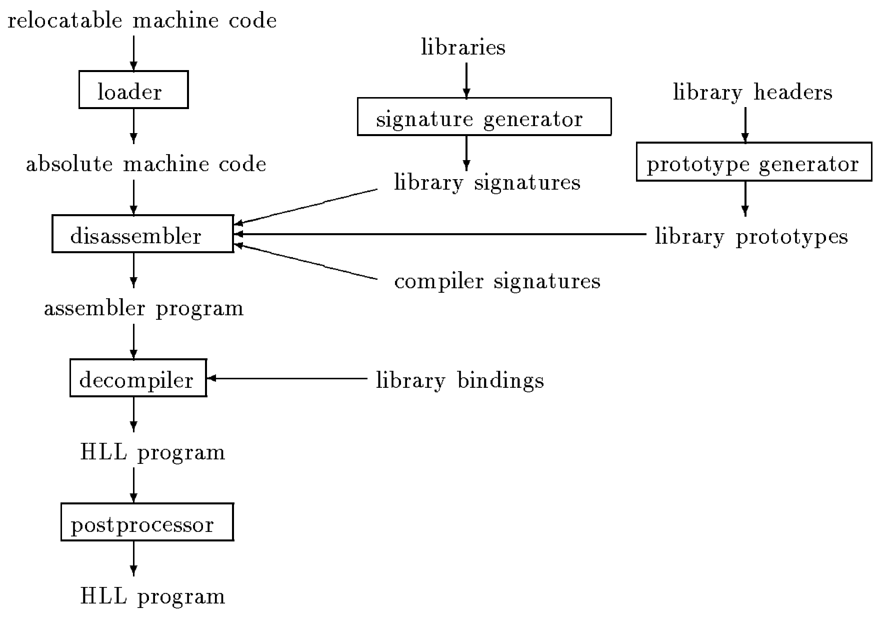


Figure 8-1: Decompilation System

8.1 The Loader

The **loader** is an operating system program that loads an executable or binary program into memory if there is sufficient free memory for the program to be loaded and run. Most binary programs contain information on the amount of memory that is required to run the program, relocation addresses, and initial segment register values. Once the program is loaded into memory, the loader transfers control to the binary program by setting up the code and instruction segments.

The structure of binary programs differs from one operating system to another, therefore, the loading of a program is dependent on the operating system and the machine the binary program runs in. The simplest form of binary programs contains only the binary image of the program, that is, a fully linked image of the program that is loaded into memory as is, without any changes made to the binary image. The **.com** files under the DOS operating system use this binary structure. Most binary programs contain not only the binary image, but also header information to determine the type of binary program (i.e. there can be different types of executable programs for the same operating system, or for different operating systems that run on the same machine) and initial register values, and a relocation table that holds word offsets from the start of the binary image which need to be relocated according to the address were the program is loaded into memory. This type of binary file is used in the **.exe** files under DOS and Windows. The general format of a binary program is shown in Figure 8-2.

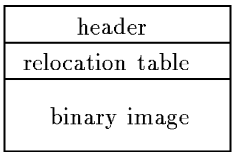


Figure 8-2: General Format of a Binary Program

The algorithm used to load a program into memory is as follows: the type of binary file is determined (on systems that allow for different types of binary files), if the file is a binary image on its own, the size of memory to be allocated is the size of the file, therefore, a block of memory of the size of the file is allocated, the file is loaded into the block of memory as is, without any modifications, and the default segment registers are set. In the case of binary files with header and relocation table information, the header is read to determine how much memory is needed to load the program, were the relocation table is, and to get other information to set up registers. A memory block of the size given in the header is allocated, the binary image of the file is then loaded into the memory block, the elements in the relocation table are relocated in memory, and segment registers are set up according to the information in the header. The algorithm is shown in Figure 8-3.

```
procedure loader (name: fileName)
/* Pre: name is the name of a binary file.
 * Post: the binary program name has been loaded into memory. */

    determine type of binary program.
    if (only binary image) then
        S = size of the binary file.
        allocate free block of memory of size S.
        load file into allocated memory block, at a predefined offset.
        setup default segment registers.
    else
        read header information.
        S = size of the binary image (from the header information).
        allocate free block of memory of size S.
        load binary image into allocated memory block.
        relocate all items from the relocation table in memory.
        setup segment registers with information from the header.
    end if
end procedure
```

Figure 8-3: Loader Algorithm

8.2 Signature Generator

A signature generator is a program that automatically generates signatures for an input file. A **signature** is a binary pattern used to recognize viruses, compilers, and library subroutines. The aim of signatures in decompilation is to undo the process performed by the linker, that is, to determine which subroutines are libraries and compiler start-up code, and replace them by their name (in the former case) or eliminate them from the target output code (in the latter case). This is the case for operating systems that do not share libraries, and therefore bind the library subroutine's object code into the program's binary image. No information on the subroutine's name or arguments is stored in the binary program, hence, without a method to distinguish them from user-written subroutines, it is impossible to differentiate them from other subroutines. In the case of operating systems that share library subroutines, the subroutine does not form part of the binary program, and a reference to the subroutine is made in the program, hence, the subroutine's name is stored as part of the binary file (most likely in the header section). The methods presented in this section are targetted at operating systems that do not share library subroutines, and therefore include them in the binary program.

Once a signature file has been generated for a set of library subroutines, an interface procedure is called to check a particular subroutine that is to be parsed by the decompiler/disassembler against the library signatures. If a subroutine is matched against one of the signatures, the subroutine is replaced by its name (i.e. the name of the subroutine in the library, such as `printf`) and is marked as not needing any more analysis. In this way the number of subroutines to be analyzed is reduced, but even better, the quality of the target high-level program is improved considerably since some subroutine calls will make use of real library names rather than arbitrary names. Also, since some of the library subroutines are written in assembler for performance reasons or due to low-level machine accesses, these routines do not have a high-level representation in most cases, and thus, can only be disassembled as opposed to decompiled; the use of a library signature recognition method eliminates the need to analyze this type of subroutines, producing better target code.

The ideas presented in this and the next section (Sections 8.2, and 8.3) were developed by Michael Van Emmerik while working at the Queensland University of Technology. These ideas are expressed in [Emm94]. Figure 8-4 has been reproduced with permission from the author.

8.2.1 Library Subroutine Signatures

A standard library file is a relocatable object file that implements different subroutines available in a particular language/compiler. A library subroutine signature is a binary pattern that uniquely identifies a subroutine in the library from any other subroutine in that same library. Since all subroutines perform different functions, a signature that contains the complete binary pattern of the subroutine will uniquely identify the subroutine from any other subroutine. The main problem with this approach is the size of the signature and the overhead created by that size. It is therefore ideal to check only a minimum number of bytes in the subroutine, hence, the signature is as small as possible. Given the great number of subroutines in a library, it is not hard to realize that for some subroutines there is need for n bytes in the signature to uniquely identify them, but n could be greater than the complete size of small subroutines; therefore, for small subroutines, the remaining bytes need to be padded with a predetermined value in order to avoid running into bytes that belong to another library subroutine. For example, if n is 23, the library function `cos` has 10 bytes, and `cos` is followed by the library function `strcpy`; the 10 bytes of `cos` form part of its signature, along with 13 bytes of padded predetermined value; otherwise, the 13 bytes would be part of `strcpy`.

Given the first n bytes of a subroutine, machine instructions that use operands which cannot be determined to be constants or offsets, or that depend on the address where the module was loaded, are considered variant bytes that can have a different value in the library file and the binary program that contains such a subroutine. It is therefore necessary to wildcard these variant byte locations, in order to generate an address-independent signature. Consider the code for a library routine `fseek()` in Figure 8-4. The `call` at instruction 108 has an offset to the subroutine that is being called. Called subroutines are not always linked at the same position, therefore, this offset address is variant; thus it is wildcarded. The `mov` at instruction 110 takes as one of its arguments a constant or offset operand; since it is not known whether this location is invariant (i.e. a constant), it is wildcarded as well. The choice of wildcard value is dependent on the machine assembler. A good candidate is a byte that is hardly used in the machine, such as `halt` in this example (opcode `F4`), or a byte that is not used in the assembler of the machine. Similar considerations are done for the padding bytes used when the signature is too small. In this example, 00 was used.

```
0100  55       push  bp
0101  8BEC     mov   bp, sp
0103  56       push  si
0104  8B7604   mov   si, [bp+4]
0107  56       push  si
0108  E8F4F4   call  ****        ; destination wildcarded
010B  59       pop   cx
010C  0BC0     or    ax, ax
010E  7405     jz    0115
0110  B8F4F4   mov   ax, ****    ; operand wildcarded
0113  EB4C     jmp   0161
0115  0000     ; padding
```

Figure 8-4: Partial Disassembly of Library Function fseek()

It is noted in this example that although the function `fseek()` has more bytes in its image, the signature is cut after 21 bytes due to the unconditional jump in instruction 113. This is done since it is unknown whether the bytes that follow the unconditional jump form part of the same library subroutine or not. In general, whenever a return or (un)conditional jump is met, the subroutine is considered finished for the purposes of library signatures, and any remaining bytes are padded. The final signature for this example is given in Figure 8-5. It should be noted that this method has some small probability of being in error since different subroutines may have the same starting code up to the first (un)conditional transfer of control.

```
558BEC568B760456E8F4F4590BC07405B8F4F4EB4C0000
```

Figure 8-5: Signature for Library Function fseek()

The algorithm to automatically generate library subroutine signatures is shown in Figure 8-6. This algorithm takes as arguments a standard library file, the name of the output signature file, and the size of the signature (in bytes), which has been experimentally found in advance.

Since different library files are provided by the compiler vendor in machines that use different memory models, a different signature file needs to be generated for each memory model. It is ideal to use a naming convention to determine the compiler vendor and memory model of the signature library, in that way, eliminating any need for extra header information saved on the signature file.

```
procedure genLibrarySignature (lib:libraryFile, signLib:file, n:integer)
/* Pre: lib is a standard library file.
 *      signLib is the name of the output library signature file.
 *      n is the size of the signature in bytes.
 * Post: the signLib file is created, and contains all library subroutine signatures. */

    openFile (signLib).
    for (all subroutines $s \in$ lib) do
        if ($s$ has $n$ consecutive bytes) then
            sign[1..n] = first $n$ bytes of $s$.
        else /* $s$ has only $m < n$ bytes */
            sign[1..m] = first $m$ bytes of $s$.
            sign[m+1..n] = paddingValue.
        end if
        for (all variant bytes $b \in$ sign[1..n]) do
            sign[b] = wildcardValue.
        end for
        write (name($s$), sign[1..n]) to the file signLib.
    end for
    closeFile (signLib).
end procedure
```

Figure 8-6: Signature Algorithm

Integration of Library Signatures and the Decompiler

Given the entry point to a subroutine, the parser disassembles instructions following all paths from the entry point. If it is known that a particular compiler was used to compile the source binary program that is currently being analyzed, the parser can check whether the subroutine is one that belongs to a library (for that particular compiler) or not. If it does, the code does not need to be parsed since it is known which subroutine was invoked, and hence, the name of the subroutine is used.

Due to the large number of subroutines present in a library, a linear search is very inefficient for checking against all possible signatures in a file. Hashing is a good technique to use in this case, and even better, perfect hashing can be used since the signatures are unique for each subroutine in a given library, and have a fixed size. Perfect hashing information can be stored in the header of the library signature file, and used by the parser whenever needing to determine whether a subroutine belongs to the library or not.

8.2.2 Compiler Signature

In order to determine which library signature file to use with a binary program, the compiler that was used to compile the original user program needs to be determined. Since different binary patterns in the compiler start-up code are used by different compiler vendors, these patterns can be manually examined and stored in a signature that uses wildcards, in the same way as done for library subroutine signatures. Different memory models will provide different compiler signatures for the same compiler, and most likely, different versions of the same compiler have different signatures, therefore, a different signature for each (compiler vendor, memory model, compiler version) is stored. Again, a naming scheme can be used to differentiate different compiler signatures.

Determining the Main Program

The entry point given by the loader is the entry to the compiler start-up code, which invokes at least a dozen subroutines to set-up its environment before invoking the main subroutine of the program; i.e. the `main` in any C program, or the `BEGIN` in a Modula-2 program. The main entry point to a program compiled with a predefined compiler is determined by manual examination of the start-up code. In all C compilers, the parameters to the `main()` function (i.e. argv, argc, envp) are pushed before the main function is invoked; therefore, it is not very hard to determine the main entry point. Most C compilers provide the source code for their start-up code, in the interest of interoperability, hence the detection of the main entry point can be done in this way too. Once it is known how to determine the main entry point, this method is stored in the compiler signature file for that particular compiler.

Integration of Compiler Signatures with the Decompiler

Before the parser analyzes any instructions at the entry point given by the loader, an interface procedure is invoked to check for different compiler signatures. This procedure determines whether the first bytes of the loaded program are equivalent to a known compiler signature, and if so, the compiler vendor, compiler version, and memory model are determined, and stored in a global structure. Once this is done, the main entry point is determined by the signature, and that entry point is treated as the starting point for the parser. From there onwards, any subroutines called by the program can be checked against the library signature file for the appropriate compiler vendor, compiler version, and memory model.

8.2.3 Manual Generation of Signatures

Automatic generation of signatures is ideal, but it has the problem of finding a unique binary pattern that uniquely identifies all different subroutines in a library. Experimental results have shown that the number of repeated signatures across a standard library file varies from as low as 5.3% to as high as 29.7% [Emm94]. Most of the repeated signatures are due to functions that have different names but the same implementation, or due to unconditional jumps after a few bytes that force the signature to be cut short early.

A manual method for the generation of signatures was described in [FZ91], and used in an 8086 C decompiling system [FZL93]. A library file for the Microsoft C version 5.0 was analyzed by manual inspection of each function, and the following information was stored for each function: function name, binary pattern for the complete function (including variant bytes), and matching method to determine whether an arbitrary subroutine matches it or not. The matching method is a series of instructions that determines how many fixed bytes of information there are starting at an offset in the binary pattern for the function, and what subroutines are called by the function. Whenever an operand cannot be determined to be an offset or a constant, those bytes are skipped (i.e. they are not compared against the bytes in the binary pattern since they are variant bytes), and when a subroutine is called, the offset address of the subroutine is not tested, but the call to the routine is performed; which in turn is matched against the patterns in the signature. In this way, all paths of the subroutine are followed and checked against the signature.

The disadvantage of the manual generation of signatures is the time involved in generating them; typically a library has over 300 subroutines, and numbers increase to over 1300 for object oriented languages. Manual generation of signatures for the one library can take days, up to a week in large library files. Also, when a new version of the compiler is available, the signatures have to be reanalyzed manually, hence, the time overhead is great. Using an automatic signature generator reduces the amount of time to generate the signatures for a complete library to a few seconds (less than a minute), with the inconvenience of repeated signatures for a percentage of the functions. These repeated functions can be manually checked, and unique signatures generated for them if necessary.

8.3 Library Prototype Generator

A library prototype generator is a program that automatically generates information on the prototypes of library subroutines; that is, the type of the arguments used by the subroutine, and the type of the return value for functions. Determining prototype information on library subroutines helps the decompiler check for the right type and number of arguments, and propagate any type information that has wrongly been considered another type due to lack of information in the analysis. Consider the following code:

```
mov   ax, 42
push  ax
call  printf
```

During data flow analysis, this code is transformed into the following code after extended register copy propagation:

```
call  printf (42)
```

Without knowing the type of arguments that `printf` takes, the constant argument `42` is considered the right argument to this function call. But, if prototype information exists on this function, the function's formal argument list would have a fixed pointer to a character (i.e. a string in C) argument, and a variable number of other parameters of unknown type. Hence, the constant `42` could be determined to be an offset into the data segment rather than a constant, and replaced by that offset. This method provides the decompiler with the following improved code:

```
printf (''Hello world\n'');
```

and the disassembly version of the program could be improved to:

```
mov   ax, offset szHelloWorld
push  ax
call  printf
```

where `szHelloWorld` is the offset 42 into the data segment which points to the null terminated string.

It is therefore useful for the decompiler to use library prototypes. Unlike library signatures, there is a need for only one library prototype file for each high-level language (i.e. the standard functions of the language must all have the same prototypes). Compiler-dependent libraries require extra prototype files. Languages like C and Modula-2 have the advantage of using header files that define all library prototypes. These prototypes can be easily parsed by a program and stored in a file in a predetermined format. Languages such as Pascal store the library prototype information in their libraries, therefore, a special parser is required to read these files.

Comment on Runtime Support Routines

Compiler runtime support routines are subroutines used by the compiler to perform a particular task. These subroutines are stored in the library file, but do not have function prototypes available to the user (i.e. they are not in the header file of the library), hence they are used only by the compiler and do not follow high-level calling conventions. Most runtime subroutines have register arguments, and return the result in registers too. Since there is no prototype available for these subroutines, it is in the interest of the decompiler to analyze them in order to determine the register argument(s) that are being used, and the return register(s) (if any).

Runtime support routines are distinguished from any other library routine by checking the library prototypes: a subroutine that forms part of the library but does not have a prototype is a runtime routine. These routines have a name (e.g. `LXMUL`) but the type of the argument(s) and return value is unknown. During decompilation, these subroutines are analyzed, and the name from the library file is used to name the subroutine. Register arguments are mapped to formal arguments.

Integration of the Library Prototypes and the Decompiler

Whenever the type of compiler used to compile the original source program is determined by means of compiler signatures, the type of language used to compile that program is known; hence, the appropriate library prototype file can be used to determine more information on library subroutines used in the program. During parsing of the program, if a subroutine is determined to be one of the subroutines in a library signature file, the prototype file for that language is accessed, and this information along with the subroutine's name is stored in the subroutine's summary information record. This process provides the data flow analyzer with a complete certainty on the types of arguments to library subroutines, therefore, these types can be back-propagated to caller subroutines whenever found to be different to the ones in the prototype. Also, if the subroutine has a return type defined, it means that the subroutine is really a function, and hence, should be treated as one.

8.4 Disassembler

A disassembler is a program that converts a source binary program to a target assembler program. Assembly code uses mnemonics which represent machine opcodes; one or more machine opcodes are mapped to the same assembly mnemonic (e.g. all machine instructions that add two operands are mapped to the `add` mnemonic).

Disassemblers are used as tools to modify existing binary files for which there is no source available, to clarify undocumented code, to recreate lost source code, and to find out how a program works[Flu89, Com91]. In recent years, disassemblers are used as debugging tools in the process of determining the existence of virus code in a binary file, and the disassembly of such a virus; selective disassembly techniques are used to detect potential malicious code [Gar88].

A disassembler is composed of two phases: the parsing of the binary program, and the generation of assembler code. The former phase is identical to the parsing phase of the decompiler (see Chapter 4, Section 4.1), and the code generator produces assembler code on the fly or from an internal representation of the binary program. Symbol table information is also stored, in order to declare all strings and constants in the data segment.

Most public domain DOS disassemblers [Zan85, GD83, Cal, Mak90, Sof88, Chr80] perform one pass over the binary image without constructing a control flow graph of the program. In most cases, parsing errors are introduced by assumptions made on memory locations, considering them code when they represent data. Some of these disassemblers comment on different DOS interrupts, and are able to disassemble not only binary files, but blocks of memory and system files. Commercial disassemblers like Sourcer [Com91] perform several passes through the binary image, refining the symbol table on each pass, and assuring a better distinction between data and code. An internal simulator is used to resolve indexed jumps and calls, by keeping track of register contents. Cross-reference information is also collected by this disassembler.

In decompilation, the disassembler can be considered part of the decompiler by adding an extra assembler code generator phase, such as in Figure 8-7, or can be used as a tool to generate an assembler program that is taken as the source input program to the decompiler, such as in the initial Figure 8-1.

8.5 Language Independent Bindings

The decompiler generates code for the particular target language it was written for. Binary programs decompiled with the aid of compiler and library signatures produce target language programs that use the names of the library routines defined in the library signature file. If the language in which the binary program was originally written in is different to the target language of the decompiler, the target program cannot be re-compiled for this language since it uses library routines defined in another language/compiler. Consider the following fragment of decompiled code in C:

```
WriteString ("Hello Pascal");
CrLf();
```

These two statements invoke Pascal library routines that implement the original Pascal statement `writeln ("Hello Pascal");` the first routine displays the string and the second performs a carriage return, line feed. In other words, since there is no `writeln` library

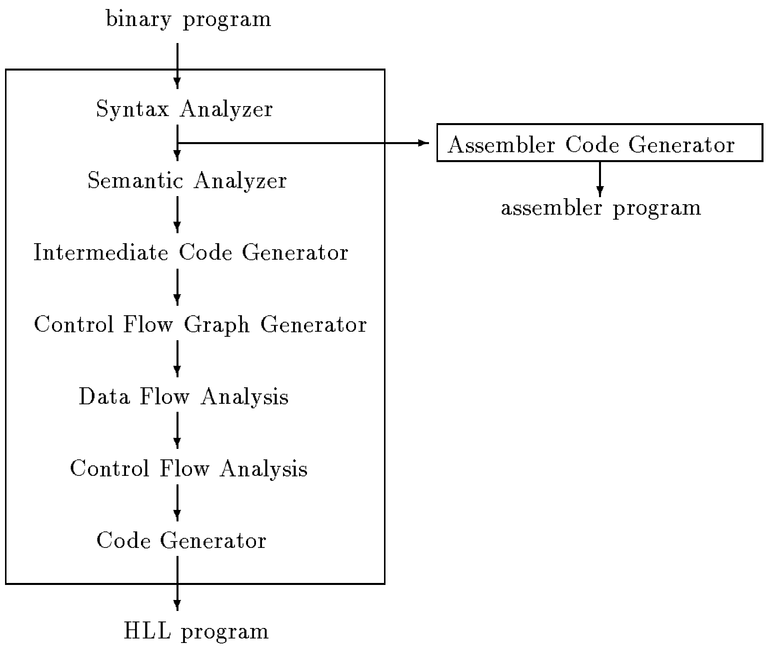


Figure 8-7: Disassembler as part of the Decompiler

routine in the Pascal libraries, this call is replaced by the calls to `WriteString` and `CrLf`. The decompiled code is correct, but since the target language is C, it cannot be re-compiled given that `WriteString` and `CrLf` do not belong to the C library routines.

The previous problem can be solved with the use of Pascal to C bindings for libraries. In this way, rather than generating the previous two statements, a call to `printf` is used, as follows:

```
printf ("Hello Pascal\n");
```

ISO committee SC22 of Working Group 11 is concerned with the creation of standards for language independent access to service facilities. This work can be used to define language independent bindings for languages such as C and Modula-2. Information on library bindings can be placed in a file and used by the code generator of the decompiler to produce target code that uses the target language's library routines.

8.6 Postprocessor

The quality of the target high-level language program generated by the decompiler can be improved by a postprocessor phase that replaces generic control structures by language-specific structures. Language-specific structures were not considered in the structuring analysis of Chapter 6 because these constructs are not general enough to be used across several languages.

In C, the `for()` loop is implemented by a `while()` loop that checks for the terminating condition. The induction variable is initialized before the loop, and is updated each time around the loop in the last statement of the `while()`. Consider the following code in C after decompilation:

```
 1   loc1 = 0;
 2   while (loc1 < 8)
     {
 3      if (loc1 != 4)
        {
 4         printf ("%d", loc1);
        }
 5      loc1 = loc1 + 1;
     }
```

The `while()` loop at statement 2 checks the local variable `loc1` against constant 8. This variable was initialized in statement 1, and is also updated in the last statement of the loop (i.e. statement 5); therefore, this variable is an induction variable, and the `while()` loop can be replaced by a `for` loop, leading to the following code:

```
 2   for (loc1 = 0; loc1 < 8; loc1 = loc1 + 1)
     {
 3      if (loc1 != 4)
        {
 4          printf ("%d", loc1);
        }
     }
```

which eliminates instructions 1 and 5, replacing them into instruction 2. Pre and post increment instructions are used in C as well, hence, the previous code can be improved to the following:

```
 2   for (loc1 = 0; loc1 < 8; loc1++)
     {
 3      if (loc1 != 4)
        {
 4          printf ("%d", loc1);
        }
     }
```

A `break` statement in C terminates the execution of the current loop, branching control to the first instruction that follows the loop. Consider the following code after decompilation:

```
 1   loc1= 0;
 2   while (loc1 < 8)
     {
 3      printf ("%d", loc1);
 4      if (loc1 == 4)
 5         goto L1;
 6      loc1 = loc1 + 1;
     }
 7 L1:
```

Instruction 4 checks local variable `loc1` against 4, and if they are equal, a `goto` jump is executed, which transfers control to label `L1`; the first instruction after the loop. This transfer of control is equivalent to a `break`, which removes the need for the label and the `goto`. Also, the loop is transformed into a `for` loop, leading to the following code:

```
2   for (loc1 = 0; loc1 < 8; loc1++)
    {
3      printf ("%d", loc1);
4      if (loc1 == 4)
5         break;
    }
```

In a similar way, `continue` statements can be found in the code. If the target language was Ada, labelled multiexit loops are allowed. These would have been structured by the decompiler as loops with several `goto` jump exits out of the loop. The target statements of these `goto` jumps can be checked for enclosing loop labels, and replaced by the appropriate `exit loopName` statement.

In general, any language-specific construct can be represented by the generic set of constructs used in the structuring algorithm of Chapter 6, Section 6.4.3; these constructs can be replaced by a postprocessor, but it is not strictly necessary to do so since the constructs are functionally equivalent.

Chapter 9

dcc

dcc is a prototype decompiler written in C for the DOS operating system. **dcc** was initially developed on a DecStation 3000 running Ultrix, and was ported to the PC architecture under DOS. **dcc** takes as input **.exe** and **.com** files for the Intel i80286 architecture, and produces target C and assembler programs. This decompiler was built using the techniques described in this thesis (Chapters 4,5,6,7, and 8), and is composed of the phases shown in Figure 9-1. As can be seen, the decompiler has a built-in loader and disassembler, and there is no postprocessing phase. The following sections describe specific aspects about **dcc**, and a series of decompiled programs are given in Section 9.7.

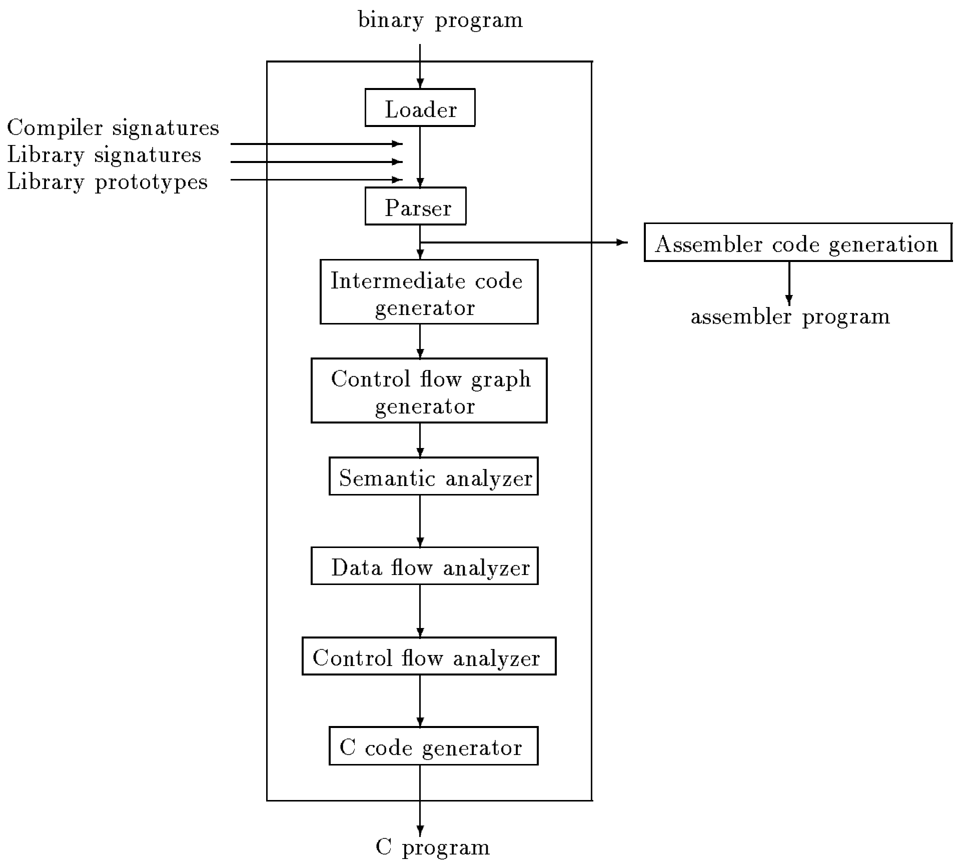


Figure 9-1: Structure of the dcc Decompiler

The main decompiler program is shown in Figure 9-2, with five major modules identified: the initArgs() which reads the user options from the command line argv[] and places them in a global program options variable; the Loader() which reads the binary program and loads it into memory; the FrontEnd() which parses the program building a call graph; the udm() which analyses the control and data flow of the program; and the BackEnd() which generates C code for the different routines in the call graph.

```
Int main(int argc, char *argv[])
{ char *filename;           /* Binary file name                   */
  CALL_GRAPH *callGraph;    /* Pointer to the program's call graph */

    filename = initArgs(argc, argv);

    /* Read a .exe or .com file and load it into memory */
    Loader (filename);

    /* Parse the program, generate Icode while building the call graph */
    FrontEnd (filename, &callGraph);

    /* Universal Decompiling Machine: process the Icode and call graph */
    udm (callGraph);

    /* Generates C for each subroutine in the call graph */
    BackEnd(filename, callGraph);
}
```

Figure 9-2: Main Decompiler Program

The DOS operating system uses a segmented machine representation. Compilers written for this architecture make use of 6 different memory models: tiny, small, medium, compact, large, and huge. Memory models are derived from the choice of 16- or 32-bit pointers for code and data. Appendix A provides information on the i80286 architecture, and Appendix B provides information on the PSP. This chapter assumes familiarization with this architecture.

Decompiler Options

dcc is executed from the command line by specifying the binary file to be decompiled. For example, to decompile the file test.exe the following command is entered:

```
dcc test.exe
```

This command produces the test.b file, which is the target C file. There are several options available to the user to get more information on the program. These options are:

- `a1`: produces an assembler file after parsing (i.e. before graph optimization).
- `a2`: produces an assembler file after graph optimization.
- `o <fileName>`: uses the `fileName` as the name for the output assembler file.
- `m`: produces a memory map of the program.
- `s`: produces statistics on the number of basic blocks before and after optimization for each subroutine's control flow graph.
- `v`: verbose option, displays information on the loaded program (default register values, image size, etc), basic blocks of each subroutine's graph, defined and used registers of each instruction, and the liveIn, liveOut, liveUse, and defined register sets of each basic block.
- `V`: veryVerbose option, displays the information displayed by the verbose option, plus information on the program's relocation table (if any), basic blocks of the control flow graph of each subroutine before graph optimization, and the derived sequence of graphs of each subroutine.
- `i`: text user interface for **dcc**. This interface was written by Michael Van Emmerik using the curses library. It allows the user to step the program, including subroutine calls. The right arrow is used to follow jumps and subroutine calls, the left arrow isused to step back to where you were before using the right arrow, up and down arrows are used to move up/down a line at a time, page up and page down are used to scroll a page up or down, and ctrl-X is used to exit the interactive interface.

9.1 The Loader

The DOS loader is an operating system program called `exec`. `Exec` checks for sufficient available memory to load the program, allocates a block of memory, builds the PSP at its base, reads the program into the allocated memory block after the PSP, sets up the segment registers and the stack, and transfers control to the program[Dun88a].

Since the decompiler needs to have control of the program, the `exec` program was not used, but a loader that performs a similar task was written. For **.exe** programs, the program header is checked for the amount of memory required and the location of the relocation table, the size of the image in bytes is dynamically allocated and the program is then loaded into memory and relocated. For **.com** programs, the amount of memory required is calculated from the size of the file, memory is dynamically allocated, and the program is loaded into memory as is. The format of these files is given in Appendix C.

Memory is represented in **dcc** by an array of bytes; a large enough array is dynamically allocated once the size of the program's image is determined. For historical reasons, **.com** programs are loaded at offset 0100h. The loader also stores information relating to the program in a `PROG` record, defined in Figure 9-3. This record stores not only the information that was on the binary file, but also the memory map, and the address (segment, offset) where the program was loaded (this address is fixed but dependent on the type of binary program).

```
typedef struct {
    int16      initCS;       /* Initial CS register value         */
    int16      initIP;       /* Initial IP register value         */
    int16      initSS;       /* Initial SS register value         */
    int16      initSP;       /* Initial SP register value         */
    boolT      fCOM;         /* Flag set if COM program (else EXE) */
    Int        cReloc;       /* # of relocation table entries     */
    dword     *relocTable;   /* Pointer to relocation table       */
    Int        cProcs;       /* Number of subroutines             */
    Int        offMain;      /* The offset  of the main() proc    */
    word       segMain;      /* The segment of the main() proc    */
    boolT      libSigs;      /* True if library signatures loaded  */
    Int        cbImage;      /* Length of image in bytes          */
    byte      *Image;        /* Entire program image              */
    byte      *map;          /* Memory bitmap pointer             */
} PROG;
```

Figure 9-3: Program Information Record

9.2 Compiler and Library Signatures

The DOS operating system does not provide a method to share libraries, therefore library routines are bound to the program's image. Compiler and library signatures were generated for several compilers due to this reason; Section 9.3.1 explains how they are used in **dcc**.

An automatic signature generator was written to generate library signatures for standard `.lib` files, as described in Chapter 8, Section 8.2.1. The length of the signature was set to 23 bytes, which was proved to be a reasonable size by experimental results. The wildcard byte was `F4` (the opcode for `HALT`) since this opcode is rarely used, and the padding byte was set to `00`. Library signatures were generated for the following C compilers: Microsoft C 5.1, Microsoft Visual C++ V1.00, Turbo C 2.01, and Borland C V3.0. A separate library signature file was used for each triplet of compiler vendor, memory model, and compiler version. Signatures were generated in a few seconds.

Since automatic signature generation was used, repeated signatures were detected. The numbers varied from as low as 5.3% for Turbo C 2.01, to as high as 29.7% for Microsoft Visual C++ V1.00. In the former case, 19 out of 357 routines had repeated signatures. These were mainly due to identical representation of routines with different names, such as `spawnvp`, `spawnvpe`, `spawnve`, `spawnv`, `spawnlp`, and `spawnl`. A few signatures were identical for similar functions, such as `tolower` and `toupper`. In only one case unrelated functions had the same signature; these functions are `brk` and `atoi`. In the latter case, 440 out of 1327 routines had the same signature. Most of these duplicates were due to internal public names that are not accessable by the user, such as `_CIcosh` and `_CIfabs`. Other signatures use different names for the same routines, especially due to the naming convention used by different memory models (i.e. the same routine works in different memory models) [Emm94].

Pascal compilers do not use standard library files. In the case of the Borland Pascal compilers, all library information is stored in a `.tpl` file, which has information on library routines and prototypes. A modified signature generator was written for `.tpl` files, and signatures were generated for Turbo Pascal version 4.0 and 5.0.

On average, the library signature files occupy 50Kb of disk space, which is moderate for the amount of library routines' information stored in them.

Compiler signatures for the above compilers were generated manually and stored as part of **dcc**. These signatures are checked for when the parser is first invoked.

The implementation of the signature and prototype generator is due to Michael Van Emmerik while working for the Queensland University of Technology. This work is reported in [Emm94].

9.2.1 Library Prototypes

A program called `parsehdr` was written to parse C library header files, isolate prototypes, and store the information about the argument types and the return type to a file. Prototypes were generated for the standard libraries used in C.

In the case of Pascal, prototype information is stored as part of the `.tpl` library file. These prototypes were not generated due to missing information regarding the exact structure of the prototype information.

9.3 The Front-end

The front-end constructs a call graph of the program while parsing the loaded program in memory. For each subroutine, the intermediate code and control flow graph are attached to the subroutine node in the call graph; hence, the parsing, intermediate code generation, and the construction of the flow graph are done in the same pass through the program's image. Data information is stored in global and local symbol tables. If the user requests for disassembly, an assembler file is written out to a file with extension `.a1`, and if the user requested interactive interface, an interactive window is displayed and the user can follow the program by stepping through the instructions. Semantic analysis is done last, followed by the displaying of the bitmap (if user requested). Figure 9-4 shows the code for the `FrontEnd()` procedure.

9.3.1 The Parser

The parser determines whether the code reached from the entry point provided by the loader is equivalent to one of the compiler signatures stored in the program, if so, the `main` to the program is determined and used as the entry point for the analysis. Whenever a

```
void FrontEnd (char *filename, CALL_GRAPH *callGraph)
{
    /* Parse image while building call graph and generating Icode */
    parse (callGraph);

    /* Transform basic block list into a graph */
    constructGraph (callGraph);

    /* Check for bytes used as both data and code */
    checkDataCode (callGraph);

    if (option.asm1)     /* disassembly of the program */
    {
        printf ("dcc: writing assembler file %s.a1\n", filename);
        disassemble (1, callGraph, filename);
    }

    if (option.Interact) /* interactive option, display window */
        interactWin (callGraph);

    /* Idiom analysis */
    semAnalyzer (callGraph);

    /* Remove redundancies from the graph */
    compressCFG (callGraph);

    if (option.stats)    /* statistics on the basic blocks */
        displayStats (callGraph);

    if (option.asm2)     /* disassembly after graph compression */
        disassemble (2, callGraph, filename);

    if (option.map)      /* display memory bitmap */
        displayMemMap();
}
```

Figure 9-4: Front-end Procedure

compiler signature is recognized, the associated library signature file is loaded. The parsing process is not affected in any way if a compiler signature is not found. In these cases, all code reached from the entry point provided by the loader is decompiled, and no library routine recognition is done. It is important to point out that some compilers have set-up routines that are hard to parse since they use indirect jumps; in these cases, the complete code cannot be parsed, and the decompilation is jeopardized.

Given the entry point to a subroutine, the parser implements the data/instruction separation algorithm described in Chapter 4, Figure 4-7. This algorithm recursively follows all paths from the entry point, and emulates loads into registers (whenever possible). When a subroutine call is met, the entry address to the subroutine becomes a new entry point which is analyzed in a recursive way, placing the subroutine information in the call graph. Register content is emulated to detect such cases as end of program via interrupts, which relies on the contents of one or more registers. Programs in which the compiler signature was recognized are known to be terminated by a routine that is executed after the finishing of the `main` program, hence, emulating the contents of registers in this case is not necessary. This parser does not make any attempt at recognizing uplevel addressing.

Figure 9-5 shows the definition of the `PROC` record, which stores information about a subroutine. Note that during parsing not all of the fields are filled with information; some are later filled by the universal decompiling machine.

```
typedef struct _proc {
    Int          procEntry;     /* label number                              */
    char         name[SYMLEN];  /* Meaningful name for this proc             */
    STATE        state;         /* Entry state                               */
    flags32      flg;           /* Combination of Icode & Procedure flags    */
    int16        cbParam;       /* Probable no. of bytes of parameters       */
    STKFRAME     args;          /* Array of formal arguments                 */
    LOCAL_ID     localId;       /* Local symbol table                        */
    ID           retVal;        /* Return value type (for functions)         */

    /* Icodes and control flow graph */
    ICODE_REC    Icode;         /* Record of ICODE records                   */
    PBB          cfg;           /* Pointer to control flow graph (cfg)       */
    PBB          *dfsLast;      /* Array of pointers to BBs in revPostorder  */
    Int          numBBs;        /* Number of basic blocks in the graph cfg   */
    boolT        hasCase;       /* Boolean: subroutine has an n-way node     */

    /* For interprocedural live analysis */
    dword        liveIn;        /* Registers used before defined             */
    dword        liveOut;       /* Registers that may be used in successors  */
    boolT        liveAnal;      /* Procedure has been analysed already       */
} PROC;
```

Figure 9-5: Procedure Record

The parser is followed by a `checkDataCode()` procedure which checks each byte in the bitmap for having two flags: data, and code; in which case the byte position is flagged as being data and code, and the corresponding subroutine is flagged as potentially using self-modifying code.

9.3.2 The Intermediate Code

The intermediate code used in **dcc** is called Icode, of which there are two types: low-level and high-level. The low-level Icode is a mapping of i80286 machine instructions to assembler mnemonics, ensuring that every Icode instruction performs one logical instruction only. For example, the instruction:

```
DIV bx
```

assigns to `ax` the quotient of `dx:ax` divided by `bx`, and assigns to `dx` the reminder of the previous quotient; hence, two logical instructions are performed by the `DIV` machine instruction. In Icode instructions, `DIV` is separated into two different instructions: `iDIV` and `iMOD`. The former performs the division of the operands, and the latter performs the modulus of the operands. Since both instructions use the registers that are overwritten by the result of the instruction (i.e. `dx` and `ax` in this example), these registers need to be placed in a temporary register before the instructions are performed. **dcc** uses register `tmp` as a temporary register. This register is forward substituted into another instruction and eliminated during data flow analysis. The above machine instruction is translated into three Icode instructions as follows:

```
iMOV tmp, dx:ax    ; tmp = dx:ax
iDIV bx            ; ax = tmp / bx
iMOD bx            ; dx = tmp % bx
```

where the dividend of both `iDIV` and `iMOD` is set to the `tmp` register rather than `dx:ax`. Figure 9-6 shows the different machine instructions that are represented by more than one Icode instruction. An example is given for each instruction.

Machine Instruction	Icode Instructions	Meaning

DIV cl	iMOV tmp, ax
	iDIV cl
	iMOD cl
		tmp = ax
		al = tmp / cl
		ah = tmp % cl

LOOP L	iSUB cx, 1 iJNCXZ L	cx = cx - 1 cx <> 0 goto L

LOOPE L	iSUB cx, 1
	iCMP cx, 0
	iJZ L
	iJNCXZ L
		cx = cx - 1
		cx == 0?
		zeroFlag == 1 goto L
		if cx <> 0 goto L

LOOPNE L	iSUB cx, 1
	iCMP cx, 0
	iJNE L
	iJNCXZ L
		cx = cx = 1
		cx == 0?
		zeroFlag == 0 goto L
		if cx <> 0 goto L

XCHG cx, bx	iMOV tmp, cx
	iMOV cx, bx
	iMOV bx, tmp
		tmp = cx
		cx = bx
		bx = tmp

Figure 9-6: Machine Instructions that Represent more than One Icode Instruction

Compound instructions such as `rep movsb` are represented by two different machine instructions but perform one logical string function; repeat while not end-of-string in this case. These instructions are represented by one Icode instruction; `iREP_MOVS` in this example.

Machine instructions that perform low-level tasks, such as input and output from a port, are most likely never generated by a compiler whilst compiling high-level language code (i.e. embedded assembler code can make use of these instructions but the high-level code does not generate these instructions). These instructions are flagged in the Icode as being non high-level, and the subroutine that makes use of these instructions is flagged as well so that assembler is generated for the subroutine. The following instructions are considered not to be generated by compilers; the instructions marked with an asterisk are sometimes non high-level, depending on the register operands used:

```
AAA, AAD, AAM, AAS, CLI, DAA, DAS, *DEC, HLT, IN, *INC, INS,
INT, INTO, IRET, JO, JNO, JP, JNP, LAHF, LOCK, *MOV, OUT, OUTS,
*POP, POPA, POPF, *PUSH, PUSHA, PUSHF, SAHF, STI, *XCHG, XLAT
```

Icode instructions have a set of Icode flags associated with them to acknowledge properties found during the parsing of the instruction. The following flags are used:

- B: byte operands (default is word operands).
- I: immediate (constant) source operand.
- No_Src: no source operand.
- No_Ops: no operands.
- Src_B: source operand is byte, destination is word.
- Im_Ops: implicit operands.
- Im_Src: implicit source operand.
- Im_Dst: implicit destination operand.
- Seg_Immed: instruction has a relocated segment value.
- Not_Hll: non high-level instruction.
- Data_Code: instruction modifies data.
- Word_Off: instruction has a word offset (i.e. could be an address).
- Terminates: instruction terminates the program.
- Target: instruction is the target of a jump instruction.
- Switch: current indirect jump determines the start of an n-way statement.
- Synthetic: instruction is a synthetic (i.e. does not exist in the binary file).
- Float_Op: the next instruction is a floating point instruction.

dcc implements the mapping of i80286 machine instructions to low-level Icodes by means of a static table indexed by machine instruction, which has information on the associated Icode, used and defined condition codes, flags, and procedures that determine the source and destination operands (different procedures are used for different operand types, thus, the same procedures are used by several different instructions). The mapping of machine instructions to Icode instructions converts 250 instructions into 108 Icode instructions. This mapping is shown in Figure 9-7.

9.3.3 The Control Flow Graph Generator

dcc implements the construction of the control flow graph for each subroutine by placing basic blocks on a list and then converting that list to a proper graph. While parsing, whenever an end of basic block instruction is met, the basic block is constructed, and the start and finish instruction indexes into the Icode array for that subroutine are stored. Instructions for which it is not possible to determine where they transfer control to (i.e. indexed jumps that are not recognized as a known n-way structure header, indirect calls, etc) are said to terminate the basic block since no more instructions are parsed along the path that contains that instruction. These nodes are called no-where nodes in **dcc**. The other types of basic blocks are the standard 1-way, 2-way, n-way, fall-through, call, and return nodes. The definition record of a basic block is shown in Figure 9-8. Most of this information is later filled in by the universal decompiling machine.

The control flow graph of each subroutine is optimized by flow-of-control optimizations which remove redundant jumps to jumps, and conditional jumps to jumps. These optimizations have the potential of removing basic blocks from the graph, therefore the numbering of the graph is left until all possible nodes are removed from the graph. At the same time, the predecessors to each basic block are determined and placed in the `*inEdges[]` array.

9.3.4 The Semantic Analyzer

dcc's semantic analyzer determines idioms and replaces them with another Icode instruction(s). The idioms checked for in **dcc** are the ones described in Chapter 4, Section 4.2.1, and grouped into the following categories: subroutine idioms, calling conventions, long variable operations, and miscellaneous idioms.

There is a series of idioms available only in C. In C, a variable can be pre and post incremented, and pre and post decremented. The machine code that represents these instructions makes use of an extra register to hold the value of the pre or post incremented/decremented variable when it is being checked against some value/variable. This extra register can be eliminated by using an idiom to transform the set of instructions into one that uses the pre/post increment/decrement operand.

In the case of a post increment/decrement variable in a conditional jump, the value of the variable is copied to a register, the variable then gets incremented or decremented, and finally, the register that holds the copy of the initial variable (i.e. before increment or decrement) is compared against the other identifier. The use of the extra register can be eliminated by using the post increment/decrement operator available in C. Therefore, these idioms can be checked for only if code is to be generated in C. Figure 9-9 shows this case.

Low-level Instruction	Machine Instruction(s)
iAAA	37
iAAD	D5
iAAM	D4
iAAS	3F
iADC	10..15, (80..83)(50..57,90..97,D0..D7)
iADD	00..05, (80..83)(40..47,80..87,C0..C7)
iAND	20..25, (80..83)(60..67,A0..A7,E0..E7)
iBOUND	62
iCALL	E8, FE50..FE57, FE90..FE9F, FED0..FED7, FF50..FF57, FF90..FF9F, FFD0..FFD7
iCALLF	9A, FE58..FE5F, FE98..FE9F, FED8..FEDF, FF58..FF5F, FF98..FF9F, FFD8..FFDF
iCLC	F8
iCLD	FC
iCLI	FA
iCMC	F5
iCMP	38..3D, (80..83)(78..7F,B8..BF,F8..FF)
iCMPS	A6, A7
iREPNE_CMPS	F2A6, F2A7
iREPE_CMPS	F3A6, F3A7
iDAA	27
iDAS	2F
iDEC	48..4F, FE48..FE4F, FE88..FE8F, FEC8..FECF, FF48..FF4F, FF88..FF8F, FFC8..FFCF
iDIV	F670..F677, F6A0..F6A7, F6F0..F6F7, F770..F777, F7A0..F7A7, F7F0..F7F7
iMOD	F670..F677, F6A0..F6A7, F6F0..F6F7, F770..F777, F7A0..F7A7, F7F0..F7F7, F678..F67F, F6A8..F6AF, F6F8..F6FF, F778..F77F, F7A8..F7AF, F7F8..F7FF
iENTER	C8
iESC	D8..DF
iHLT	F4
iIDIV	F678..F67F, F6A8..F6AF, F6F8..F6FF, F778..F77F, F7A8..F7AF, F7F8..F7FF
iIMUL	69, 6B, F668..F66F, F6A8..F6AF, F6E8..F6EF, F768..F76F, F7A8..F7AF, F7E8..F7EF
iIN	E4, E5, EC, ED
iINC	40..47, FE40..FE47, FE80..FE87, FEC0..FEC7, FF40..FF47, FF80..FF87, FFC0..FFC7
iINS	6C, 6D
iREP_INS	F36C, F36D
iINT	CC, CD

Figure 9-7: Low-level Intermediate Code for the i80286

Low-level Instruction	Machine Instruction(s)
iINTO	CE
iIRET	CF
iJB	72
iJBE	76
iJAE	73
iJA	77
iJE	74
iJNE	75
iJL	7C
iJGE	7D
iJLE	7E
iJG	7F
iJS	78
iJNS	79
iJO	70
iJNO	71
iJP	7A
iJNP	7B
iJCXZ	E3
iJNCXZ	E0..E2
iJMP	E9, EB, FE60..FE67, FEA0..FEA7, FEE0..FEE7, FF60..FF67, FFA0..FFA7, FFE0..FFE7
iJMPF	EA, FE68..FE6F, FEA8..FEAF, FEE8..FEEF, FF68..FF6F, FFA8..FFAF, FFE8..FFEF
iLAHF	9F
iLDS	C5
iLEA	8D
iLEAVE	C9
iLES	C4
iLOCK	F0
iLODS	AC, AD
iREP_LODS	F3AC, F3AD
iMOV	88..8C, 8E, A0..A3, B0..BF, C6, C7
iMOVS	A4, A5
iREP_MOVS	F3A4, F3A5
iMUL	F660..F667, F6A0..F6A7, F6E0..F6E7, F760..F767, F7A0..F7A7, F7E0..F7E7
iNEG	F658..F65F, F698..F69F, F6D8..F6DF, F758..F75F, F798..F79F, F7D8..F7DF
iNOT	F650..F657, F690..F697, F6D0..F6D7, F750..F757, F790..F797, F7D0..F7D7
iNOP	90
iOR	08..0D, (80..83)(48..4F,88..8F,C8..CF)

Figure 9-7: Low-level Intermediate Code for the i80286 - Continued

Low-level Instruction	Machine Instruction(s)
iOUT	E6, E7, EE, EF
iOUTS	6E, 6F
iREP_OUTS	F36E, F36F
iPOP	07, 17, 1F, 58..5F, 8F
iPOPA	61
iPOPF	9D
iPUSH	06, 0E, 16, 1E, 50..57, 68, 6A, FE70..FE77, FEB0..FEB7, FEF0..FEF7, FF70..FF77, FFB0..FFB7, FFF0..FFF7
iPUSHA	60
iPUSHF	9C
iRCL	(C0,C1,D0..D3)(50..57,90..97,D0..D7)
iRCR	(C0,C1,D0..D3)(58..5F,98..9F,D8..DF)
iREPE	F3
iREPNE	F2
iRET	C2, C3
iRETF	CA, CB
iROL	(C0,C1,DO..D3)(40..47,80..87,C0..C7)
iROR	(C0,C1,D0..D3)(48..4F,88..8F,C8..CF)
iSAHF	9E
iSAR	(C0,C1,D0..D3)(78..7F,B8..BF,F8..FF)
iSHL	(C0,C1,D0..D3)(60..67,A0..A7,E0..E7)
iSHR	(C0,C1,D0..D3)(68..6F,A8..AF,E8..EF)
iSBB	18..1D, (80..83)(58..5F,98..9F,D8..DF)
iSCAS	AE, AF
iREPE_SCAS	F3AE, F3AF
iREPNE_SCAS	F2AE, F2AF
iSIGNEX	98, 99
iSTC	F9
iSTD	FD
iSTI	FB
iSTOS	AA, AB
iREP_STOS	F3AA, F3AB
iSUB	28..2D, (80..83)(68..6F,A8..AF,E8..EF)
iTEST	84, 85, A8, A9, F640..F647, F680..F687, F6C0..F6C7, F740..F747, F780..F787, F7C0..F7C7
iWAIT	9B
iXCHG	86, 87, 91..97
iXLAT	D7
iXOR	30..35, (80..83)(70..77,B0..B7,F0..F7)

Figure 9-7: Low-level Intermediate Code for the i80286 - Continued

```
typedef struct _BB {
    byte            nodeType;       /* Type of node                     */
    Int             start;          /* First instruction offset         */
    Int             finish;         /* Last instruction in this BB      */
    flags32         flg;            /* BB flags                         */
    Int             numHlIcodes;    /* # of high-level Icodes           */

    /* In edges and out edges */
    Int             numInEdges;     /* Number of in edges               */
    struct _BB    **inEdges;        /* Array of pointers to in-edges    */
    Int             numOutEdges;    /* Number of out edges              */
    union typeAdr {
      dword         ip;             /* Out edge Icode address           */
      struct _BB   *BBptr;          /* Out edge pointer to successor BB */
      interval     *intPtr;         /* Out edge pointer to next interval */
    }              *outEdges;       /* Array of pointers to out-edges   */

    /* For interval and derived sequence construction */
    Int             beenOnH;        /* #times been on header list H     */
    Int             inEdgeCount;    /* #inEdges (to find intervals)     */
    struct _BB     *reachingInt;    /* Reaching interval header         */
    interval       *inInterval;     /* Node's interval                  */
    interval       *correspInt;     /* Corresponding interval in Gi-1   */

    /* For live register analysis */
    dword           liveUse;        /* LiveUse(b)                       */
    dword           def;            /* Def(b)                           */
    dword           liveIn;         /* LiveIn(b)                        */
    dword           liveOut;        /* LiveOut(b)                       */

    /* For structuring analysis */
    Int             preorder;       /* DFS #: first visit of the node   */
    Int             revPostorder;   /* DFS #: last visit of the node    */
    Int             immedDom;       /* Immediate dominator (revPostorder) */
    Int             ifFollow;       /* follow node (if node is 2-way)   */
    Int             loopType;       /* Type of loop (if any)            */
    Int             latchNode;      /* latching node of the loop        */
    Int             numBackEdges;   /* # of back edges      */
    Int             loopFollow;     /* node that follows the loop       */
    Int             caseFollow;     /* follow node for n-way node       */

    /* Other fields */
    Int             traversed;      /* Boolean: traversed yet?          */
    struct _BB     *next;           /* Next (initial list link)         */
} BB;
```

Figure 9-8: Basic Block Record

```
mov  reg, var           mov  reg, var
inc  var         or     dec  var
cmp  var, Y             cmp  var, Y
jX   label              jX   label

                  ⇓

jcond (var++ X Y)      jcond (var-- X Y)
```

Figure 9-9: Post-increment or Post-decrement in a Conditional Jump

In a similar way, a pre increment/decrement makes use of an intermediate register. The variable is first incremented/decremented, then it is moved onto a register, which is compared against another identifier, and then the conditional jump occurs. In this case, the intermediate register is used because identifiers other than a register cannot be used in the compare instruction. This intermediate register can be eliminated by means of a pre increment/decrement operator, as shown in Figure 9-10.

```
inc var                 dec var
mov reg, var     or     mov reg, var
cmp reg, Y              cmp reg, Y
jX  lab                 jX  lab

                ⇓

jcond (++var X Y)    jcond (--var X Y)
```

Figure 9-10: Pre Increment/Decrement in Conditional Jump

C-dependent idioms are implemented in **dcc**. As seen in the general format of these idioms, a series of low-level Icode instructions is replaced by one high-level `jcond` instruction. This instruction is flagged as being a high-level instruction so that it is not processed again by the data flow analyzer. Also, all other instructions involved in these idioms are flagged as not representing high-level instructions.

After idiom recognition, simple type propagation is done on signed integers, signed bytes, and long variables. When propagating long variables across conditionals, the propagation modifies the control flow graph by removing a node from the graph, as described in Chapter 4, Section 4.2.2. Since the high-level condition is determined from the type of graph, the corresponding high-level `jcond` instruction is written and that instruction is flagged as being a high-level instruction.

9.4 The Disassembler

dcc implements a built-in disassembler that generates assembler files. The assembly file contains only information on the assembler mnemonics of the program (i.e. the code segment) and does not display any information relating to data. All the information used by the disassembler is collected by the parser and intermediate code phases of the decompiler; since there is almost a 1:1 mapping of low-level Icodes to assembler mnemonics, the assembler code generator is mostly concerned with output formatting.

The disassembler handles one subroutine at a time; given a call graph, the graph is traversed in a depth-first search to generate assembler for nested subroutines first. The user has two options for generating assembler files: to generate assembler straight after the parsing phase, and to generate assembler after graph optimization. The former case generates assembler that is as close as possible to the binary image; the latter case may miss certain jump instructions that were considered redundant by the graph optimizer. The disassembler is also used by the decompiler when generating target C code; if a subroutine is flagged as being a non high-level subroutine, assembler code is generated for that subroutine after generating the subroutine's header and comments in C.

9.5 The Universal Decompiling Machine

The universal decompiling machine (udm) is composed of two phases; the data flow analysis phase which transforms the low-level Icode to an optimal high-level Icode representation, and the control flow analysis phase which traverses the graph of each subroutine to determine the bounds of loops and conditionals; these bounds are later used by the code generator. Figure 9-11 shows the code for the `udm()` procedure.

9.5.1 Data Flow Analysis

The first part of the data flow analysis is the removal of condition codes. Condition codes are classified into two sets as follows: the set of condition codes that are likely to have been generated by a compiler (the HLCC set), and the set of conditions that are likely to have been hand-crafted in assembler (the NHLCC set). From the 9 condition codes available in the Intel i80286[Int86] (overflow, direction, interrupt enable, trap, sign, zero, auxiliary carry, parity and carry), only 4 flags are likely to be high-level; these are, carry, direction, zero and sign. These flags are modified by instructions that are likely to be high-level (i.e the ones that were not flagged as being non high-level), and thus this set is the one that is analyzed for condition code removal. From the probable high-level instructions, 30 instructions define flags in the HLCC set; ranging from 1 to 3 flags defined by an instruction, and 25 instruction use flags; normally using one or two flags per instruction. **dcc** implements dead-condition code elimination and condition code propagation, as described in Chapter 5, Sections 5.4.2 and 5.4.3. These optimizations remove all references to condition codes and creates `jcond` instructions that have an associated Boolean conditional expression. This analysis is overlapped with the initial mapping of all other low-level Icodes to high-level Icodes in terms of registers. The initial mapping of Icodes is explained in Appendix D.

```
void udm (CALL_GRAPH *callGraph)
{ derSeq *derivedG;

    /* Data flow analysis - optimizations on Icode */
    dataFlow (callGraph);

    /* Control flow analysis -- structure the graphs */
    /* Build derived sequences for each subroutine */
    buildDerivedSeq (callGraph, &derivedG);

    if (option.VeryVerbose) /* display derived sequence for each subroutine */
        displayDerivedSeq (derivedG);

    /* Graph structuring */
    structure (callGraph, derivedG);
}
```

Figure 9-11: Procedure for the Universal Decompiling Machine

The second part of the analysis is the generation of summary information on the operands of the Icode instructions and basic blocks in the graph. For each subroutine a definition-use and use-definition analysis is done; the associated chains are constructed for each instruction. While constructing these chains, dead-register elimination is performed, as described in Chapter 5, Section 5.4.1. Next, an intraprocedural live register analysis is performed for each subroutine to determine any register arguments used by the subroutine. This analysis is described in Chapter 5, Section 5.4.4. Finally, an interprocedural live register analysis is done next to determine registers that are returned by functions; the analysis is described in Chapter 5, Section 5.4.5.

Dead-register elimination determines the purpose of the `DIV` machine instructions, as this instruction is used for both quotient and remainder of operands. The following intermediate code

```
1  asgn tmp, dx:ax      ; ud(tmp) = {2,3}
2  asgn ax, tmp / bx    ; ud(ax) = {}
3  asgn dx, tmp % bx    ; ud(dx) = {4}
4  asgn [bp-2], dx      /* no further use of ax before redefinition */
```

determines that register `ax` is not used before redefinition as its use-definition chain in instruction 2 is empty. Since this definition is dead, the instruction is eliminated, hence eliminating the division of the operands, and leading to the following code:

```
1  asgn tmp, dx:ax      ; ud(tmp) = {2,3}
3  asgn dx, tmp % bx    ; ud(dx) = {4}
4  asgn [bp-2], dx
```

All instructions that had instruction 2 in their use-definition chain need to be updated to reflect the fact that the register is not used any more since it was used to define a dead register; hence, the ud() chain in instruction 1 is updated in this example, leading to the final code:

```
1  asgn tmp, dx:ax       ; ud(tmp) = {3}
3  asgn dx, tmp % bx     ; ud(dx) = {4}
4  asgn [bp-2], dx
```

The third and last part of the analysis is the usage of the use-definition chains on registers to perform extended register copy propagation, as described in Chapter 5, Section 5.4.10. This analysis removes redundant register references, determines high-level expressions, places actual parameters on the subroutine's list, and propagates argument types across subroutine calls. A temporary expression stack is used throughout the analysis to eliminate the intermediate pseudo high-level instructions `push` and `pop`.

In the previous example, forward substitution determines that the initial `DIV` instruction was used to determine the modulus between two operands (which are placed in registers `dx:ax` and `bx` in this case):

```
4  asgn [bp-2], dx:ax % bx
```

9.5.2 Control Flow Analysis

There are two parts to the control flow analyzer: the first part constructs a derived sequence of graphs for each subroutine in the call graph and calculates intervals. This sequence is used by the structuring algorithm to determine the bounds of loops and the nesting level of such loops. Once the derived sequence of graphs is built for the one subroutine, the graph is tested for reducibility; if the limit n-th order graph is not a trivial graph, the subroutine is irreducible.

The second part of the analysis is the structuring of the control flow graphs of the program. The structuring algorithm determines the bounds of loops and conditionals (2-way and n-way structures); these bounds are later used during code generation. Loops are structured by means of intervals, and their nesting level is determined by the order in which they are found in the derived sequence of graphs, as described in Chapter 6, Section 6.6.1. Pre-tested, post-tested and endless loops are determined by this algorithm. Conditionals are structured by means of a reverse traversal of the depth-first search tree of the graph; in this way nested conditionals are found first. The method for structuring 2-way and n-way conditionals is described in Chapter 6, Sections 6.6.2 and 6.6.3. This method takes into account compound Boolean conditions, and removes some nodes from the graph by storing the Boolean conditional information of two or more nodes in the one node.

9.6 The Back-end

The back-end is composed in its entirety of the C code generator. This module opens the output file and gives it an extension `.b` (b for beta), writes the program header to it, and then invokes the code generator. Once code has been generated for the complete graph, the file is closed. Figure 9-12 shows code for the back-end procedure.

```
void BackEnd (char *fileName, CALL_GRAPH *callGraph)
{ FILE   *fp;     /* Output C file  */

    /* Open output file with extension .b */
    openFile (fp, filename, ".b", "wt");
    printf ("dcc: Writing C beta file %s.b\n", fileName);

    /* Header information */
    writeHeader (fp, fileName);

    /* Process each procedure at a time */
    writeCallGraph (fileName, callGraph, fp);

    /* Close output file */
    fclose (fp);
    printf ("dcc: Finished writing C beta file\n");
}
```

Figure 9-12: Back-end Procedure

9.6.1 Code Generation

dcc implements the C code generator described in Chapter 7, Section 7.1. The program's call graph is traversed in a depth-first fashion to generate C code for the leaf subroutines first (i.e. in reverse invocation order if the graph is reducible). For each subroutine, code for the control flow graph is generated according to the structures in the graph; the bounds of loops and conditional structures have been marked in the graph by the structuring phase. Code is generated in a recursive way, so if a node is reached twice along the recursion, a `goto` jump is used to transfer control to the code associated with such a node.

Since registers that are found in leaves of an expression are given a name during code generation (i.e. after all local variables have been defined in the local variable definition section), and instructions for which code has been generated may have a label associated with them if a `goto` jump is generated later on, code cannot be generated directly to a file but needs to be stored in an intermediate data structure until the code for a complete subroutine has been generated; then it can be copied to the target output file, and the structure is reused for the next subroutine in the call graph. The data structure used by **dcc** to handle subroutine declarations and code is called a bundle. A bundle is composed of two arrays of lines, one for subroutine declarations, and the other for the subroutine code. Subroutine declarations include not only the subroutine header, but also the comments and the local variable definitions. The array of lines can grow dynamically if the initial allocated array size is too small. The definition of the bundle data structure is shown in Figure 9-13.

```
typedef struct {
    Int     numLines;   /* Number of lines in the table   */
    Int     allocLines; /* Number of lines allocated in the table */
    char    **str;      /* Table of strings */
} strTable;

typedef struct {
    strTable    decl;   /* Declarations */
    strTable    code;   /* C code       */
} bundle;
```

Figure 9-13: Bundle Data Structure Definition

The comments and error messages displayed by **dcc** are listed in Appendix E.

9.7 Results

This section presents a series of programs decompiled by **dcc**. The original programs were written in C, and compiled with Borland Turbo C under DOS. These programs make use of base type variables (i.e. byte, integer and long), and illustrate different aspects of the decompilation process. These programs were run in batch mode, generating the disassembly file `.a2`, the C file `.b`, the call graph of the program, and statistics on the intermediate code instructions. The statistics reflect the percentage of intermediate instruction reduction on all subroutines for which C is generated; subroutines which translate to assembler are not considered in the statistics. For each program, a total count on low-level and high-level instructions, and a total percentage reduction is given.

The first three programs illustrate operations on the different three base types. The original C programs have the same code, but their variables have been defined as a different type. The next four programs are benchmark programs from the Plum-Hall benchmark suite. These programs were written by Eric S. Raymond and are freely available on the network [Ray89]. These programs were modified to ask for the arguments to the program with `scanf()` rather than scanning for them in the `argv[]` command line array since arrays are not supported by **dcc**. Finally, the last three programs calculate Fibonacci numbers, compute the cyclic redundancy check (CRC) for a character, and multiply two matrixes. This last program is introduced to show how array expressions are derived from the low-level intermediate code.

9.7.1 Intops.exe

Intops is a program that computes different operations on two integer variables, and displays the final result of these variables. The disassembly C program is shown in Figure 9-14, the decompiled C program in Figure 9-15, and the initial C program in Figure 9-16. The program has the following call graph:

```
main
   printf
```

As can be seen in the disassembly of the program, the second variable was placed in register `si`, and the first variable was placed on the stack at offset `-2`. Synthetic instructions were generated by the parser for the `IDIV` machine instruction; this instruction was used as a division in one case, and as a modulus in the other. The intermediate code makes use of the temporary register `tmp`, as previously explained in Section 9.3.2; this register is eliminated during data flow analysis. For each operation, the operands of the instruction are moved to registers, the operation is performed on registers, and the result is placed back on the variables. There are no control structures in the program. The idioms and data flow analyses reduce the number of intermediate instructions by 77.78%, as shown in Figure 9-17.

```
       main  PROC  NEAR
000 0002FA 55                     PUSH           bp
001 0002FB 8BEC                   MOV            bp, sp
002 0002FD 83EC02                 SUB            sp, 2
003 000300 56                     PUSH           si
004 000301 C746FEFF00             MOV   word ptr [bp-2], 0FFh
005 000306 BE8F00                 MOV            si, 8Fh
006 000309 8B46FE                 MOV            ax, [bp-2]
007 00030C 03C6                   ADD            ax, si
008 00030E 8BF0                   MOV            si, ax
009 000310 8B46FE                 MOV            ax, [bp-2]
010 000313 2BC6                   SUB            ax, si
011 000315 8946FE                 MOV            [bp-2], ax
012 000318 8B46FE                 MOV            ax, [bp-2]
013 00031B F7E6                   MUL            si
014 00031D 8946FE                 MOV            [bp-2], ax
015 000320 8BC6                   MOV            ax, si
016 000322 99                     CWD
017                               MOV            tmp, dx:ax       ;Synthetic inst
018 000323 F77EFE                 IDIV  word ptr [bp-2]
019                               MOD   word ptr [bp-2]           ;Synthetic inst
020 000326 8BF0                   MOV            si, ax
021 000328 8BC6                   MOV            ax, si
022 00032A 99                     CWD
023                               MOV            tmp, dx:ax       ;Synthetic inst
024 00032B F77EFE                 IDIV  word ptr [bp-2]
025                               MOD   word ptr [bp-2]           ;Synthetic inst
026 00032E 8BF2                   MOV            si, dx
027 000330 8B46FE                 MOV            ax, [bp-2]
028 000333 B105                   MOV            cl, 5
029 000335 D3E0                   SHL            ax, cl
030 000337 8946FE                 MOV            [bp-2], ax
031 00033A 8BC6                   MOV            ax, si
032 00033C 8A4EFE                 MOV            cl, [bp-2]
033 00033F D3F8                   SAR            ax, cl
034 000341 8BF0                   MOV            si, ax
035 000343 56                     PUSH           si
036 000344 FF76FE                 PUSH  word ptr [bp-2]
037 000347 B89401                 MOV            ax, 194h
038 00034A 50                     PUSH           ax
039 00034B E8AC06                 CALL  near ptr printf
040 00034E 83C406                 ADD            sp, 6
041 000351 5E                     POP            si
042 000352 8BE5                   MOV            sp, bp
043 000354 5D                     POP            bp
044 000355 C3                     RET
       main  ENDP
```

Figure 9-14: Intops.a2

```
/*
 * Input file : intops.exe
 * File type : EXE
 */

#include "dcc.h"


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;

    loc1 = 255;
    loc2 = 143;
    loc2 = (loc1 + loc2);
    loc1 = (loc1 - loc2);
    loc1 = (loc1 * loc2);
    loc2 = (loc2 / loc1);
    loc2 = (loc2 % loc1);
    loc1 = (loc1 << 5);
    loc2 = (loc2 >> loc1);
    printf ("a = %d, b = %d\n", loc1, loc2);
}
```

Figure 9-15: Intops.b

```
#define TYPE int

main()
{ TYPE a, b;

    a = 255;
    b = 143;
    b = a + b;
    a = a - b;
    a = a * b;
    b = b / a;
    b = b % a;
    a = a << 5;
    b = b >> a;
    printf ("a = %d, b = %d\n", a, b);
}
```

Figure 9-16: Intops.c

Subroutine	Low-level	High-level	% Reduction
main	45	10	77.78
total	45	10	77.78

Figure 9-17: Intops Statistics

9.7.2 Byteops.exe

Byteops is a similar program to intops, with the difference that the two variables are bytes rather than integers. The disassembly program is shown in Figure 9-18, the decompiled C version in Figure 9-19, and the initial C program in Figure 9-20. The program has the following call graph:

```
main
   printf
```

As can be seen in the disassembly of the program, the local variables are placed on the stack at offsets `-1` and `-2`. There are 22.41% more instructions in this program when compared against the intops.a2 program since some machine instructions such as `IDIV` take word registers as operands rather than byte registers; hence, the byte registers are either padded or sign-extended to form a word register. The final number of high-level instructions is the same in both programs, hence the reduction in the number of intermediate instructions is greater in this program. It reached 82.76%, as shown in Figure 9-21.

```
       main  PROC  NEAR
000 0002FA 55                 PUSH            bp
001 0002FB 8BEC               MOV             bp, sp
002 0002FD 83EC02             SUB             sp, 2
003 000300 C646FEFF           MOV   byte ptr  [bp-2], 0FFh
004 000304 C646FF8F           MOV   byte ptr  [bp-1], 8Fh
005 000308 8A46FE             MOV             al, [bp-2]
006 00030B 0246FF             ADD             al, [bp-1]
007 00030E 8846FF             MOV             [bp-1], al
008 000311 8A46FE             MOV             al, [bp-2]
009 000314 2A46FF             SUB             al, [bp-1]
010 000317 8846FE             MOV             [bp-2], al
011 00031A 8A46FE             MOV             al, [bp-2]
012 00031D B400               MOV             ah, 0
013 00031F 8A56FF             MOV             dl, [bp-1]
014 000322 B600               MOV             dh, 0
015 000324 F7E2               MUL             dx
016 000326 8846FE             MOV             [bp-2], al
017 000329 8A46FF             MOV             al, [bp-1]
018 00032C B400               MOV             ah, 0
019 00032E 8A56FE             MOV             dl, [bp-2]
020 000331 B600               MOV             dh, 0
021 000333 8BDA               MOV             bx, dx
022 000335 99                 CWD
023                           MOV             tmp, dx:ax      ;Synthetic inst
024 000336 F7FB               IDIV            bx
025                           MOD             bx              ;Synthetic inst
026 000338 8846FF             MOV             [bp-1], al
027 00033B 8A46FF             MOV             al, [bp-1]
028 00033E B400               MOV             ah, 0
029 000340 8A56FE             MOV             dl, [bp-2]
030 000343 B600               MOV             dh, 0
031 000345 8BDA               MOV             bx, dx
032 000347 99                 CWD
033                           MOV             tmp, dx:ax      ;Synthetic inst
034 000348 F7FB               IDIV            bx
035                           MOD             bx              ;Synthetic inst
036 00034A 8856FF             MOV             [bp-1], dl
037 00034D 8A46FE             MOV             al, [bp-2]
038 000350 B105               MOV             cl, 5
039 000352 D2E0               SHL             al, cl
040 000354 8846FE             MOV             [bp-2], al
041 000357 8A46FF             MOV             al, [bp-1]
042 00035A 8A4EFE             MOV             cl, [bp-2]
043 00035D D2E8               SHR             al, cl
044 00035F 8846FF             MOV             [bp-1], al
```

Figure 9-18: Byteops.a2

```
045 000362 8A46FF               MOV           al, [bp-1]
046 000365 B400                 MOV           ah, 0
047 000367 50                   PUSH          ax
048 000368 8A46FE               MOV           al, [bp-2]
049 00036B B400                 MOV           ah, 0
050 00036D 50                   PUSH          ax
051 00036E B89401               MOV           ax, 194h
052 000371 50                   PUSH          ax
053 000372 E8AB06               CALL  near ptr printf
054 000375 83C406               ADD           sp, 6
055 000378 8BE5                 MOV           sp, bp
056 00037A 5D                   POP           bp
057 00037B C3                   RET
       main  ENDP
```

Figure 9-18: Byteops.a2 – Continued

```
/*
 * Input file : byteops.exe
 * File type : EXE
 */

#include "dcc.h"


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;

    loc1 = 255;
    loc2 = 143;
    loc2 = (loc1 + loc2);
    loc1 = (loc1 - loc2);
    loc1 = (loc1 * loc2);
    loc2 = (loc2 / loc1);
    loc2 = (loc2 % loc1);
    loc1 = (loc1 << 5);
    loc2 = (loc2 >> loc1);
    printf ("a = %d, b = %d\n", loc1, loc2);
}
```

Figure 9-19: Byteops.b

```
#define TYPE unsigned char

main()
{ TYPE a, b;

    a = 255;
    b = 143;
    b = a + b;
    a = a - b;
    a = a * b;
    b = b / a;
    b = b % a;
    a = a << 5;
    b = b >> a;
    printf ("a = %d, b = %d\n", a, b);
}
```

Figure 9-20: Byteops.c

Subroutine	Low-level	High-level	% Reduction
main	58	10	82.76
total	58	10	82.76

Figure 9-21: Byteops Statistics

9.7.3 Longops.exe

The longops programs is similar to the intops and byteops programs, but makes use of two long variables. The disassembly program is shown in Figure 9-22, the decompiled C program in Figure 9-23, and the initial C program in Figure 9-24. The program has the following call graph:

```
main
   LXMUL@
   LDIV@
   LMOD@
   LXLSH@
   LXRSH@
   printf
```

Operations performed on long variables make use of idioms and run-time support routines of the compiler. In this program, long addition and subtraction are performed by the idioms of Chapter 4, Section 4.2.1, and the run-time routines `LXMUL@`, `LDIV@`, `LMOD@`, `LXLSH@`, and `LXRSH@` are used for long multiplication, division, modulus, left-shift, and right-shift accordingly. From these run-time routines, long multiplication, left-shift, and right-shift are translatable into C; macros are used to access the low or high part of a variable in some cases. The division and modulus routines are untranslatable into C, so assembler is generated for them. The long variables are placed on the stack at offsets `-4` and `-8` (see `main` subroutine). The `main` program has 28.57% more instructions than the intops program, and 7.9% more instructions than the byteops program. The increase in the number of instructions has two causes: first, the transfer of long variables to registers now takes two instructions rather than one (i.e. the high and low part are transfered to different registers), and second, the subroutine call instructions to run-time support routines. The final decompiled `main` program still generates the same number of high-level instructions as in the previous two programs, with a reduction in the number of intermediate instructions of 84.13%, as shown in Figure 9-25. Overall, the reduction in the number of instructions is 58.97%, which is low due to the run-time routines that were translated to C, which did not make use of a lot of register movement since the arguments were in registers and these routines were initially written in assembler.

```
       LXRSH@  PROC  FAR
000 001269 80F910             CMP          cl, 10h
001 00126C 7310               JAE          L1
002 00126E 8BDA               MOV          bx, dx
003 001270 D3E8               SHR          ax, cl
004 001272 D3FA               SAR          dx, cl
005 001274 F6D9               NEG          cl
006 001276 80C110             ADD          cl, 10h
007 001279 D3E3               SHL          bx, cl
008 00127B 0BC3               OR           ax, bx
009 00127D CB                 RETF
010 00127E 80E910        L1:  SUB          cl, 10h
011 001281 8BC2               MOV          ax, dx
012 001283 99                 CWD
013 001284 D3F8               SAR          ax, cl
014 001286 CB                 RETF
       LXRSH@  ENDP

       LXLSH@  PROC  FAR
000 001287 80F910             CMP          cl, 10h
001 00128A 7310               JAE          L2
002 00128C 8BD8               MOV          bx, ax
003 00128E D3E0               SHL          ax, cl
004 001290 D3E2               SHL          dx, cl
005 001292 F6D9               NEG          cl
006 001294 80C110             ADD          cl, 10h
007 001297 D3EB               SHR          bx, cl
008 001299 0BD3               OR           dx, bx
009 00129B CB                 RETF
010 00129C 80E910        L2:  SUB          cl, 10h
011 00129F 8BD0               MOV          dx, ax
012 0012A1 33C0               XOR          ax, ax
013 0012A3 D3E2               SHL          dx, cl
014 0012A5 CB                 RETF
       LXLSH@  ENDP

       LMOD@  PROC  FAR
000 0011CF B90200             MOV          cx, 2
002 0011D7 55                 PUSH         bp
003 0011D8 56                 PUSH         si
004 0011D9 57                 PUSH         di
005 0011DA 8BEC               MOV          bp, sp
006 0011DC 8BF9               MOV          di, cx
007 0011DE 8B460A             MOV          ax, [bp+0Ah]
008 0011E1 8B560C             MOV          dx, [bp+0Ch]
009 0011E4 8B5E0E             MOV          bx, [bp+0Eh]
```

Figure 9-22: Longops.a2

```
010 0011E7 8B4E10                 MOV           cx, [bp+10h]
011 0011EA 0BC9                   OR            cx, cx
012 0011EC 7508                   JNE           L3
013 0011EE 0BD2                   OR            dx, dx
014 0011F0 7469                   JE            L4
015 0011F2 0BDB                   OR            bx, bx
016 0011F4 7465                   JE            L4
017 0011F6 F7C70100          L3:  TEST          di, 1
018 0011FA 751C                   JNE           L5
019 0011FC 0BD2                   OR            dx, dx
020 0011FE 790A                   JNS           L6
021 001200 F7DA                   NEG           dx
022 001202 F7D8                   NEG           ax
023 001204 83DA00                 SBB           dx, 0
024 001207 83CF0C                 OR            di, 0Ch
025 00120A 0BC9              L6:  OR            cx, cx
026 00120C 790A                   JNS           L5
027 00120E F7D9                   NEG           cx
028 001210 F7DB                   NEG           bx
029 001212 83D900                 SBB           cx, 0
030 001215 83F704                 XOR           di, 4
031 001218 8BE9              L5:  MOV           bp, cx
032 00121A B92000                 MOV           cx, 20h
033 00121D 57                     PUSH          di
034 00121E 33FF                   XOR           di, di
035 001220 33F6                   XOR           si, si
036 001222 D1E0              L7:  SHL           ax, 1
037 001224 D1D2                   RCL           dx, 1
038 001226 D1D6                   RCL           si, 1
039 001228 D1D7                   RCL           di, 1
040 00122A 3BFD                   CMP           di, bp
041 00122C 720B                   JB            L8
042 00122E 7704                   JA            L9
043 001230 3BF3                   CMP           si, bx
044 001232 7205                   JB            L8
045 001234 2BF3              L9:  SUB           si, bx
046 001236 1BFD                   SBB           di, bp
047 001238 40                     INC           ax
048 001239 E2E7              L8:  LOOP          L7
049 00123B 5B                     POP           bx
050 00123C F7C30200               TEST          bx, 2
051 001240 7406                   JE            L10
052 001242 8BC6                   MOV           ax, si
053 001244 8BD7                   MOV           dx, di
054 001246 D1EB                   SHR           bx, 1
```

Figure 9-22: Longops.a2 – Continued

```
055 001248 F7C30400       L10: TEST           bx, 4
056 00124C 7407                JE             L11
057 00124E F7DA                NEG            dx
058 001250 F7D8                NEG            ax
059 001252 83DA00              SBB            dx, 0
060 001255 5F             L11: POP            di
061 001256 5E                  POP            si
062 001257 5D                  POP            bp
063 001258 CA0800              RETF           8
064                       L4:  MOV            tmp, dx:ax      ;Synthetic inst
065 00125B F7F3                DIV            bx
066                            MOD            bx              ;Synthetic inst
067 00125D F7C70200            TEST           di, 2
068 001261 7402                JE             L12
069 001263 8BC2                MOV            ax, dx
070 001265 33D2           L12: XOR            dx, dx
071 001267 EBEC                JMP            L11
       LMOD@  ENDP

       LDIV@  PROC  FAR
000 0011C6 33C9                XOR            cx, cx
002 0011D7 55                  PUSH           bp
003 0011D8 56                  PUSH           si
004 0011D9 57                  PUSH           di
005 0011DA 8BEC                MOV            bp, sp
006 0011DC 8BF9                MOV            di, cx
007 0011DE 8B460A              MOV            ax, [bp+0Ah]
008 0011E1 8B560C              MOV            dx, [bp+0Ch]
009 0011E4 8B5E0E              MOV            bx, [bp+0Eh]
010 0011E7 8B4E10              MOV            cx, [bp+10h]
011 0011EA 0BC9                OR             cx, cx
012 0011EC 7508                JNE            L13
013 0011EE 0BD2                OR             dx, dx
014 0011F0 7469                JE             L14
015 0011F2 0BDB                OR             bx, bx
016 0011F4 7465                JE             L14
017 0011F6 F7C70100       L13: TEST           di, 1
018 0011FA 751C                JNE            L15
019 0011FC 0BD2                OR             dx, dx
020 0011FE 790A                JNS            L16
021 001200 F7DA                NEG            dx
022 001202 F7D8                NEG            ax
023 001204 83DA00              SBB            dx, 0
024 001207 83CF0C              OR             di, 0Ch
025 00120A 0BC9           L16: OR             cx, cx
```

Figure 9-22: Longops.a2 – Continued

```
026 00120C 790A                JNS            L15
027 00120E F7D9                NEG            cx
028 001210 F7DB                NEG            bx
029 001212 83D900              SBB            cx, 0
030 001215 83F704              XOR            di, 4
031 001218 8BE9           L15: MOV            bp, cx
032 00121A B92000              MOV            cx, 20h
033 00121D 57                  PUSH           di
034 00121E 33FF                XOR            di, di
035 001220 33F6                XOR            si, si
036 001222 D1E0           L17: SHL            ax, 1
037 001224 D1D2                RCL            dx, 1
038 001226 D1D6                RCL            si, 1
039 001228 D1D7                RCL            di, 1
040 00122A 3BFD                CMP            di, bp
041 00122C 720B                JB             L18
042 00122E 7704                JA             L19
043 001230 3BF3                CMP            si, bx
044 001232 7205                JB             L18
045 001234 2BF3           L19: SUB            si, bx
046 001236 1BFD                SBB            di, bp
047 001238 40                  INC            ax
048 001239 E2E7           L18: LOOP           L17
049 00123B 5B                  POP            bx
050 00123C F7C30200            TEST           bx, 2
051 001240 7406                JE             L20
052 001242 8BC6                MOV            ax, si
053 001244 8BD7                MOV            dx, di
054 001246 D1EB                SHR            bx, 1
055 001248 F7C30400       L20: TEST           bx, 4
056 00124C 7407                JE             L21
057 00124E F7DA                NEG            dx
058 001250 F7D8                NEG            ax
059 001252 83DA00              SBB            dx, 0
060 001255 5F             L21: POP            di
061 001256 5E                  POP            si
062 001257 5D                  POP            bp
063 001258 CA0800              RETF           8
064                       L14: MOV            tmp, dx:ax      ;Synthetic inst
065 00125B F7F3                DIV            bx
066                            MOD            bx              ;Synthetic inst
067 00125D F7C70200            TEST           di, 2
068 001261 7402                JE             L22
069 001263 8BC2                MOV            ax, dx
070 001265 33D2           L22: XOR            dx, dx
071 001267 EBEC                JMP            L21
       LDIV@  ENDP
```

Figure 9-22: Longops.a2 – Continued

```
       LXMUL@  PROC  FAR
000 0009C3 56                  PUSH          si
001                            MOV           tmp, ax         ;Synthetic inst
002                            MOV           ax, si          ;Synthetic inst
003                            MOV           si, tmp         ;Synthetic inst
004                            MOV           tmp, ax         ;Synthetic inst
005                            MOV           ax, dx          ;Synthetic inst
006                            MOV           dx, tmp         ;Synthetic inst
007 0009C6 85C0                TEST          ax, ax
008 0009C8 7402                JE            L23
009 0009CA F7E3                MUL           bx
010                       L23: MOV           tmp, ax         ;Synthetic inst
011                            MOV           ax, cx          ;Synthetic inst
012                            MOV           cx, tmp         ;Synthetic inst
013 0009CD 85C0                TEST          ax, ax
014 0009CF 7404                JE            L24
015 0009D1 F7E6                MUL           si
016 0009D3 03C8                ADD           cx, ax
017                       L24: MOV           tmp, ax         ;Synthetic inst
018                            MOV           ax, si          ;Synthetic inst
019                            MOV           si, tmp         ;Synthetic inst
020 0009D6 F7E3                MUL           bx
021 0009D8 03D1                ADD           dx, cx
022 0009DA 5E                  POP           si
023 0009DB CB                  RETF
       LXMUL@  ENDP

       main  PROC  NEAR
000 0002FA 55                  PUSH          bp
001 0002FB 8BEC                MOV           bp, sp
002 0002FD 83EC08              SUB           sp, 8
003 000300 C746FA0000          MOV   word ptr [bp-6], 0
004 000305 C746F8FF00          MOV   word ptr [bp-8], 0FFh
005 00030A C746FE0000          MOV   word ptr [bp-2], 0
006 00030F C746FC8F00          MOV   word ptr [bp-4], 8Fh
007 000314 8B56FA              MOV           dx, [bp-6]
008 000317 8B46F8              MOV           ax, [bp-8]
009 00031A 0346FC              ADD           ax, [bp-4]
010 00031D 1356FE              ADC           dx, [bp-2]
011 000320 8956FE              MOV           [bp-2], dx
012 000323 8946FC              MOV           [bp-4], ax
013 000326 8B56FA              MOV           dx, [bp-6]
014 000329 8B46F8              MOV           ax, [bp-8]
015 00032C 2B46FC              SUB           ax, [bp-4]
016 00032F 1B56FE              SBB           dx, [bp-2]
017 000332 8956FA              MOV           [bp-6], dx
```

Figure 9-22: Longops.a2 – Continued

```
018 000335 8946F8             MOV              [bp-8], ax
019 000338 8B56FA             MOV              dx, [bp-6]
020 00033B 8B46F8             MOV              ax, [bp-8]
021 00033E 8B4EFE             MOV              cx, [bp-2]
022 000341 8B5EFC             MOV              bx, [bp-4]
023 000344 9AC3081000         CALL    far ptr LXMUL@
024 000349 8956FA             MOV              [bp-6], dx
025 00034C 8946F8             MOV              [bp-8], ax
026 00034F FF76FA             PUSH   word ptr [bp-6]
027 000352 FF76F8             PUSH   word ptr [bp-8]
028 000355 FF76FE             PUSH   word ptr [bp-2]
029 000358 FF76FC             PUSH   word ptr [bp-4]
030 00035B 9AC6101000         CALL    far ptr LDIV@
031 000360 8956FE             MOV              [bp-2], dx
032 000363 8946FC             MOV              [bp-4], ax
033 000366 FF76FA             PUSH   word ptr [bp-6]
034 000369 FF76F8             PUSH   word ptr [bp-8]
035 00036C FF76FE             PUSH   word ptr [bp-2]
036 00036F FF76FC             PUSH   word ptr [bp-4]
037 000372 9ACF101000         CALL    far ptr LMOD@
038 000377 8956FE             MOV              [bp-2], dx
039 00037A 8946FC             MOV              [bp-4], ax
040 00037D 8B56FA             MOV              dx, [bp-6]
041 000380 8B46F8             MOV              ax, [bp-8]
042 000383 B105               MOV              cl, 5
043 000385 9A87111000         CALL    far ptr LXLSH@
044 00038A 8956FA             MOV              [bp-6], dx
045 00038D 8946F8             MOV              [bp-8], ax
046 000390 8B56FE             MOV              dx, [bp-2]
047 000393 8B46FC             MOV              ax, [bp-4]
048 000396 8A4EF8             MOV              cl, [bp-8]
049 000399 9A69111000         CALL    far ptr LXRSH@
050 00039E 8956FE             MOV              [bp-2], dx
051 0003A1 8946FC             MOV              [bp-4], ax
052 0003A4 FF76FE             PUSH   word ptr [bp-2]
053 0003A7 FF76FC             PUSH   word ptr [bp-4]
054 0003AA FF76FA             PUSH   word ptr [bp-6]
055 0003AD FF76F8             PUSH   word ptr [bp-8]
056 0003B0 B89401             MOV              ax, 194h
057 0003B3 50                 PUSH             ax
058 0003B4 E8C406             CALL   near ptr printf
059 0003B7 83C40A             ADD              sp, 0Ah
060 0003BA 8BE5               MOV              sp, bp
061 0003BC 5D                 POP              bp
062 0003BD C3                 RET
       main  ENDP
```

Figure 9-22: Longops.a2 – Continued

```
/*
 * Input file : longops.exe
 * File type : EXE
 */

#include "dcc.h"


long LXMUL@ (long arg0, long arg1)
/* Uses register arguments:
 *     arg0 = dx:ax.
 *     arg1 = cx:bx.
 * Runtime support routine of the compiler.
 */
{
int loc1;
int loc2; /* tmp */

    loc2 = LO(arg0);
    LO(arg0) = loc1;
    loc1 = loc2;
    loc2 = LO(arg0);
    LO(arg0) = HI(arg0);
    if ((LO(arg0) & LO(arg0)) != 0) {
        LO(arg0) = (LO(arg0) * LO(arg1));
    }
    loc2 = LO(arg0);
    LO(arg0) = HI(arg1);
    HI(arg1) = loc2;
    if ((LO(arg0) & LO(arg0)) != 0) {
        LO(arg0) = (LO(arg0) * loc1);
        HI(arg1) = (HI(arg1) + LO(arg0));
    }
    loc2 = LO(arg0);
    LO(arg0) = loc1;
    loc1 = loc2;
    arg0 = (LO(arg0) * LO(arg1));
    HI(arg0) = (HI(arg0) + HI(arg1));
    return (arg0);
}
```

Figure 9-23: Longops.b

```
long LDIV@ (long arg0, long arg2)
/* Takes 8 bytes of parameters.
 * Runtime support routine of the compiler.
 * Untranslatable routine.  Assembler provided.
 * Return value in registers dx:ax.
 * Pascal calling convention.
 */
{
    /* disassembly code here */
}


long LMOD@ (long arg0, long arg2)
/* Takes 8 bytes of parameters.
 * Runtime support routine of the compiler.
 * Untranslatable routine.  Assembler provided.
 * Return value in registers dx:ax.
 * Pascal calling convention.
 */
{
    /* disassembly code here */
}


long LXLSH@ (long arg0, char arg1)
/* Uses register arguments:
 *     arg0 = dx:ax.
 *     arg1 = cl.
 * Runtime support routine of the compiler.
 */
{
int loc1; /* bx */

    if (arg1 < 16) {
        loc1 = LO(arg0);
        LO(arg0) = (LO(arg0) << arg1);
        HI(arg0) = (HI(arg0) << arg1);
        HI(arg0) = (HI(arg0) | (loc1 >> (!arg1 + 16)));
        return (arg0);
    }
    else {
        HI(arg0) = LO(arg0);
        LO(arg0) = 0;
        HI(arg0) = (HI(arg0) << (arg1 - 16));
        return (arg0);
    }
}
```

Figure 9-23: Longops.b – Continued

```
long LXRSH@ (long arg0, char arg1)
/* Uses register arguments:
 *     arg0 = dx:ax.
 *     arg1 = cl.
 * Runtime support routine of the compiler.
 */
{
int loc1; /* bx */

    if (arg1 < 16) {
        loc1 = HI(arg0);
        LO(arg0) = (LO(arg0) >> arg1);
        HI(arg0) = (HI(arg0) >> arg1);
        LO(arg0) = (LO(arg0) | (loc1 << (!arg1 + 16)));
        return (arg0);
    }
    else {
        arg0 = HI(arg0);
        LO(arg0) = (LO(arg0) >> (arg1 - 16));
        return (arg0);
    }
}


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
long loc1;
long loc2;

    loc2 = 255;
    loc1 = 143;
    loc1 = (loc2 + loc1);
    loc2 = (loc2 - loc1);
    loc2 = LXMUL@ (loc2, loc1);
    loc1 = LDIV@ (loc1, loc2);
    loc1 = LMOD@ (loc1, loc2);
    loc2 = LXLSH@ (loc2, 5);
    loc1 = LXRSH@ (loc1, loc1);
    printf ("a = %ld, b = %ld\n", loc2, loc1);
}
```

Figure 9-23: Longops.b – Continued

```
#define TYPE long

main()
{ TYPE a, b;

    a = 255;
    b = 143;
    b = a + b;
    a = a - b;
    a = a * b;
    b = b / a;
    b = b % a;
    a = a << 5;
    b = b >> a;
    printf ("a = %ld, b = %ld\n", a, b);
}
```

Figure 9-24: Longops.c

Subroutine	Low-level	High-level	% Reduction
LXMUL@	24	19	20.83
LDIV@	72	-	-
LMOD@	72	-	-
LXLSH@	15	10	33.33
LXRSH@	15	9	40.00
main	63	10	84.13
total	117	48	58.97

Figure 9-25: Longops Statistics

9.7.4 Benchsho.exe

Benchsho is a program from the Plum-Hall benchmark suite, which benchmarks short integers. The program makes use of two long variables to iterate through the loop, and three (short) integer variables to execute 1000 operations. The disassembly program is shown in Figure 9-27, the decompiled C program in Figure 9-28, and the initial C program in Figure 9-29. The program has the following call graph:

```
main
    scanf
    printf
```

As seen in the disassembly of the program, the long variables are located in the stack at offsets `-4` and `-8`, and the integer variables are located at offsets `-14`, `-12`, and `-10`. The final C code makes use of the integer variable `loc6` to hold the result of a Boolean expression (i.e. 0 or 1) and assign it to the corresponding variable. This Boolean variable is a register variable (register `ax`) and could have been eliminated from the code with further analysis of the control flow graph, in a similar way to the structuring of compound conditions.

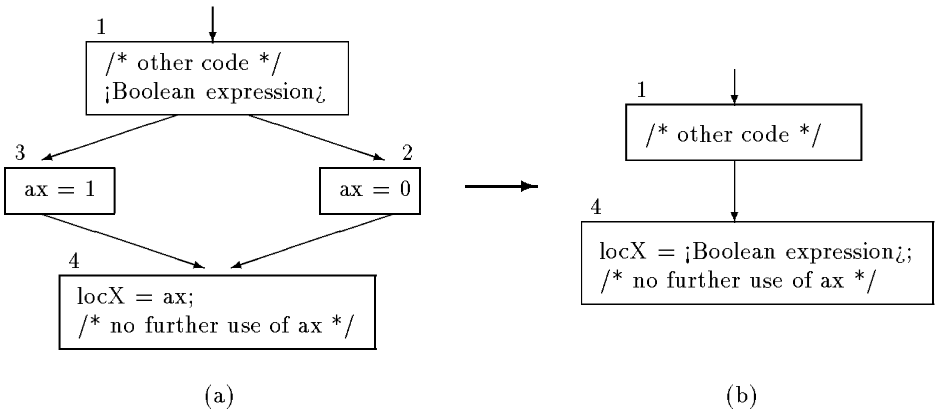


Figure 9-26: Control Flow Graph for Boolean Assignment

For example, graph (a) in Figure 9-26 can be reduced to graph (b) if the following conditions are satisfied:

1. Node 1 is a 2-way node.
2. Nodes 2 and 3 have one in-edge from node 1 only, and lead to a common node 4.
3. Nodes 2 and 3 have one instruction only. This instruction assigns 0 and 1 respectively to a register.
4. Node 4 assigns the register of nodes 2 and 3 to a local variable. The register is not further used before redefinition in the program.

Since the register is used only once to store the intermediate result of a Boolean expression evaluation, it is eliminated from the final code by assigning the Boolean expression to the target variable. This transformation not only removes the involved register, but also the two nodes that assigned a value to it (i.e. nodes 2 and 3 in the graph of Figure 9-26).

It is clear that the two Boolean assignments of Figure 9-28 can be transformed into the following code:

```
loc1 = (loc2 == loc3);
/* other code */
loc1 = (loc2 > loc3);
```

which would make the final C program an exact decompilation of the original C program. Without this transformation, the generated C code is functionally equivalent to the initial C code, and structurally equivalent to the decompiled graph. Since the graph of a Boolean assignment is structured by nature, the non-implementation of this transformation does not generate unstructured code in any way, unlike the case of compound conditions, which are unstructured graphs by nature that are transformed into structured graphs.

Without the graph optimization, the final decompiled code generated by **dcc** produces a 75.25% reduction on the number of intermediate instructions, as shown in Figure 9-30. For each Boolean assignment of the initial C code, there are three extra instructions due to the use of a temporary local variable (`loc6` in this case).

```
       main  PROC  NEAR
000 0002FA 55                    PUSH          bp
001 0002FB 8BEC                  MOV           bp, sp
002 0002FD 83EC0E                SUB           sp, 0Eh
003 000300 8D46FC                LEA           ax, [bp-4]
004 000303 50                    PUSH          ax
005 000304 B89401                MOV           ax, 194h
006 000307 50                    PUSH          ax
007 000308 E8E914                CALL  near ptr scanf
008 00030B 59                    POP           cx
009 00030C 59                    POP           cx
010 00030D FF76FE                PUSH  word ptr [bp-2]
011 000310 FF76FC                PUSH  word ptr [bp-4]
012 000313 B89801                MOV           ax, 198h
013 000316 50                    PUSH          ax
014 000317 E8510C                CALL  near ptr printf
015 00031A 83C406                ADD           sp, 6
016 00031D 8D46F2                LEA           ax, [bp-0Eh]
017 000320 50                    PUSH          ax
018 000321 B8B201                MOV           ax, 1B2h
019 000324 50                    PUSH          ax
020 000325 E8CC14                CALL  near ptr scanf
021 000328 59                    POP           cx
022 000329 59                    POP           cx
023 00032A 8D46F4                LEA           ax, [bp-0Ch]
024 00032D 50                    PUSH          ax
025 00032E B8B601                MOV           ax, 1B6h
026 000331 50                    PUSH          ax
027 000332 E8BF14                CALL  near ptr scanf
028 000335 59                    POP           cx
029 000336 59                    POP           cx
030 000337 C746FA0000            MOV   word ptr [bp-6], 0
031 00033C C746F80100            MOV   word ptr [bp-8], 1

033 0003BD 8B56FA           L1:  MOV           dx, [bp-6]
034 0003C0 8B46F8                MOV           ax, [bp-8]
035 0003C3 3B56FE                CMP           dx, [bp-2]
036 0003C6 7D03                  JGE           L2

038 000344 C746F60100       L3:  MOV   word ptr [bp-0Ah], 1
040 0003AF 837EF628         L4:  CMP   word ptr [bp-0Ah], 28h
041 0003B3 7E96                  JLE           L5
042 0003B5 8346F801              ADD   word ptr [bp-8], 1
043 0003B9 8356FA00              ADC   word ptr [bp-6], 0
044                              JMP           L1               ;Synthetic inst
```

Figure 9-27: Benchsho.a2

```
045 00034B 8B46F2          L5:  MOV               ax, [bp-0Eh]
046 00034E 0346F4               ADD               ax, [bp-0Ch]
047 000351 0346F6               ADD               ax, [bp-0Ah]
048 000354 8946F2               MOV               [bp-0Eh], ax
049 000357 8B46F2               MOV               ax, [bp-0Eh]
050 00035A D1F8                 SAR               ax, 1
051 00035C 8946F4               MOV               [bp-0Ch], ax
052 00035F 8B46F4               MOV               ax, [bp-0Ch]
053 000362 BB0A00               MOV               bx, 0Ah
054 000365 99                   CWD
055                             MOV               tmp, dx:ax       ;Synthetic inst
056 000366 F7FB                 IDIV              bx
057                             MOD               bx               ;Synthetic inst
058 000368 8956F2               MOV               [bp-0Eh], dx
059 00036B 8B46F4               MOV               ax, [bp-0Ch]
060 00036E 3B46F6               CMP               ax, [bp-0Ah]
061 000371 7505                 JNE               L6
062 000373 B80100               MOV               ax, 1

064 00037A 8946F2          L7:  MOV               [bp-0Eh], ax
065 00037D 8B46F2               MOV               ax, [bp-0Eh]
066 000380 0B46F6               OR                ax, [bp-0Ah]
067 000383 8946F4               MOV               [bp-0Ch], ax
068 000386 8B46F4               MOV               ax, [bp-0Ch]
069 000389 F7D8                 NEG               ax
070 00038B 1BC0                 SBB               ax, ax
071 00038D 40                   INC               ax
072 00038E 8946F2               MOV               [bp-0Eh], ax
073 000391 8B46F2               MOV               ax, [bp-0Eh]
074 000394 0346F6               ADD               ax, [bp-0Ah]
075 000397 8946F4               MOV               [bp-0Ch], ax
076 00039A 8B46F4               MOV               ax, [bp-0Ch]
077 00039D 3B46F6               CMP               ax, [bp-0Ah]
078 0003A0 7E05                 JLE               L8
079 0003A2 B80100               MOV               ax, 1

081 0003A9 8946F2          L9:  MOV               [bp-0Eh], ax
082 0003AC FF46F6               INC     word ptr  [bp-0Ah]
083                             JMP               L4               ;Synthetic inst

084 0003A7 33C0            L8:  XOR               ax, ax
085                             JMP               L9               ;Synthetic inst
086 000378 33C0            L6:  XOR               ax, ax
087                             JMP               L7               ;Synthetic inst
088 0003CB 7F08            L2:  JG                L10
089 0003CD 3B46FC               CMP               ax, [bp-4]
090 0003D0 7703                 JA                L10
```

Figure 9-27: Benchsho.a2 – Continued

```
092 0003D5 FF76F2         L10: PUSH  word ptr [bp-0Eh]
093 0003D8 B8BA01              MOV            ax, 1BAh
094 0003DB 50                  PUSH           ax
095 0003DC E88C0B              CALL  near ptr printf
096 0003DF 59                  POP            cx
097 0003E0 59                  POP            cx
098 0003E1 8BE5                MOV            sp, bp
099 0003E3 5D                  POP            bp
100 0003E4 C3                  RET
       main  ENDP
```

Figure 9-27: Benchsho.a2 – Continued

```
/*
 * Input file : benchsho.exe
 * File type : EXE
 */
#include "dcc.h"

void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{ int loc1; int loc2; int loc3;
  long loc4; long loc5; int loc6; /* ax */

    scanf ("%ld", &loc5);
    printf ("executing %ld iterations\n", loc5);
    scanf ("%ld", &loc1);
    scanf ("%ld", &loc2);
    loc4 = 1;
    while ((loc4 <= loc5)) {
        loc3 = 1;
        while ((loc3 <= 40)) {
            loc1 = ((loc1 + loc2) + loc3);
            loc2 = (loc1 >> 1);
            loc1 = (loc2 % 10);
            if (loc2 == loc3) {
                loc6 = 1;
            }
            else {
                loc6 = 0;
            }
            loc1 = loc6;
            loc2 = (loc1 | loc3);
            loc1 = !loc2;
            loc2 = (loc1 + loc3);
            if (loc2 > loc3) {
                loc6 = 1;
            }
            else {
                loc6 = 0;
            }
            loc1 = loc6;
            loc3 = (loc3 + 1);
        }
        loc4 = (loc4 + 1);
    }
    printf ("a=%d\n", loc1);
}
```

Figure 9-28: Benchsho.b

```
/* benchsho - benchmark for  short  integers
 * Thomas Plum, Plum Hall Inc, 609-927-3770
 * If machine traps overflow, use an  unsigned  type
 * Let  T  be the execution time in milliseconds
 * Then  average time per operator  =  T/major  usec
 * (Because the inner loop has exactly 1000 operations)
 */
#define STOR_CL auto
#define TYPE short
#include <stdio.h>

main (int ac, char *av[])
{  STOR_CL TYPE a, b, c;
   long d, major;

    scanf ("%ld", &major);
    printf("executing %ld iterations\n", major);
    scanf ("%ld", &a);
    scanf ("%ld", &b);
    for (d = 1; d <= major; ++d)
    {
        /* inner loop executes 1000 selected operations */
        for (c = 1; c <= 40; ++c)
        {
            a = a + b + c;
            b = a >> 1;
            a = b % 10;
            a = b == c;
            b = a | c;
            a = !b;
            b = a + c;
            a = b > c;
        }
    }
    printf("a=%d\n", a);
}
```

Figure 9-29: Benchsho.c

Subroutine	Low-level	High-level	% Reduction
main	101	25	75.25
total	101	25	75.25

Figure 9-30: Benchsho Statistics

9.7.5 Benchlng.exe

Benchlng is a program from the Plum-Hall benchmark suite, which benchmarks long variables. The program is exactly the same as the benchsho.exe program, but makes use of long variables rather than short integers. The disassembly program is shown in Figure 9-31, the decompiled C program in Figure 9-32, and the initial C program in Figure 9-33. The program has the following call graph:

```
main
    scanf
    printf
    LMOD@
```

As seen from the disassembly of the program, the long variables are located in the stack at offsets `-4`, `-20`, `-16`, `-8`, and `-12`. The final C decompiled code makes use of five long variables and an integer variable `loc6`. This latter variable is used as a Boolean variable to hold the contents of a Boolean expression evaluation. Three Boolean expression evaluations are seen in the final C code:

```
loc1 == loc2
LO(loc1) | HI(loc1)
loc1 > loc2
```

All these expressions can be transformed into Boolean assignment by means of the transformation described in the previous Section. The generated code would look like this:

```
loc4 = (loc1 == loc2);
/* other code here */
loc4 = (LO(loc1) | HI(loc1));
/* other code here */
loc4 = (loc1 > loc2);
```

The second Boolean expression checks the low and high part of a long variable and `ors` them together; this is equivalent to a logical negation of the long variable, which would lead to the following final code:

```
loc4 = !loc1;
```

The benchlng program as compared to the benchsho program has 27.34% more low-level instructions in the `main` program (the `LMOD@` subroutine calculates the modulus of long variables and is untranslatable to a high-level language), three more instructions in the high-level representation of `main` (due to the logical negation of a long variable, which makes use of the temporary Boolean variable `loc6`), and performs a reduction of 79.86% instructions as shown in Figure 9-34.

```
       LMOD@  PROC  FAR
000 001EEB B90200             MOV        cx, 2
002 001EF3 55                 PUSH       bp
003 001EF4 56                 PUSH       si
004 001EF5 57                 PUSH       di
005 001EF6 8BEC               MOV        bp, sp
006 001EF8 8BF9               MOV        di, cx
007 001EFA 8B460A             MOV        ax, [bp+0Ah]
008 001EFD 8B560C             MOV        dx, [bp+0Ch]
009 001F00 8B5E0E             MOV        bx, [bp+0Eh]
010 001F03 8B4E10             MOV        cx, [bp+10h]
011 001F06 0BC9               OR         cx, cx
012 001F08 7508               JNE        L1
013 001F0A 0BD2               OR         dx, dx
014 001F0C 7469               JE         L2
015 001F0E 0BDB               OR         bx, bx
016 001F10 7465               JE         L2
017 001F12 F7C70100      L1:  TEST       di, 1
018 001F16 751C               JNE        L3
019 001F18 0BD2               OR         dx, dx
020 001F1A 790A               JNS        L4
021 001F1C F7DA               NEG        dx
022 001F1E F7D8               NEG        ax
023 001F20 83DA00             SBB        dx, 0
024 001F23 83CF0C             OR         di, 0Ch
025 001F26 0BC9          L4:  OR         cx, cx
026 001F28 790A               JNS        L3
027 001F2A F7D9               NEG        cx
028 001F2C F7DB               NEG        bx
029 001F2E 83D900             SBB        cx, 0
030 001F31 83F704             XOR        di, 4
031 001F34 8BE9          L3:  MOV        bp, cx
032 001F36 B92000             MOV        cx, 20h
033 001F39 57                 PUSH       di
034 001F3A 33FF               XOR        di, di
035 001F3C 33F6               XOR        si, si
036 001F3E D1E0          L5:  SHL        ax, 1
037 001F40 D1D2               RCL        dx, 1
038 001F42 D1D6               RCL        si, 1
039 001F44 D1D7               RCL        di, 1
040 001F46 3BFD               CMP        di, bp
041 001F48 720B               JB         L6
042 001F4A 7704               JA         L7
043 001F4C 3BF3               CMP        si, bx
044 001F4E 7205               JB         L6
```

Figure 9-31: Benchlng.a2

```
045 001F50 2BF3          L7:  SUB            si, bx
046 001F52 1BFD               SBB            di, bp
047 001F54 40                 INC            ax
048 001F55 E2E7          L6:  LOOP           L5
049 001F57 5B                 POP            bx
050 001F58 F7C30200           TEST           bx, 2
051 001F5C 7406               JE             L8
052 001F5E 8BC6               MOV            ax, si
053 001F60 8BD7               MOV            dx, di
054 001F62 D1EB               SHR            bx, 1
055 001F64 F7C30400      L8:  TEST           bx, 4
056 001F68 7407               JE             L9
057 001F6A F7DA               NEG            dx
058 001F6C F7D8               NEG            ax
059 001F6E 83DA00             SBB            dx, 0
060 001F71 5F            L9:  POP            di
061 001F72 5E                 POP            si
062 001F73 5D                 POP            bp
063 001F74 CA0800             RETF           8
064                      L2:  MOV            tmp, dx:ax      ;Synthetic inst
065 001F77 F7F3               DIV            bx
066                           MOD            bx              ;Synthetic inst
067 001F79 F7C70200           TEST           di, 2
068 001F7D 7402               JE             L10
069 001F7F 8BC2               MOV            ax, dx
070 001F81 33D2          L10: XOR            dx, dx
071 001F83 EBEC               JMP            L9
       LMOD@  ENDP

       main  PROC  NEAR
000 0002FA 55                 PUSH           bp
001 0002FB 8BEC               MOV            bp, sp
002 0002FD 83EC14             SUB            sp, 14h
003 000300 8D46FC             LEA            ax, [bp-4]
004 000303 50                 PUSH           ax
005 000304 B89401             MOV            ax, 194h
006 000307 50                 PUSH           ax
007 000308 E85D15             CALL  near ptr scanf
008 00030B 59                 POP            cx
009 00030C 59                 POP            cx
010 00030D FF76FE             PUSH  word ptr [bp-2]
011 000310 FF76FC             PUSH  word ptr [bp-4]
012 000313 B89801             MOV            ax, 198h
013 000316 50                 PUSH           ax
014 000317 E8C50C             CALL  near ptr printf
```

Figure 9-31: Benchlng.a2 – Continued

```
015 00031A 83C406               ADD            sp, 6
016 00031D 8D46EC               LEA            ax, [bp-14h]
017 000320 50                   PUSH           ax
018 000321 B8B201               MOV            ax, 1B2h
019 000324 50                   PUSH           ax
020 000325 E84015               CALL  near ptr scanf
021 000328 59                   POP            cx
022 000329 59                   POP            cx
023 00032A 8D46F0               LEA            ax, [bp-10h]
024 00032D 50                   PUSH           ax
025 00032E B8B601               MOV            ax, 1B6h
026 000331 50                   PUSH           ax
027 000332 E83315               CALL  near ptr scanf
028 000335 59                   POP            cx
029 000336 59                   POP            cx
030 000337 C746FA0000           MOV   word ptr [bp-6], 0
031 00033C C746F80100           MOV   word ptr [bp-8], 1

033 00042D 8B56FA          L11: MOV            dx, [bp-6]
034 000430 8B46F8               MOV            ax, [bp-8]
035 000433 3B56FE               CMP            dx, [bp-2]
036 000436 7D03                 JGE            L12

038 000344 C746F60000      L13: MOV   word ptr [bp-0Ah], 0
039 000349 C746F40100           MOV   word ptr [bp-0Ch], 1
041 000411 837EF600        L14: CMP   word ptr [bp-0Ah], 0
042 000415 7D03                 JGE            L15

044 000351 8B56EE          L16: MOV            dx, [bp-12h]
045 000354 8B46EC               MOV            ax, [bp-14h]
046 000357 0346F0               ADD            ax, [bp-10h]
047 00035A 1356F2               ADC            dx, [bp-0Eh]
048 00035D 0346F4               ADD            ax, [bp-0Ch]
049 000360 1356F6               ADC            dx, [bp-0Ah]
050 000363 8956EE               MOV            [bp-12h], dx
051 000366 8946EC               MOV            [bp-14h], ax
052 000369 8B56EE               MOV            dx, [bp-12h]
053 00036C 8B46EC               MOV            ax, [bp-14h]
054 00036F D1FA                 SAR            dx, 1
055 000371 D1D8                 RCR            ax, 1
056 000373 8956F2               MOV            [bp-0Eh], dx
057 000376 8946F0               MOV            [bp-10h], ax
058 000379 33D2                 XOR            dx, dx
059 00037B B80A00               MOV            ax, 0Ah
```

Figure 9-31: Benchlng.a2 – Continued

```
060 00037E 52                 PUSH            dx
061 00037F 50                 PUSH            ax
062 000380 FF76F2             PUSH  word ptr [bp-0Eh]
063 000383 FF76F0             PUSH  word ptr [bp-10h]
064 000386 9AEB1D1000         CALL   far ptr LMOD@
065 00038B 8956EE             MOV             [bp-12h], dx
066 00038E 8946EC             MOV             [bp-14h], ax
067 000391 8B56F2             MOV             dx, [bp-0Eh]
068 000394 8B46F0             MOV             ax, [bp-10h]
069 000397 3B56F6             CMP             dx, [bp-0Ah]
070 00039A 750A               JNE             L17
071 00039C 3B46F4             CMP             ax, [bp-0Ch]
072 00039F 7505               JNE             L17
073 0003A1 B80100             MOV             ax, 1
075 0003A8 99            L18: CWD
076 0003A9 8956EE             MOV             [bp-12h], dx
077 0003AC 8946EC             MOV             [bp-14h], ax
078 0003AF 8B56EE             MOV             dx, [bp-12h]
079 0003B2 8B46EC             MOV             ax, [bp-14h]
080 0003B5 0B46F4             OR              ax, [bp-0Ch]
081 0003B8 0B56F6             OR              dx, [bp-0Ah]
082 0003BB 8956F2             MOV             [bp-0Eh], dx
083 0003BE 8946F0             MOV             [bp-10h], ax
084 0003C1 8B46F0             MOV             ax, [bp-10h]
085 0003C4 0B46F2             OR              ax, [bp-0Eh]
086 0003C7 7505               JNE             L19
087 0003C9 B80100             MOV             ax, 1

089 0003D0 99            L20: CWD
090 0003D1 8956EE             MOV             [bp-12h], dx
091 0003D4 8946EC             MOV             [bp-14h], ax
092 0003D7 8B56EE             MOV             dx, [bp-12h]
093 0003DA 8B46EC             MOV             ax, [bp-14h]
094 0003DD 0346F4             ADD             ax, [bp-0Ch]
095 0003E0 1356F6             ADC             dx, [bp-0Ah]
096 0003E3 8956F2             MOV             [bp-0Eh], dx
097 0003E6 8946F0             MOV             [bp-10h], ax
098 0003E9 8B56F2             MOV             dx, [bp-0Eh]
099 0003EC 8B46F0             MOV             ax, [bp-10h]
100 0003EF 3B56F6             CMP             dx, [bp-0Ah]
101 0003F2 7C0C               JL              L21
102 0003F4 7F05               JG              L22
103 0003F6 3B46F4             CMP             ax, [bp-0Ch]
104 0003F9 7605               JBE             L21
```

Figure 9-31: Benchlng.a2 – Continued

```
105 0003FB B80100         L22: MOV           ax, 1

107 000402 99             L23: CWD
108 000403 8956EE              MOV           [bp-12h], dx
109 000406 8946EC              MOV           [bp-14h], ax
110 000409 8346F401            ADD  word ptr [bp-0Ch], 1
111 00040D 8356F600            ADC  word ptr [bp-0Ah], 0
112                            JMP           L14             ;Synthetic inst
113 000400 33C0           L21: XOR           ax, ax
114                            JMP           L23             ;Synthetic inst
115 0003CE 33C0           L19: XOR           ax, ax
116                            JMP           L20             ;Synthetic inst
117 0003A6 33C0           L17: XOR           ax, ax
118                            JMP           L18             ;Synthetic inst
119 00041A 7F09           L15: JG            L24
120 00041C 837EF428            CMP  word ptr [bp-0Ch], 28h
121 000420 7703                JA            L24
123 000425 8346F801       L24: ADD  word ptr [bp-8], 1
124 000429 8356FA00            ADC  word ptr [bp-6], 0
125                            JMP           L11             ;Synthetic inst
126 00043B 7F08           L12: JG            L25
127 00043D 3B46FC              CMP           ax, [bp-4]
128 000440 7703                JA            L25

130 000445 FF76EE         L25: PUSH word ptr [bp-12h]
131 000448 FF76EC              PUSH word ptr [bp-14h]
132 00044B B8BA01              MOV           ax, 1BAh
133 00044E 50                  PUSH          ax
134 00044F E88D0B              CALL near ptr printf
135 000452 83C406              ADD           sp, 6
136 000455 8BE5                MOV           sp, bp
137 000457 5D                  POP           bp
138 000458 C3                  RET
       main  ENDP
```

Figure 9-31: Benchlng.a2 – Continued

```
/*
 * Input file : benchlng.exe
 * File type : EXE
 */

#include "dcc.h"

long LMOD@ (long arg0, long arg2)
/* Takes 8 bytes of parameters.
 * Runtime support routine of the compiler.
 * Untranslatable routine.  Assembler provided.
 * Return value in registers dx:ax.
 * Pascal calling convention.
 */
{
    /* disassembly code here */
}

void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
long loc1;
long loc2;
long loc3;
long loc4;
long loc5;
int loc6; /* ax */

    scanf ("%ld", &loc5);
    printf ("executing %ld iterations\n", loc5);
    scanf ("%ld", &loc2);
    scanf ("%ld", &loc4);
    loc3 = 1;
    while ((loc3 <= loc5)) {
        loc2 = 1;
        while ((loc2 <= 40)) {
            loc4 = ((loc4 + loc1) + loc2);
            loc1 = (loc4 >> 1);
            loc4 = LMOD@ (loc1, 10);
            if (loc1 == loc2) {
                loc6 = 1;
            }
```

Figure 9-32: Benchlng.b

```
            else {
                loc6 = 0;
            }
            loc4 = loc6;
            loc1 = (loc4 | loc2);
            if ((LO(loc1) | HI(loc1)) == 0) {
                loc6 = 1;
            }
            else {
                loc6 = 0;
            }
            loc4 = loc6;
            loc1 = (loc4 + loc2);
            if (loc1 > loc2) {
                loc6 = 1;
            }
            else {
                loc6 = 0;
            }
            loc4 = loc6;
            loc2 = (loc2 + 1);
        }
        loc3 = (loc3 + 1);
    }
    printf ("a=%d\n", loc4);
}
```

Figure 9-32: Benchlng.b – Continued

```
/* benchlng - benchmark for long integers
 * Thomas Plum, Plum Hall Inc, 609-927-3770
 * If machine traps overflow, use an  unsigned  type
 * Let  T  be the execution time in milliseconds
 * Then  average time per operator  =  T/major  usec
 * (Because the inner loop has exactly 1000 operations)
 */
#define TYPE long
#include <stdio.h>

main (int ac, char *av[])
{  TYPE a, b, c;
   long d, major;

    scanf ("%ld", &major);
    printf("executing %ld iterations\n", major);
    scanf ("%ld", &a);
    scanf ("%ld", &b);
    for (d = 1; d <= major; ++d)
    {
        /* inner loop executes 1000 selected operations */
        for (c = 1; c <= 40; ++c)
        {
            a = a + b + c;
            b = a >> 1;
            a = b % 10;
            a = b == c;
            b = a | c;
            a = !b;
            b = a + c;
            a = b > c;
        }
    }
    printf("a=%d\n", a);
}
```

Figure 9-33: Benchlng.c

Subroutine	Low-level	High-level	% Reduction
LMOD@	72	-	-
main	139	28	79.86
total	139	28	79.86

Figure 9-34: Benchlng Statistics

9.7.6 Benchmul.exe

Benchmul is another program from the Plum-Hall benchmarks. This program benchmarks integer multiplication by executing 1000 multiplications in a loop. The disassembly program is shown in Figure 9-35, the decompiled C program in Figure 9-36, and the initial C program in Figure 9-37. This program has the following call graph:

```
main
   scanf
   printf
```

Benchmul makes use of two long variables to loop a large number of times through the program, and three integer variables that perform the operations; one of these variables is not actually used in the program. As seen from the disassembly, the long variables are located on the stack at offsets `-4` and `-8`, and the integer variables are at offsets `-12`, `-10`, and on the register variable `si`. The final C code is identical to the initial C code, and a reduction of 86.36% of instructions was achieved by this program, as seen in Figure 9-38.

```
       main  PROC  NEAR
000 0002FA 55                   PUSH           bp
001 0002FB 8BEC                 MOV            bp, sp
002 0002FD 83EC0C               SUB            sp, 0Ch
003 000300 56                   PUSH           si
004 000301 8D46FC               LEA            ax, [bp-4]
005 000304 50                   PUSH           ax
006 000305 B89401               MOV            ax, 194h
007 000308 50                   PUSH           ax
008 000309 E8D014               CALL  near ptr scanf
009 00030C 59                   POP            cx
010 00030D 59                   POP            cx
011 00030E FF76FE               PUSH  word ptr [bp-2]
012 000311 FF76FC               PUSH  word ptr [bp-4]
013 000314 B89801               MOV            ax, 198h
014 000317 50                   PUSH           ax
015 000318 E8380C               CALL  near ptr printf
016 00031B 83C406               ADD            sp, 6
017 00031E 8D46F4               LEA            ax, [bp-0Ch]
018 000321 50                   PUSH           ax
019 000322 B8B201               MOV            ax, 1B2h
020 000325 50                   PUSH           ax
021 000326 E8B314               CALL  near ptr scanf
022 000329 59                   POP            cx
023 00032A 59                   POP            cx
024 00032B 8D46F6               LEA            ax, [bp-0Ah]
025 00032E 50                   PUSH           ax
026 00032F B8B501               MOV            ax, 1B5h
027 000332 50                   PUSH           ax
028 000333 E8A614               CALL  near ptr scanf
029 000336 59                   POP            cx
030 000337 59                   POP            cx
031 000338 C746FA0000           MOV   word ptr [bp-6], 0
032 00033D C746F80100           MOV   word ptr [bp-8], 1

034 0003AA 8B56FA          L1:  MOV            dx, [bp-6]
035 0003AD 8B46F8               MOV            ax, [bp-8]
036 0003B0 3B56FE               CMP            dx, [bp-2]
037 0003B3 7C8F                 JL             L2
038 0003B5 7F05                 JG             L3
039 0003B7 3B46FC               CMP            ax, [bp-4]
040 0003BA 7688                 JBE            L2
041 0003BC FF76F4          L3:  PUSH  word ptr [bp-0Ch]
042 0003BF B8B801               MOV            ax, 1B8h
043 0003C2 50                   PUSH           ax
044 0003C3 E88D0B               CALL  near ptr printf
```

Figure 9-35: Benchmul.a2

```
045 0003C6 59                   POP            cx
046 0003C7 59                   POP            cx
047 0003C8 5E                   POP            si
048 0003C9 8BE5                 MOV            sp, bp
049 0003CB 5D                   POP            bp
050 0003CC C3                   RET
051 000344 BE0100          L2:  MOV            si, 1

053 00039D 83FE28          L4:  CMP            si, 28h
054 0003A0 7EA7                 JLE            L5
055 0003A2 8346F801             ADD   word ptr [bp-8], 1
056 0003A6 8356FA00             ADC   word ptr [bp-6], 0
057                             JMP            L1              ;Synthetic inst
058 000349 8B46F4          L5:  MOV            ax, [bp-0Ch]
059 00034C F766F4               MUL   word ptr [bp-0Ch]
060 00034F F766F4               MUL   word ptr [bp-0Ch]
061 000352 F766F4               MUL   word ptr [bp-0Ch]
062 000355 F766F4               MUL   word ptr [bp-0Ch]
063 000358 F766F4               MUL   word ptr [bp-0Ch]
064 00035B F766F4               MUL   word ptr [bp-0Ch]
065 00035E F766F4               MUL   word ptr [bp-0Ch]
066 000361 F766F4               MUL   word ptr [bp-0Ch]
067 000364 F766F4               MUL   word ptr [bp-0Ch]
068 000367 F766F4               MUL   word ptr [bp-0Ch]
069 00036A F766F4               MUL   word ptr [bp-0Ch]
070 00036D F766F4               MUL   word ptr [bp-0Ch]
071 000370 F766F4               MUL   word ptr [bp-0Ch]
072 000373 F766F4               MUL   word ptr [bp-0Ch]
073 000376 F766F4               MUL   word ptr [bp-0Ch]
074 000379 F766F4               MUL   word ptr [bp-0Ch]
075 00037C F766F4               MUL   word ptr [bp-0Ch]
076 00037F F766F4               MUL   word ptr [bp-0Ch]
077 000382 F766F4               MUL   word ptr [bp-0Ch]
078 000385 F766F4               MUL   word ptr [bp-0Ch]
079 000388 F766F4               MUL   word ptr [bp-0Ch]
080 00038B F766F4               MUL   word ptr [bp-0Ch]
081 00038E F766F4               MUL   word ptr [bp-0Ch]
082 000391 F766F4               MUL   word ptr [bp-0Ch]
083 000394 BA0300               MOV            dx, 3
084 000397 F7E2                 MUL            dx
085 000399 8946F4               MOV            [bp-0Ch], ax
086 00039C 46                   INC            si
087                             JMP            L4              ;Synthetic inst
       main  ENDP
```

Figure 9-35: Benchmul.a2 – Continued

```
/*
 * Input file : benchmul.exe
 * File type : EXE
 */

#include "dcc.h"


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
long loc3;
long loc4;
int loc5;

    scanf ("%ld", &loc4);
    printf ("executing %ld iterations\n", loc4);
    scanf ("%d", &loc1);
    scanf ("%d", &loc2);
    loc3 = 1;
    while ((loc3 <= loc4)) {
        loc5 = 1;
        while ((loc5 <= 40)) {
            loc1 = (((((((((((((((((((((((((loc1 * loc1) * loc1) * loc1)
                   * loc1) * loc1) * loc1) * loc1) * loc1) * loc1) *
                   loc1) * loc1) * loc1) * loc1) * loc1) * loc1) * loc1)
                   * loc1) * loc1) * loc1) * loc1) * loc1) * loc1) *
                   loc1) * loc1) * 3);
            loc5 = (loc5 + 1);
        }
        loc3 = (loc3 + 1);
    }
    printf ("a=%d\n", loc1);
}
```

Figure 9-36: Benchmul.b

```
/* benchmul - benchmark for  int multiply
 * Thomas Plum, Plum Hall Inc, 609-927-3770
 * If machine traps overflow, use an  unsigned  type
 * Let  T  be the execution time in milliseconds
 * Then  average time per operator  =  T/major  usec
 * (Because the inner loop has exactly 1000 operations)
 */
#define STOR_CL auto
#define TYPE int
#include <stdio.h>

main (int ac, char *av[])
{  STOR_CL TYPE a, b, c;
   long d, major;

    scanf ("%ld", &major);
    printf("executing %ld iterations\n", major);
    scanf ("%d", &a);
    scanf ("%d", &b);
    for (d = 1; d <= major; ++d)
    {
        /* inner loop executes 1000 selected operations */
        for (c = 1; c <= 40; ++c)
        {
            a = 3 *a*a*a*a*a*a*a*a * a*a*a*a*a*a*a*a *
                                     a*a*a*a*a*a*a*a * a; /* 25 * */
        }
    }
    printf("a=%d\n", a);
}
```

Figure 9-37: Benchmul.c

Subroutine	Low-level	High-level	% Reduction
main	88	12	86.36
total	88	12	86.36

Figure 9-38: Benchmul Statistics

9.7.7 Benchfn.exe

Benchfn is a program from the Plum-Hall benchmark suite, which benchmarks function calls; 1000 subroutine calls are done each time around the loop. The disassembly program is shown in Figure 9-39, the decompiled C program in Figure 9-40, and the initial C program in Figure 9-41. This program has the following call graph:

```
main
   scanf
   printf
   proc_1
      proc_2
         proc_3
            proc_4
```

Benchfn has four procedures and a `main` program. Three of the four procedures invoke other procedure, and the fourth procedure is empty. The percentage of reduction on the number of intermediate instructions is not as high in this program as compared to the previous programs since there are not many expressions in the program (which is not normally the case with high-level programs). As seen in the statistics of this program (see Figure 9-42), the empty procedure has a 100% reduction since the procedure prologue and trailer low-level instructions are eliminated in the C program; the other three procedures have an average of 29.30% reduction of instructions on 29 procedure calls performed by them, and the main program has an 81.08% reduction of instructions since expressions and assignments are used in this procedure. The overall average for the program is low, 56.10%, and is due to the lack of assignment statements in this program.

```
       proc_4  PROC  NEAR
000 0002FA 55                 PUSH          bp
001 0002FB 8BEC               MOV           bp, sp
002 0002FD 5D                 POP           bp
003 0002FE C3                 RET
       proc_4  ENDP

       proc_3  PROC  NEAR
000 0002FF 55                 PUSH          bp
001 000300 8BEC               MOV           bp, sp
002 000302 E8F5FF             CALL  near ptr proc_4
003 000305 E8F2FF             CALL  near ptr proc_4
004 000308 E8EFFF             CALL  near ptr proc_4
005 00030B E8ECFF             CALL  near ptr proc_4
006 00030E E8E9FF             CALL  near ptr proc_4
007 000311 E8E6FF             CALL  near ptr proc_4
008 000314 E8E3FF             CALL  near ptr proc_4
009 000317 E8E0FF             CALL  near ptr proc_4
010 00031A E8DDFF             CALL  near ptr proc_4
011 00031D E8DAFF             CALL  near ptr proc_4
012 000320 5D                 POP           bp
013 000321 C3                 RET
       proc_3  ENDP

       proc_2  PROC  NEAR
000 000322 55                 PUSH          bp
001 000323 8BEC               MOV           bp, sp
002 000325 E8D7FF             CALL  near ptr proc_3
003 000328 E8D4FF             CALL  near ptr proc_3
004 00032B E8D1FF             CALL  near ptr proc_3
005 00032E E8CEFF             CALL  near ptr proc_3
006 000331 E8CBFF             CALL  near ptr proc_3
007 000334 E8C8FF             CALL  near ptr proc_3
008 000337 E8C5FF             CALL  near ptr proc_3
009 00033A E8C2FF             CALL  near ptr proc_3
010 00033D E8BFFF             CALL  near ptr proc_3
011 000340 E8BCFF             CALL  near ptr proc_3
012 000343 5D                 POP           bp
013 000344 C3                 RET
       proc_2  ENDP

       proc_1  PROC  NEAR
000 000345 55                 PUSH          bp
001 000346 8BEC               MOV           bp, sp
002 000348 E8D7FF             CALL  near ptr proc_2
```

Figure 9-39: Benchfn.a2

```
003 00034B E8D4FF               CALL  near ptr proc_2
004 00034E E8D1FF               CALL  near ptr proc_2
005 000351 E8CEFF               CALL  near ptr proc_2
006 000354 E8CBFF               CALL  near ptr proc_2
007 000357 E8C8FF               CALL  near ptr proc_2
008 00035A E8C5FF               CALL  near ptr proc_2
009 00035D E8C2FF               CALL  near ptr proc_2
010 000360 E8BFFF               CALL  near ptr proc_2
011 000363 5D                   POP            bp
012 000364 C3                   RET
       proc_1  ENDP

       main  PROC  NEAR
000 000365 55                   PUSH           bp
001 000366 8BEC                 MOV            bp, sp
002 000368 83EC08               SUB            sp, 8
003 00036B 8D46FC               LEA            ax, [bp-4]
004 00036E 50                   PUSH           ax
005 00036F B89401               MOV            ax, 194h
006 000372 50                   PUSH           ax
007 000373 E85614               CALL  near ptr scanf
008 000376 59                   POP            cx
009 000377 59                   POP            cx
010 000378 FF76FE               PUSH  word ptr [bp-2]
011 00037B FF76FC               PUSH  word ptr [bp-4]
012 00037E B89801               MOV            ax, 198h
013 000381 50                   PUSH           ax
014 000382 E8BE0B               CALL  near ptr printf
015 000385 83C406               ADD            sp, 6
016 000388 C746FA0000           MOV   word ptr [bp-6], 0
017 00038D C746F80100           MOV   word ptr [bp-8], 1

019 00039F 8B56FA          L1:  MOV            dx, [bp-6]
020 0003A2 8B46F8               MOV            ax, [bp-8]
021 0003A5 3B56FE               CMP            dx, [bp-2]
022 0003A8 7CEA                 JL             L2
023 0003AA 7F05                 JG             L3
024 0003AC 3B46FC               CMP            ax, [bp-4]
025 0003AF 76E3                 JBE            L2
026 0003B1 B8B201          L3:  MOV            ax, 1B2h
027 0003B4 50                   PUSH           ax
028 0003B5 E88B0B               CALL  near ptr printf
029 0003B8 59                   POP            cx
030 0003B9 8BE5                 MOV            sp, bp
```

Figure 9-39: Benchfn.a2 – Continued

```
031 0003BB 5D                 POP           bp
032 0003BC C3                 RET
033 000394 E8AEFF         L2: CALL  near ptr proc_1
034 000397 8346F801          ADD   word ptr [bp-8], 1
035 00039B 8356FA00          ADC   word ptr [bp-6], 0
036                           JMP           L1              ;Synthetic inst
       main  ENDP
```

Figure 9-39: Benchfn.a2 – Continued

```
/*
 * Input file : benchfn.exe
 * File type : EXE
 */

#include "dcc.h"


void proc_4 ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
}


void proc_3 ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
    proc_4 ();
}

void proc_2 ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
    proc_3 ();
    proc_3 ();
    proc_3 ();
    proc_3 ();
    proc_3 ();
    proc_3 ();
    proc_3 ();
```

Figure 9-40: Benchfn.b

```
    proc_3 ();
    proc_3 ();
    proc_3 ();
}


void proc_1 ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
    proc_2 ();
    proc_2 ();
    proc_2 ();
    proc_2 ();
    proc_2 ();
    proc_2 ();
    proc_2 ();
    proc_2 ();
    proc_2 ();
}


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
long loc1;
long loc2;

    scanf ("%ld", &loc2);
    printf ("executing %ld iterations\n", loc2);
    loc1 = 1;
    while ((loc1 <= loc2)) {
        proc_1 ();
        loc1 = (loc1 + 1);
    }
    printf ("finished\n");
}
```

Figure 9-40: Benchfn.b – Continued

```
/* benchfn - benchmark for function calls
 * Thomas Plum, Plum Hall Inc, 609-927-3770
 * Let  T  be the execution time in milliseconds
 * Then  average time per operator  =  T/major  usec
 * (Because the inner loop has exactly 1000 operations)
 */
#include <stdio.h>

f3() { ;}
f2() { f3();f3();f3();f3();f3();f3();f3();f3();f3();f3();} /* 10 */
f1() { f2();f2();f2();f2();f2();f2();f2();f2();f2();f2();} /* 10 */
f0() { f1();f1();f1();f1();f1();f1();f1();f1();f1();} /* 9 */

main (int ac, char *av[])
{  long d, major;

    scanf ("%ld", &major);
    printf("executing %ld iterations\n", major);
    for (d = 1; d <= major; ++d)
        f0();     /* executes 1000 calls */
    printf ("finished\n");
}
```

Figure 9-41: Benchfn.c

Subroutine	Low-level	High-level	% Reduction
proc_4	4	0	100.00
proc_3	14	10	28.57
proc_2	14	10	28.57
proc_1	13	9	30.77
main	37	7	81.08
total	82	36	56.10

Figure 9-42: Benchfn Statistics

9.7.8 Fibo.exe

Fibo is a program that calculates the Fibonacci of input numbers. The computation of the Fibonacci number is done in a recursive function (two recursions are used). The disassembly program is shown in Figure 9-43, the decompiled C program in Figure 9-44, and the initial C program in Figure 9-45. Fibo has the following call graph:

```
main
   scanf
   printf
   exit
   proc_1
      proc_1
```

The `main` of the decompiled C program has the same number of instructions as the initial C program; the `for()` loop is represented by a `while()` loop. The recursive Fibonacci function, `proc_1` in the decompiled program, makes use of five instructions as opposed to three instructions in the initial code. These extra instructions are due to a copy of the argument to a local variable (`loc1 = arg0;`), and the placement of the result in a register variable along two different paths (i.e. two different possible results) before returning this value. The code is functionally equivalent to the initial code in all ways. Note that on the second recursive invocation of `proc_1`, the actual parameter expression is `(loc1 + -2)`; which is equivalent to `(loc1 - 2)`. The former expression comes from the disassembly of the program which makes use of the addition of a local variable and a negative number, rather than the subtraction of a positive number. As seen in the statistics of the program (see Figure 9-46, the individual and overall reduction on the number of intermediate instruction is 80.77%.

```
       proc_1  PROC  NEAR
000 00035B 55                 PUSH           bp
001 00035C 8BEC               MOV            bp, sp
002 00035E 56                 PUSH           si
003 00035F 8B7604             MOV            si, [bp+4]
004 000362 83FE02             CMP            si, 2
005 000365 7E1C               JLE            L1
006 000367 8BC6               MOV            ax, si
007 000369 48                 DEC            ax
008 00036A 50                 PUSH           ax
009 00036B E8EDFF             CALL  near ptr proc_1
010 00036E 59                 POP            cx
011 00036F 50                 PUSH           ax
012 000370 8BC6               MOV            ax, si
013 000372 05FEFF             ADD            ax, 0FFFEh
014 000375 50                 PUSH           ax
015 000376 E8E2FF             CALL  near ptr proc_1
016 000379 59                 POP            cx
017 00037A 8BD0               MOV            dx, ax
018 00037C 58                 POP            ax
019 00037D 03C2               ADD            ax, dx

021 000388 5E            L2:  POP            si
022 000389 5D                 POP            bp
023 00038A C3                 RET
024 000383 B80100        L1:  MOV            ax, 1
025 000386 EB00               JMP            L2
       proc_1  ENDP

       main  PROC  NEAR
000 0002FA 55                 PUSH           bp
001 0002FB 8BEC               MOV            bp, sp
002 0002FD 83EC04             SUB            sp, 4
003 000300 56                 PUSH           si
004 000301 57                 PUSH           di
005 000302 B89401             MOV            ax, 194h
006 000305 50                 PUSH           ax
007 000306 E8080C             CALL  near ptr printf
008 000309 59                 POP            cx
009 00030A 8D46FC             LEA            ax, [bp-4]
010 00030D 50                 PUSH           ax
011 00030E B8B101             MOV            ax, 1B1h
012 000311 50                 PUSH           ax
013 000312 E88514             CALL  near ptr scanf
014 000315 59                 POP            cx
015 000316 59                 POP            cx
```

Figure 9-43: Fibo.a2

```
016 000317 BE0100              MOV          si, 1

018 000349 3B76FC         L3:  CMP          si, [bp-4]
019 00034C 7ECE                JLE          L4
020 00034E 33C0                XOR          ax, ax
021 000350 50                  PUSH         ax
022 000351 E87300              CALL  near ptr exit
023 000354 59                  POP          cx
024 000355 5F                  POP          di
025 000356 5E                  POP          si
026 000357 8BE5                MOV          sp, bp
027 000359 5D                  POP          bp
028 00035A C3                  RET
029 00031C B8B401         L4:  MOV          ax, 1B4h
030 00031F 50                  PUSH         ax
031 000320 E8EE0B              CALL  near ptr printf
032 000323 59                  POP          cx
033 000324 8D46FE              LEA          ax, [bp-2]
034 000327 50                  PUSH         ax
035 000328 B8C301              MOV          ax, 1C3h
036 00032B 50                  PUSH         ax
037 00032C E86B14              CALL  near ptr scanf
038 00032F 59                  POP          cx
039 000330 59                  POP          cx
040 000331 FF76FE              PUSH  word ptr [bp-2]
041 000334 E82400              CALL  near ptr proc_1
042 000337 59                  POP          cx
043 000338 8BF8                MOV          di, ax
044 00033A 57                  PUSH         di
045 00033B FF76FE              PUSH  word ptr [bp-2]
046 00033E B8C601              MOV          ax, 1C6h
047 000341 50                  PUSH         ax
048 000342 E8CC0B              CALL  near ptr printf
049 000345 83C406              ADD          sp, 6
050 000348 46                  INC          si
051                            JMP          L3                ;Synthetic inst
       main  ENDP
```

Figure 9-43: Fibo.a2 – Continued

```
/*
 * Input file : fibo.exe
 * File type : EXE
 */
#include "dcc.h"

int proc_1 (int arg0)
/* Takes 2 bytes of parameters.
 * High-level language prologue code.
 * C calling convention.
 */
{
int loc1;
int loc2; /* ax */

    loc1 = arg0;
    if (loc1 > 2) {
        loc2 = (proc_1 ((loc1 - 1)) + proc_1 ((loc1 + -2)));
    }
    else {
        loc2 = 1;
    }
    return (loc2);
}

void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;  int loc2;
int loc3;  int loc4;

    printf ("Input number of iterations: ");
    scanf ("%d", &loc1);
    loc3 = 1;
    while ((loc3 <= loc1)) {
        printf ("Input number: ");
        scanf ("%d", &loc2);
        loc4 = proc_1 (loc2);
        printf ("fibonacci(%d) = %u\n", loc2, loc4);
        loc3 = (loc3 + 1);
    }
    exit (0);
}
```

Figure 9-44: Fibo.b

```
#include <stdio.h>

int main()
{ int i, numtimes, number;
  unsigned value, fib();

    printf("Input number of iterations: ");
    scanf ("%d", &numtimes);
    for (i = 1; i <= numtimes; i++)
    {
        printf ("Input number: ");
        scanf ("%d", &number);
        value = fib(number);
        printf("fibonacci(%d) = %u\n", number, value);
    }
    exit(0);
}

unsigned fib(x)  /* compute fibonacci number recursively */
int x;
{
    if (x > 2)
        return (fib(x - 1) + fib(x - 2));
    else
        return (1);
}
```

Figure 9-45: Fibo.c

Subroutine	Low-level	High-level	% Reduction
proc_1	26	5	80.77
main	52	10	80.77
total	78	15	80.77

Figure 9-46: Fibo Statistics

9.7.9 Crc.exe

Crc is a program that calculates the cyclic redundancy check (CRC) for a 1-character message block, and then passes the resulting CRC back into the CRC functions to see if the "received" 1-character message and CRC are correct. The disassembly program is shown in Figure 9-47, the decompiled C program in Figure 9-48, and the initial C program in Figure 9-49. Crc has the following call graph:

```
main
   proc_1
   proc_2
      LXLSH@
      LXRSH@
   proc_3
      proc_2
   printf
```

As seen in the initial C program, crc has three functions and a `main` procedure. The decompiled version of the program has five functions and a `main` program; the two extra functions are runtime support routines to support long right and left shifts (`LXRSH@` and `LXLSH@` respectively). These two routines were initially written in assembler, and are translated into C by accessing the low and high parts of the long argument. As seen in the statistics of the program (see Figure 9-50), the user functions have a reduction of over 80% intermediate instructions. These functions have the same number of high-level instructions when compared with the original program. Function `proc_1` is the `crc_clear` function that returns zero. This function has a 83.33% reduction of intermediate instructions due to the overhead provided by the procedure prologue and trailer code. Function `proc_2` is the `crc_update` function that calculates the CRC for the input argument according to the CCITT recommended CRC generator function. This function uses 32 bits to compute the result, and returns the lower 16 bits as the function's value. The decompiled version of this function propagates the fact that only 16 bits are used for the result to the invoked runtime routine `LXRSH@`, and hence this latter function only returns an integer (16 bits) rather than a long integer; the code is much simpler than its homologous `LXLSH@` (which returns a long integer). The reduction in the number of instruction is of 84.62%. Function `proc_3` is the `crc_finish` function which returns the final two CRC characters that are to be transmitted at the end of the block. This function calls the `crc_update` function twice; one as an argument of the other. The reduction on the number of instructions is high (93.75%) since all 16 low-level instructions are transformed into 1 high-level return instruction. Finally, the `main` program invokes the functions in the right order; a reduction of 82.09% is achieved. Note that integers are used in this program rather than characters since there is no use of the character variables as such characters (i.e. an unsigned character generates the same code). The overall intermediate instruction reduction on the program is of 77.78%, which is less than 80% due to the runtime routines.

```
       proc_3  PROC  NEAR
000 000385 55                PUSH           bp
001 000386 8BEC              MOV            bp, sp
002 000388 33C0              XOR            ax, ax
003 00038A 50                PUSH           ax
004 00038B 33C0              XOR            ax, ax
005 00038D 50                PUSH           ax
006 00038E FF7604            PUSH  word ptr [bp+4]
007 000391 E86FFF            CALL  near ptr proc_2
008 000394 59                POP            cx
009 000395 59                POP            cx
010 000396 50                PUSH           ax
011 000397 E869FF            CALL  near ptr proc_2
012 00039A 8BE5              MOV            sp, bp
014 00039E 5D                POP            bp
015 00039F C3                RET
       proc_3  ENDP

       LXRSH@  PROC  FAR
000 001585 80F910            CMP            cl, 10h
001 001588 7310              JAE            L1
002 00158A 8BDA              MOV            bx, dx
003 00158C D3E8              SHR            ax, cl
004 00158E D3FA              SAR            dx, cl
005 001590 F6D9              NEG            cl
006 001592 80C110            ADD            cl, 10h
007 001595 D3E3              SHL            bx, cl
008 001597 0BC3              OR             ax, bx
009 001599 CB                RETF
010 00159A 80E910       L1:  SUB            cl, 10h
011 00159D 8BC2              MOV            ax, dx
012 00159F 99                CWD
013 0015A0 D3F8              SAR            ax, cl
014 0015A2 CB                RETF
       LXRSH@  ENDP

       proc_1  PROC  NEAR
000 0002FA 55                PUSH           bp
001 0002FB 8BEC              MOV            bp, sp
002 0002FD 33C0              XOR            ax, ax
004 000301 5D                POP            bp
005 000302 C3                RET
       proc_1  ENDP
```

Figure 9-47: Crc.a2

```
       LXLSH@  PROC  FAR
000 0015A3 80F910              CMP            cl, 10h
001 0015A6 7310                JAE            L2
002 0015A8 8BD8                MOV            bx, ax
003 0015AA D3E0                SHL            ax, cl
004 0015AC D3E2                SHL            dx, cl
005 0015AE F6D9                NEG            cl
006 0015B0 80C110              ADD            cl, 10h
007 0015B3 D3EB                SHR            bx, cl
008 0015B5 0BD3                OR             dx, bx
009 0015B7 CB                  RETF
010 0015B8 80E910         L2:  SUB            cl, 10h
011 0015BB 8BD0                MOV            dx, ax
012 0015BD 33C0                XOR            ax, ax
013 0015BF D3E2                SHL            dx, cl
014 0015C1 CB                  RETF
       LXLSH@  ENDP

       proc_2  PROC  NEAR
000 000303 55                  PUSH           bp
001 000304 8BEC                MOV            bp, sp
002 000306 83EC06              SUB            sp, 6
003 000309 8B4604              MOV            ax, [bp+4]
004 00030C 99                  CWD
005 00030D B108                MOV            cl, 8
006 00030F 9AA3141000          CALL   far ptr LXLSH@
007 000314 52                  PUSH           dx
008 000315 50                  PUSH           ax
009 000316 8A4606              MOV            al, [bp+6]
010 000319 98                  CWD
011 00031A 99                  CWD
012 00031B 5B                  POP            bx
013 00031C 59                  POP            cx
014 00031D 03D8                ADD            bx, ax
015 00031F 13CA                ADC            cx, dx
016 000321 894EFC              MOV            [bp-4], cx
017 000324 895EFA              MOV            [bp-6], bx
018 000327 C746FE0000          MOV   word ptr [bp-2], 0

020 000365 837EFE08       L3:  CMP   word ptr [bp-2], 8
021 000369 7CC3                JL             L4
022 00036B 8B56FC              MOV            dx, [bp-4]
023 00036E 8B46FA              MOV            ax, [bp-6]
024 000371 2500FF              AND            ax, 0FF00h
025 000374 81E2FF00            AND            dx, 0FFh
026 000378 B108                MOV            cl, 8
027 00037A 9A85141000          CALL   far ptr LXRSH@
```

Figure 9-47: Crc.a2 – Continued

```
029 000381 8BE5                 MOV           sp, bp
030 000383 5D                   POP           bp
031 000384 C3                   RET
032 00032E 8B56FC          L4:  MOV           dx, [bp-4]
033 000331 8B46FA               MOV           ax, [bp-6]
034 000334 D1E0                 SHL           ax, 1
035 000336 D1D2                 RCL           dx, 1
036 000338 8956FC               MOV           [bp-4], dx
037 00033B 8946FA               MOV           [bp-6], ax
038 00033E 8B56FC               MOV           dx, [bp-4]
039 000341 8B46FA               MOV           ax, [bp-6]
040 000344 250000               AND           ax, 0
041 000347 81E20001             AND           dx, 100h
042 00034B 0BD0                 OR            dx, ax
043 00034D 7413                 JE            L5
044 00034F 8B56FC               MOV           dx, [bp-4]
045 000352 8B46FA               MOV           ax, [bp-6]
046 000355 350021               XOR           ax, 2100h
047 000358 81F21001             XOR           dx, 110h
048 00035C 8956FC               MOV           [bp-4], dx
049 00035F 8946FA               MOV           [bp-6], ax
050 000362 FF46FE          L5:  INC   word ptr [bp-2]
051                             JMP           L3                ;Synthetic inst
       proc_2  ENDP

       main  PROC  NEAR
000 0003A0 55                   PUSH          bp
001 0003A1 8BEC                 MOV           bp, sp
002 0003A3 83EC06               SUB           sp, 6
003 0003A6 C646FD41             MOV   byte ptr [bp-3], 41h
004 0003AA E84DFF               CALL  near ptr proc_1
005 0003AD 8946FA               MOV           [bp-6], ax
006 0003B0 8A46FD               MOV           al, [bp-3]
007 0003B3 98                   CWD
008 0003B4 50                   PUSH          ax
009 0003B5 FF76FA               PUSH  word ptr [bp-6]
010 0003B8 E848FF               CALL  near ptr proc_2
011 0003BB 59                   POP           cx
012 0003BC 59                   POP           cx
013 0003BD 8946FA               MOV           [bp-6], ax
014 0003C0 FF76FA               PUSH  word ptr [bp-6]
015 0003C3 E8BFFF               CALL  near ptr proc_3
016 0003C6 59                   POP           cx
017 0003C7 8946FA               MOV           [bp-6], ax
018 0003CA 8B46FA               MOV           ax, [bp-6]
019 0003CD 2500FF               AND           ax, 0FF00h
020 0003D0 B108                 MOV           cl, 8
```

Figure 9-47: Crc.a2 – Continued

```
021 0003D2 D3E8               SHR            ax, cl
022 0003D4 8846FE             MOV            [bp-2], al
023 0003D7 8A46FA             MOV            al, [bp-6]
024 0003DA 24FF               AND            al, 0FFh
025 0003DC 8846FF             MOV            [bp-1], al
026 0003DF FF76FA             PUSH  word ptr [bp-6]
027 0003E2 B89401             MOV            ax, 194h
028 0003E5 50                 PUSH           ax
029 0003E6 E8FC08             CALL  near ptr printf
030 0003E9 59                 POP            cx
031 0003EA 59                 POP            cx
032 0003EB E80CFF             CALL  near ptr proc_1
033 0003EE 8946FA             MOV            [bp-6], ax
034 0003F1 8A46FD             MOV            al, [bp-3]
035 0003F4 98                 CWD
036 0003F5 50                 PUSH           ax
037 0003F6 FF76FA             PUSH  word ptr [bp-6]
038 0003F9 E807FF             CALL  near ptr proc_2
039 0003FC 59                 POP            cx
040 0003FD 59                 POP            cx
041 0003FE 8946FA             MOV            [bp-6], ax
042 000401 8A46FE             MOV            al, [bp-2]
043 000404 98                 CWD
044 000405 50                 PUSH           ax
045 000406 FF76FA             PUSH  word ptr [bp-6]
046 000409 E8F7FE             CALL  near ptr proc_2
047 00040C 59                 POP            cx
048 00040D 59                 POP            cx
049 00040E 8946FA             MOV            [bp-6], ax
050 000411 8A46FF             MOV            al, [bp-1]
051 000414 98                 CWD
052 000415 50                 PUSH           ax
053 000416 FF76FA             PUSH  word ptr [bp-6]
054 000419 E8E7FE             CALL  near ptr proc_2
055 00041C 59                 POP            cx
056 00041D 59                 POP            cx
057 00041E 8946FA             MOV            [bp-6], ax
058 000421 FF76FA             PUSH  word ptr [bp-6]
059 000424 B89A01             MOV            ax, 19Ah
060 000427 50                 PUSH           ax
061 000428 E8BA08             CALL  near ptr printf
062 00042B 59                 POP            cx
063 00042C 59                 POP            cx
064 00042D 8BE5               MOV            sp, bp
065 00042F 5D                 POP            bp
066 000430 C3                 RET
       main  ENDP
```

Figure 9-47: Crc.a2 – Continued

```
/*
 * Input file : crc.exe
 * File type : EXE
 */

#include "dcc.h"


int proc_1 ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
    return (0);
}


long LXLSH@ (long arg0, char arg1)
/* Uses register arguments:
 *     arg0 = dx:ax.
 *     arg1 = cl.
 * Runtime support routine of the compiler.
 */
{
int loc1; /* bx */

    if (arg1 < 16) {
        loc1 = LO(arg0);
        LO(arg0) = (LO(arg0) << arg1);
        HI(arg0) = (HI(arg0) << arg1);
        HI(arg0) = (HI(arg0) | (loc1 >> (!arg1 + 16)));
        return (arg0);
    }
    else {
        HI(arg0) = LO(arg0);
        LO(arg0) = 0;
        HI(arg0) = (HI(arg0) << (arg1 - 16));
        return (arg0);
    }
}
```

Figure 9-48: Crc.b

```
int LXRSH@ (long arg0, char arg1)
/* Uses register arguments:
 *     arg0 = dx:ax.
 *     arg1 = cl.
 * Runtime support routine of the compiler.
 */
{
int loc1; /* bx */

    if (arg1 < 16) {
        loc1 = HI(arg0);
        LO(arg0) = (LO(arg0) >> arg1);
        HI(arg0) = (HI(arg0) >> arg1);
        return ((LO(arg0) | (loc1 << (!arg1 + 16))));
    }
    else {
        return ((HI(arg0) >> (arg1 - 16)));
    }
}


int proc_2 (int arg0, unsigned char arg1)
/* Takes 4 bytes of parameters.
 * High-level language prologue code.
 * C calling convention.
 */
{
int loc1;
long loc2;

    loc2 = (LXLSH@ (arg0, 8) + arg1);
    loc1 = 0;
    while ((loc1 < 8)) {
        loc2 = (loc2 << 1);
        if ((loc2 & 0x1000000) != 0) {
            loc2 = (loc2 ^ 0x1102100);
        }
        loc1 = (loc1 + 1);
    }
    return (LXRSH@ ((loc2 & 0xFFFF00), 8));
}
```

Figure 9-48: Crc.b – Continued

```
int proc_3 (int arg0)
/* Takes 2 bytes of parameters.
 * High-level language prologue code.
 * C calling convention.
 */
{
    return (proc_2 (proc_2 (arg0, 0), 0));
}


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
int loc3;
int loc4;

    loc1 = 65;
    loc2 = proc_1 ();
    loc2 = proc_2 (loc2, loc1);
    loc2 = proc_3 (loc2);
    loc3 = ((loc2 & 0xFF00) >> 8);
    loc4 = (loc2 & 255);
    printf ("%04x\n", loc2);
    loc2 = proc_1 ();
    loc2 = proc_2 (loc2, loc1);
    loc2 = proc_2 (loc2, loc3);
    loc2 = proc_2 (loc2, loc4);
    printf ("%04x\n", loc2);
}
```

Figure 9-48: Crc.b – Continued

```
/*
*   crc_clear:
*       This function clears the CRC to zero. It should be called prior to
*       the start of the processing of a block for both received messages,
*       and messages to be transmitted.
*
*       Calling sequence:
*
*       short crc;
*       crc = crc_clear();
*/
short crc_clear()
{
        return(0);
}
/*
*   crc_update:
*       this function must be called once for each character which is
*       to be included in the CRC for messages to be transmitted.
*       This function is called once for each character which is included
*       in the CRC of a received message, AND once for each of the two CRC
*       characters at the end of the received message. If the resulting
*       CRC is zero, then the message has been correctly received.
*
*   Calling sequence:
*
*       crc = crc_update(crc,next_char);
*/
short crc_update(crc,crc_char)
short crc;
char crc_char;
{
        long x;
        short i;

/* "x" will contain the character to be processed in bits 0-7 and the CRC    */
/* in bits 8-23. Bit 24 will be used to test for overflow, and then cleared  */
/* to prevent the sign bit of "x" from being set to 1. Bits 25-31 are not    */
/* used. ("x" is treated as though it is a 32 bit register).                 */
        x = ((long)crc << 8) + crc_char;    /* Get the CRC and the character */

/* Repeat the following loop 8 times (for the 8 bits of the character).      */
        for(i = 0;i < 8;i++)
        {
```

Figure 9-49: Crc.c

```
/* Shift the high-order bit of the character into the low-order bit of the   */
/* CRC, and shift the high-order bit of the CRC into bit 24.                 */
                x = x << 1;                        /* Shift "x" left one bit */

/* Test to see if the old high-order bit of the CRC was a 1.                 */
                if(x & 0x01000000)                     /* Test bit 24 of "x" */

/* If the old high-order bit of the CRC was a 1, exclusive-or it with a one  */
/* to set it to 0, and exclusive-or the CRC with hex 1021 to produce the     */
/* CCITT-recommended CRC generator of: X**16 + X**12 + X**5 + 1. To produce  */
/* the CRC generator of: X**16 + X**15 + X**2 + 1, change the constant from  */
/* 0x01102100 to 0x01800500. This will exclusive-or the CRC with hex 8005    */
/* and produce the same CRC that IBM uses for their synchronous transmission */
/* protocols.                                                                */
                        x = x ^ 0x01102100;     /* Exclusive-or "x" with a...*/
                                              /* ...constant of hex 01102100 */
/* And repeat 8 times.                                                       */
        }                                              /* End of "for" loop */

/* Return the CRC as the 16 low-order bits of this function's value.         */
        return(((x & 0x00ffff00) >> 8)); /* AND off the unneeded bits and... */
                                     /* ...shift the result 8 bits to the right */
}
/*
*   crc_finish:
*       This function must be called once after all the characters in a block
*       have been processed for a message which is to be TRANSMITTED. It
*       returns the calculated CRC bytes, which should be transmitted as the
*       two characters following the block. The first of these 2 bytes
*       must be taken from the high-order byte of the CRC, and the second
*       must be taken from the low-order byte of the CRC. This routine is NOT
*       called for a message which has been RECEIVED.
*
*   Calling sequence:
*
*       crc = crc_finish(crc);
*/
short crc_finish(crc)
short crc;
{
/* Call crc_update twice, passing it a character of hex 00 each time, to     */
/* flush out the last 16 bits from the CRC calculation, and return the       */
/* result as the value of this function.                                     */
        return(crc_update(crc_update(crc,'\0'),'\0'));
}
```

Figure 9-49: Crc.c – Continued

```
/*
* This is a sample of the use of the CRC functions, which calculates the
* CRC for a 1-character message block, and then passes the resulting CRC back
* into the CRC functions to see if the "received" 1-character message and CRC
* are correct.
*/
main()
{
        short crc;                                     /* The calculated CRC */
        char crc_char;                            /* The 1-character message */
        char x, y;            /* 2 places to hold the 2 "received" CRC bytes */

        crc_char = 'A';                     /* Define the 1-character message */
        crc = crc_clear();      /* Reset the CRC to "transmit" a new message */
        crc = crc_update(crc,crc_char);   /* Update the CRC for the first... */
                                   /* ...(and only) character of the message */
        crc = crc_finish(crc);        /* Finish the transmission calculation */
        x = (char)((crc & 0xff00) >> 8);  /* Extract the high-order CRC byte */
        y = (char)(crc & 0x00ff);          /* And extract the low-order byte */
        printf("%04x\n",crc);                           /* Print the results */

        crc = crc_clear();                 /* Prepare to "receive" a message */
        crc = crc_update(crc,crc_char);   /* Update the CRC for the first... */
                                   /* ...(and only) character of the message */
        crc = crc_update(crc,x);     /* Pass both bytes of the "received"... */
        crc = crc_update(crc,y);          /* ...CRC through crc_update, too */
        printf("%04x\n",crc);    /* If the result was 0, then the message... */
                                          /* ...was received without error */
}
```

Figure 9-49: Crc.c – Continued

Subroutine	Low-level	High-level	% Reduction
proc_1	6	1	83.33
LXLSH@	15	10	33.33
LXRSH@	15	6	60.00
proc_2	52	8	84.62
proc_3	16	1	93.75
main	67	12	82.09
total	171	38	77.78

Figure 9-50: Crc Statistics

9.7.10 Matrixmu

Matrixmu is a program that multiplies two matrixes. This program is incomplete in the sense that it does not initialize the matrixes, but was decompiled to show that the forward substitution method of Chapter 5, Section 5.4.10 is able to find array expressions. The conversion of this expression into an array was not done in **dcc**, but was explained in Chapter 5, Section 5.5. The disassembly program is shown in Figure 9-51, the decompiled C program in Figure 9-52, and the initial C program in Figure 9-53. The call graph for this program is as follows:

```
main
   proc_1
```

Both user procedures are decompiled with the same number of high-level instructions; 10 for the matrix multiplication procedure, and 1 for the `main` program. The reduction on the number of instructions is over 85% due to the large number of low-level instructions involved on the computation of an array offset. In the disassembled version of the program, the basic block at lines `026` to `069` of procedure `proc_1` has 44 instructions which are converted into two high-level instructions; a reduction of 95.45% intermediate instructions. Overall, this program has a 86.90% reduction of intermediate instructions, as shown in Figure 9-54.

```
       proc_1  PROC  NEAR
000 0002FA 55                     PUSH           bp
001 0002FB 8BEC                   MOV            bp, sp
002 0002FD 83EC02                 SUB            sp, 2
003 000300 56                     PUSH           si
004 000301 57                     PUSH           di
005 000302 33F6                   XOR            si, si

007 000378 83FE05          L1:    CMP            si, 5
008 00037B 7C89                   JL             L2
009 00037D 5F                     POP            di
010 00037E 5E                     POP            si
011 00037F 8BE5                   MOV            sp, bp
012 000381 5D                     POP            bp
013 000382 C3                     RET
014 000306 33FF            L2:    XOR            di, di

016 000372 83FF04          L3:    CMP            di, 4
017 000375 7C93                   JL             L4
018 000377 46                     INC            si
019                               JMP            L1              ;Synthetic inst
020 00030A C746FE0000      L4:    MOV   word ptr [bp-2], 0

022 00036B 837EFE04        L5:    CMP   word ptr [bp-2], 4
023 00036F 7CA0                   JL             L6
024 000371 47                     INC            di
025                               JMP            L3              ;Synthetic inst
026 000311 8BDE            L6:    MOV            bx, si
027 000313 D1E3                   SHL            bx, 1
028 000315 D1E3                   SHL            bx, 1
029 000317 D1E3                   SHL            bx, 1
030 000319 035E04                 ADD            bx, [bp+4]
031 00031C 8B46FE                 MOV            ax, [bp-2]
032 00031F D1E0                   SHL            ax, 1
033 000321 03D8                   ADD            bx, ax
034 000323 8B07                   MOV            ax, [bx]
035 000325 50                     PUSH           ax
036 000326 8B46FE                 MOV            ax, [bp-2]
037 000329 BA0A00                 MOV            dx, 0Ah
038 00032C F7E2                   MUL            dx
039 00032E 8BD8                   MOV            bx, ax
040 000330 035E06                 ADD            bx, [bp+6]
041 000333 8BC7                   MOV            ax, di
042 000335 D1E0                   SHL            ax, 1
043 000337 03D8                   ADD            bx, ax
044 000339 58                     POP            ax
```

Figure 9-51: Matrixmu.a2

```
045 00033A F727              MUL   word ptr [bx]
046 00033C 50                PUSH           ax
047 00033D 8BC6              MOV            ax, si
048 00033F BA0A00            MOV            dx, 0Ah
049 000342 F7E2              MUL            dx
050 000344 8BD8              MOV            bx, ax
051 000346 035E08            ADD            bx, [bp+8]
052 000349 8BC7              MOV            ax, di
053 00034B D1E0              SHL            ax, 1
054 00034D 03D8              ADD            bx, ax
055 00034F 58                POP            ax
056 000350 0307              ADD            ax, [bx]
057 000352 50                PUSH           ax
058 000353 8BC6              MOV            ax, si
059 000355 BA0A00            MOV            dx, 0Ah
060 000358 F7E2              MUL            dx
061 00035A 8BD8              MOV            bx, ax
062 00035C 035E08            ADD            bx, [bp+8]
063 00035F 8BC7              MOV            ax, di
064 000361 D1E0              SHL            ax, 1
065 000363 03D8              ADD            bx, ax
066 000365 58                POP            ax
067 000366 8907              MOV            [bx], ax
068 000368 FF46FE            INC   word ptr [bp-2]
069                          JMP            L5             ;Synthetic inst
        proc_1  ENDP

        main  PROC  NEAR
000 000383 55                PUSH           bp
001 000384 8BEC              MOV            bp, sp
002 000386 83EC78            SUB            sp, 78h
003 000389 8D46D8            LEA            ax, [bp-28h]
004 00038C 50                PUSH           ax
005 00038D 8D46B0            LEA            ax, [bp-50h]
006 000390 50                PUSH           ax
007 000391 8D4688            LEA            ax, [bp-78h]
008 000394 50                PUSH           ax
009 000395 E862FF            CALL  near ptr proc_1
010 000398 83C406            ADD            sp, 6
011 00039B 8BE5              MOV            sp, bp
012 00039D 5D                POP            bp
013 00039E C3                RET
       main  ENDP
```

Figure 9-51: Matrixmu.a2 – Continued

```
/*
 * Input file : matrixmu.exe
 * File type : EXE
 */

#include "dcc.h"


void proc_1 (int arg0, int arg1, int arg2)
/* Takes 6 bytes of parameters.
 * High-level language prologue code.
 * C calling convention.
 */
{
int loc1;
int loc2;
int loc3;

    loc2 = 0;
    while ((loc2 < 5)) {
        loc3 = 0;
        while ((loc3 < 4)) {
            loc1 = 0;
            while ((loc1 < 4)) {
                *((((loc2 * 10) + arg2) + (loc3 << 1))) =
                       ((*((((loc2 << 3) + arg0) + (loc1 << 1))) *
                         *((((loc1 * 10) + arg1) + (loc3 << 1)))) +
                         *((((loc2 * 10) + arg2) + (loc3 << 1))));
                loc1 = (loc1 + 1);
            }
            loc3 = (loc3 + 1);
        }
        loc2 = (loc2 + 1);
    }
}


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
int loc3;

    proc_1 (&loc3, &loc2, &loc1);
}
```

Figure 9-52: Matrixmu.b

```
#define n 5
#define m 4

static void multMatrix (int a[n][m], int b[m][n], int c[n][n])
{ int i,j,k;

    for (i=0; i<n; i++)
      for (j=0; j<m; j++)
        for (k=0; k<m; k++)
          c[i][j] = a[i][k] * b[k][j] + c[i][j];
}

main()
{ int a[n][m], b[n][m], c[n][m];

    multMatrix (a, b, c);
}
```

Figure 9-53: Matrixmu.c

Subroutine	Low-level	High-level	% Reduction
proc_1	70	10	85.71
main	14	1	92.86
total	84	11	86.90

Figure 9-54: Matrixmu Statistics

9.7.11 Overall Results

The summary results of the 10 programs that were presented in the previous sections are given in Figure 9-55. The total number of low-level intermediate instructions is 963, compared with the final 306 high-level instructions, which gives a reduction of instructions of 76.25%. This reduction of instructions is mainly due to the optimizations performed during data flow analysis, particularly extended register copy propagation (Chapter 5, Section 5.4.10). The recognition of idioms in the low-level code also reduces the number of instructions and helps in the determination of data types such as long integers. Decompiled programs have the same number of user subroutines, plus any runtime support routines used in the program. These latter routines are sometimes translatable into a high-level representation; assembler is generated whenever they are untranslatable.

Program	Low-level	High-level	% Reduction
intops	45	10	77.78
byteops	58	10	82.76
longops	117	48	58.97
benchsho	101	25	75.25
benchlng	139	28	79.86
benchmul	88	12	86.36
benchfn	82	36	56.10
fibo	78	15	80.77
crc	171	38	77.78
matrixmu	84	11	86.90
total	963	306	76.25

Figure 9-55: Results for Tested Programs

Chapter 10

Conclusions

This thesis has presented techniques for the reverse compilation or decompilation of binary programs, and provided algorithms for the implementation of the different phases of the decompiler. The methodology was implemented and tested in a prototype decompiler, **dcc**, which runs under DOS and Unix.

Decompilers use similar principles and techniques used in compilers. A decompiler has seven different phases, which incorporate compiler and optimization phases. There is no lexical analysis phase due to the simplicity of the source machine language. The syntax analysis phase parses the source binary program separating code from data, and placing data references in the symbol table. The main difficulty with the separation of code from data is that they are represented in the same way in von Neumann machines. The intermediate code generation phase generates a low-level intermediate representation of the program. The semantic analysis phase checks the semantic meaning of groups of low-level instructions (idioms), gathers type information, and propagates it across the intermediate representation. The control flow graph generation phase generates a control flow graph of each subroutine of the program, and attaches the intermediate representation information to the nodes of the graph. The data flow analysis phase analyzes the low-level intermediate code and converts it into a high-level intermediate representation available in any high-level language. The transformation of instructions eliminates all low-level references to condition codes and registers, and introduces the high-level concept of expression. Subroutines that are not representable in a high-level language are flagged. The structure of the program is analyzed in the control flow analysis phase, which structures the control flow graphs of each subroutine in the program. Finally, the code generation phase generates high-level code based on the high-level intermediate representation and the structured graph of each subroutine.

A complete decompilation of a program makes use of not only the decompiler but other related tools: the loader, the signature generator, the prototype generator, the disassembler, and the postprocessor. The loader loads the source binary program into memory, the signature generator generates signatures for known compilers and their libraries (if required), the prototype generator determines the formal argument types for library subroutines, the disassembler parses the program and produces an assembler output file, the decompiler makes use of the signature information to reduce the number of subroutines to decompile (i.e. it does not attempt to decompile library routines if they are recognized by a signature or the loader), and the postprocessor transforms the output decompiled high-level program into a semantically equivalent program that makes use of specific control structures available in the target language. In practice, a decompiler can take as input a binary program or an assembler program, and produce a high-level language output program. Most literature available on decompilers make use of the latter approach; an assembler source program. This thesis concentrates on source binary programs, which have far less information than assembler programs.

The techniques described in this thesis are general enough to construct decompilers for different machine architectures. The phases are grouped into 3 different modules that separate machine and language dependent features: the front-end is a machine dependent module that parses the source binary program and produces a low-level intermediate representation of the program and a control flow graph of each subroutine; the universal decompiling machine is a machine and language independent module that analyzes the intermediate code and the structure of the graph(s) and generates a high-level intermediate representation of the program and a structured graph(s); and the back-end is a target language dependent module that generates high-level target code from the intermediate representation and the structure of the graph. In this way, a decompiler for a different machine can be built by writing a new front-end for that machine, and a decompiler for a different target high-level language can be built by writing a new back-end for the target language. This approach is limited in practice by the choice of low-level intermediate language representation.

The significant contributions of this thesis are the types of analyses done in the universal decompiling machine: data flow analysis and control flow analysis, which transform the low-level (machine-like) intermediate code into a high-level (HLL-like) intermediate representation. The data flow analyzer describes optimization techniques based on compiler optimization principles, which eliminate the low-level concepts of condition codes and registers, and introduces the high-level concept of expression. These techniques take into account interprocedural analysis, register spilling, and type propagation. The control flow analyzer describes structuring algorithms to determine the underlying high-level control structures of the program. These algorithms structure the graph according to a predefined, generic set of control structures available in most commonly used languages.

The implementation of these techniques in the prototype decompiler **dcc** demonstrates the feasibility of the presented techniques. **dcc** is a decompiler for the DOS operating system and the Intel i80286 machine architecture which generates target C programs. This decompiler runs on a DecStation 3100 under Unix, and on Intel machines under DOS. **dcc** makes use of compiler and library signature recognition to decompile user routines only (whenever possible), rather than decompiling compiler start-up and library routines as well. Whenever a compiler signature is not determined, all subroutines available in the source binary program are decompiled; several of the library and compiler start-up routines are untranslatable into a high-level language representation and hence are disassembled only. **dcc** provides comments for each subroutine, and has command switches to generate the bitmap of the program, the call graph, an output assembler file, statistics on the number of low-level and high-level instructions in each subroutine, and information on the control flow graph of each subroutine.

Decompilation is used in two main areas of computer science: software maintenance and security. A decompiler is used in software maintenance to recover lost or inaccessible source code, translate code written in an obsolete language into a newer language, structure old code written in an unstructured way (i.e. spaghetti code), migrate applications to a new hardware platform, and debug binary programs that are known to have a bug. In security, a decompiler is used to verify binary programs and the correctness of the code produced by a compiler for safety-critical systems; where the compiler is not trusted to generate correct code; and to check for the existence of malicious code such as viruses.

Further work on decompilation can be done in two areas: the separation of code and data, and the determination of data types such as arrays, records, and pointers. The former area needs a robust method of determining n-way branch statements (i.e. indexed jumps) and indirect subroutine calls. The latter area needs heuristic methods to identify different types of compound data types and propagate their values. Efficient implementation of the algorithms would provide a faster decompiler, although the speed of decompilation is not a concern given that a program is normally decompiled once only.

Appendix A

i8086 – i80286 Architecture

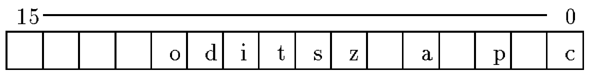
The Intel iAPX 8086, 8088, 80186 and 80286 machine architectures consist of the same type of registers, memory structure and input/output port organization[Int86, Int87]. These architectures are downwards compatible, hence the 80286 supports all machine instructions supported by the 8086 architecture. The registers of these 16-bit word machines are classified into five different sets according to their usage: data, pointer, index, control, and segment registers; this classification is shown in Figure A-1.

Type	Register	Function
Data	`ax`	accumulator
	`bx`	base register in some addressing modes
	`cx`	counter
	`dx`	general purpose
Pointer	`sp`	stack pointer
	`bp`	base pointer
Index	`si`	source
	`di`	destination
Control	`ip`	instruction pointer
	flags	flags or status word
Segment	`cs`	code segment
	`ds`	data segment
	`ss`	stack segment
	`es`	extra segment

Figure A-1: Register Classification

Data or general purpose registers can be accessed as word or byte registers. Each register has a high and low byte with the following naming convention: register names that replace the `x` by a `h` access the high byte of that register; and register names that replace the `x` by an `l` access the low byte of that register. The flags register is a special purpose register that keeps track of the condition codes set up by different instructions. The structure of this register is shown in Figure A-2. As can be seen, not all bits are used; unused bits are reserved by Intel.

Memory is structured as an array of 8-bit bytes stored in little-endian convention (i.e. most significant byte of a word is stored at the highest memory address). Memory is divided



c: carry
p: parity
a: auxiliary carry
z: zero
s: sign
t: trap
i: interrupt
d: direction
o: overflow

Figure A-2: Structure of the Flags Register

into banks of segments, each segment is a linear sequence of 64K bytes; therefore memory is addressed via a segment and offset pair.

Input/output port organization consists of up to 64Kb of 8-bit ports or 32Kb of 16-bit ports, located in a separate addressing space from the memory space.

A.1 Instruction Format

The length of an 80286 instruction varies from 1 up to 6 bytes. There are two types of opcodes: 1-byte opcodes and compound opcodes. 1-byte opcodes use the first byte of an instruction as the opcode, followed by the fields byte, at most 2 bytes of displacement, and at most 2 bytes of data. The fields byte contains information about registers, immediate operands, and/or displacement data. Compound opcodes store part of the opcode in the first byte of the instruction, and part in three bits of the second byte of the instruction (see Figure A-3). The first byte determines the group table to which the instruction belongs, and the 3-bit opcode of the second byte determines the index into the table (i.e. there are 8 entries into the table). The remaining bits of the second byte are used as the fields byte. The rest of the instruction is structured in the same way as for 1-byte opcodes[LG86].

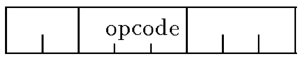


Figure A-3: Compound Opcodes' Second Byte

In the 80286, almost all byte combinations are valid opcodes. There are 229 1-byte opcodes, 29 compound-opcodes and 6 prefix instructions. A complete list of the machine language instructions, mnemonics and operands is found in Section A.2.

The fields byte is used to calculate the effective address (EA) of the operand. This byte is made up of 3 fields: the *reg* 3-bit field which takes the value of a register, the *r/m* 3-bit field which is used as a second register or a memory operand, and the *mod* 2-bit field which

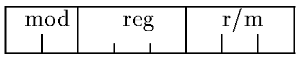


Figure A-4: The Fields Byte

determines the number of displacement bytes (DISP), whether *r/m* is used as a register or a memory operand, or the effective address of instructions that are not indexed nor based-indexed. The structure of this byte is shown in Figure A-4. An algorithm to interpret the fields byte is shown in Figure A-5.

```
case (mod) of {
 0:  if (r/m == 6)    /* get 2 bytes displacement */
       EA = dispHi:dispLo;
     else             /* no extra bytes */
       DISP = 0;
 1:  /* get 1 byte displacement */
     DISP = dispLo sign-extended to 16 bits;
 2:  /* get 2 bytes displacement */
     DISP = dispHi:dispLo;
 3:  /* Indexed */
     r/m is treated as a register field;
}
```

Figure A-5: Algorithm to Interpret the Fields Byte

The EA for indexed and based-indexed operands is calculated according to the *r/m* field; each value is mapped to an indexed register or a combination of indexed and based registers, as shown in Figure A-6.

Value of r/m	Indexed register(s)
0	bx + si
1	bx + di
2	bp + si
3	bp + di
4	si
5	di
6	bp
7	bx

Figure A-6: Mapping of r/m field

The final effective address is calculated as the addition of the displacement (DISP) and the register(s) given by the *r/m* bits.

Each combination of *mod, r/m* values uses a default segment register for its addressing, these default segments are shown in Figure A-7. Although the effective address of an operand is determined by the combination of the *mod, r/m* fields, the final physical address is calculated by adding the EA to the contents of the default segment register multiplied by 16. As a general rule, when the `bp` register is used, the default segment is `ss`, otherwise the default segment is `ds`.

r/m / mod	0	1	2
0	DS	DS	DS
1	DS	DS	DS
2	SS	SS	SS
3	SS	SS	SS
4	DS	DS	DS
5	DS	DS	DS
6	DS	SS	SS
7	DS	DS	DS

Figure A-7: Default Segments

The segment override prefix is a 1 byte opcode that permits exceptions to the default segment register to be used by the next instruction (i.e. it is only valid for 1 instruction; the one that follows it). The segment is determined by a 2-bit field (bits 3 and 4) of the prefix byte. All other fields take constant values, as illustrated in Figure A-8.

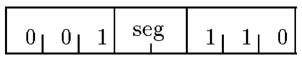


Figure A-8: Segment Override Prefix

There are two repeat prefix opcodes, `repne` and `repe`. These opcodes repeat the execution of the next instruction while register `cx` is not equal or equal to zero. They are normally used with string instructions such as `movs` and `ins` to repeat a condition while it is not end of string.

A.2 Instruction Set

The instruction set of the i80286 is described in terms of the machine opcode, the assembler mnemonic, and the assembler operands to the instruction. The following conventions are used to describe such an instruction set:

- reg8: 8-bit register.
- reg16: 16-bit register.
- mem8: 8-bit memory value.
- mem16: 16-bit memory value.
- immed8: 8-bit immediate value.
- immed16: 16-bit immediate value.
- immed32: 32-bit immediate value.
- segReg: 16-bit segment register.

Figure A-9 show all 1-byte opcodes. Compound opcodes are referenced as indexes into a table, each table has 8 posible values. The tables are shown in Figures A-10, A-11, A-12, and A-13. These figures are summaries of figures described in [Int86, Int87].

Machine Opcode	Assembler Mnemonic and Operands
00	ADD reg8/mem8,reg8
01	ADD reg16/mem16,reg16
02	ADD reg8,reg8/mem8
03	ADD reg16,reg16/mem16
04	ADD AL,immed8
05	ADD AX,immed16
06	PUSH es
07	POP es
08	OR reg8/mem8,reg8
09	OR reg16/mem16,reg16
0A	OR reg8,reg8/mem8
0B	OR reg16,reg16/mem16
0C	OR al,immed8
0D	OR ax,immed16
0E	PUSH cs
0F	Not used
10	ADC reg8/mem8,reg8
11	ADC reg16/mem16,reg16
12	ADC reg8,reg8/mem8
13	ADC reg16,reg16/mem16
14	ADC al,immed8
15	ADC ax,immed16
16	PUSH ss
17	POP ss
18	SBB reg8/mem8,reg8
19	SBB reg16/mem16,reg16
1A	SBB reg8,reg8/mem8
1B	SBB reg16,reg16/mem16
1C	SBB al,immed8
1D	SBB ax,immed16
1E	PUSH ds
1F	POP ds
20	AND reg8/mem8,reg8
21	AND reg16/mem16,reg16
22	AND reg8,reg8/mem8
23	AND reg16,reg16/mem16
24	AND al,immed8
25	AND ax,immed16
26	Segment override
27	DAA

Figure A-9: 1-byte Opcodes

Machine Opcode	Assembler Mnemonic and Operands
28	SUB reg8/mem8,reg8
29	SUB reg16/mem16,reg16
2A	SUB reg8,reg8/mem8
2B	SUB reg16,reg16/mem16
2C	SUB al,immed8
2D	SUB ax,immed16
2E	Segment override
2F	DAS
30	XOR reg8/mem8,reg8
31	XOR reg16/mem16,reg16
32	XOR reg8,reg8/mem8
33	XOR reg16,reg16/mem16
34	XOR al,immed8
35	XOR ax,immed16
36	Segment override
37	AAA
38	CMP reg8/mem8,reg8
39	CMP reg16/mem16,reg16
3A	CMP reg8,reg8/mem8
3B	CMP reg16,reg16/mem16
3C	CMP al,immed8
3D	CMP ax,immed16
3E	Segment override
3F	AAS
40	INC ax
41	INC cx
42	INC dx
43	INC bx
44	INC sp
45	INC bp
46	INC si
47	INC di
48	DEC ax
49	DEC cx
4A	DEC dx
4B	DEC bx
4C	DEC sp
4D	DEC bp
4E	DEC si
4F	DEC di

Figure A-9: 1-byte opcodes – Continued

Machine Opcode	Assembler Mnemonic and Operands
50	PUSH ax
51	PUSH cx
52	PUSH dx
53	PUSH bx
54	PUSH sp
55	PUSH bp
56	PUSH si
57	PUSH di
58	POP ax
59	POP cx
5A	POP dx
5B	POP bx
5C	POP sp
5D	POP bp
5E	POP si
5F	POP di
60	PUSHA
61	POPA
62	BOUND reg16/mem16,reg16
63	Not used
64	Not used
65	Not used
66	Not used
67	Not used
68	PUSH immed16
69	IMUL reg16/mem16,immed16
6A	PUSH immed8
6B	IMUL reg8/mem8,immed8
6C	INSB
6D	INSW
6E	OUTSB
6F	OUTSW
70	JO immed8
71	JNO immed8
72	JB immed8
73	JNB immed8
74	JZ immed8
75	JNZ immed8
76	JBE immed8
77	JA immed8

Figure A-9: 1-byte Opcodes – Continued

Machine Opcode	Assembler Mnemonic and Operands
78	JS immed8
79	JNS immed8
7A	JP immed8
7B	JNP immed8
7C	JL immed8
7D	JNL immed8
7E	JLE immed8
7F	JG immed8
80	Table2 reg8
81	Table2 reg16
82	Table2 reg8
83	Table2 reg8, reg16
84	TEST reg8/mem8,reg8
85	TEST reg16/mem16,reg16
86	XCHG reg8,reg8
87	XCHG reg16,reg16
88	MOV reg8/mem8,reg8
89	MOV reg16/mem16,reg16
8A	MOV reg8,reg8/mem8
8B	MOV reg16,reg16/mem16
8C	MOV reg16/mem16,segReg
8D	LEA reg16,reg16/mem16
8E	MOV segReg,reg16/mem16
8F	POP reg16/mem16
90	NOP
91	XCHG ax,cx
92	XCHG ax,dx
93	XCHG ax,bx
94	XCHG ax,sp
95	XCHG ax,bp
96	XCHG ax,si
97	XCHG ax,di
98	CBW
99	CWD
9A	CALL immed32
9B	WAIT
9C	PUSHF
9D	POPF
9E	SAHF
9F	LAHF

Figure A-9: 1-byte Opcodes – Continued

Machine Opcode	Assembler Mnemonic and Operands
A0	MOV al,[mem8]
A1	MOV ax,[mem16]
A2	MOV [mem8],al
A3	MOV [mem16],ax
A4	MOVSB
A5	MOVSW
A6	CMPSB
A7	CMPSW
A8	TEST al,[mem8]
A9	TEST ax,[mem16]
AA	STOSB
AB	STOSW
AC	LODSB
AD	LODSW
AE	SCASB
AF	SCASW
B0	MOV al,immed8
B1	MOV cl,immed8
B2	MOV dl,immed8
B3	MOV bl,immed8
B4	MOV ah,immed8
B5	MOV ch,immed8
B6	MOV dh,immed8
B7	MOV bh,immed8
B8	MOV ax,immed16
B9	MOV cx,immed16
BA	MOV dx,immed16
BB	MOV bx,immed16
BC	MOV sp,immed16
BD	MOV bp,immed16
BE	MOV si,immed16
BF	MOV di,immed16
C0	Table1 reg8
C1	Table1 reg8, reg16
C2	RET immed16
C3	RET
C4	LES reg16/mem16,mem16
C5	LDS reg16/mem16,mem16
C6	MOV reg8/mem8,immed8
C7	MOV reg16/mem16,immed16

Figure A-9: 1-byte Opcodes – Continued

Machine Opcode	Assembler Mnemonic and Operands
C8	ENTER immed16, immed8
C9	LEAVE
CA	RET immed16
CB	RET
CC	INT 3
CD	INT immed8
CE	INTO
CF	IRET
D0	Table1 reg8
D1	Table1 reg16
D2	Table1 reg8
D3	Table1 reg16
D4	AAM
D5	AAD
D6	Not used
D7	XLAT [bx]
D8	ESC immed8
D9	ESC immed8
DA	ESC immed8
DB	ESC immed8
DC	ESC immed8
DD	ESC immed8
DE	ESC immed8
DF	ESC immed8
E0	LOOPNE immed8
E1	LOOPE immed8
E2	LOOP immed8
E3	JCXZ immed8
E4	IN al,immed8
E5	IN ax,immed16
E6	OUT al,immed8
E7	OUT ax,immed16
E8	CALL immed16
E9	JMP immed16
EA	JMP immed32
EB	JMP immed8
EC	IN al,dx
ED	IN ax,dx
EE	OUT al,dx
EF	OUT ax,dx

Figure A-9: 1-byte Opcodes – Continued

Machine Opcode	Assembler Mnemonic and Operands
F0	LOCK
F1	Not used
F2	REPNE
F3	REP
F4	HLT
F5	CMC
F6	Table3 reg8
F7	Table3 reg16
F8	CLC
F9	STC
FA	CLI
FB	STI
FC	CLD
FD	STD
FE	Table4 reg8
FF	Table4 reg16

Figure A-9: 1-byte Opcodes – Continued

Index	Assembler Mnemonic
0	ROL
1	ROR
2	RCL
3	RCR
4	SHL
5	SHR
6	Not used
7	SAR

Figure A-10: Table1 Opcodes

Index	Assembler Mnemonic
0	ADD
1	OR
2	ADC
3	SBB
4	AND
5	SUB
6	XOR
7	CMP

Figure A-11: Table2 Opcodes

Index	Assembler Mnemonic
0	TEST
1	Not used
2	NOT
3	NEG
4	MUL
5	IMUL
6	DIV
7	IDIV

Figure A-12: Table3 Opcodes

Index	Assembler Mnemonic
0	INC
1	DEC
2	CALL
3	CALL
4	JMP
5	JMP
6	PUSH
7	Not used

Figure A-13: Table4 Opcodes

Appendix B

Program Segment Prefix

The program segment prefix or PSP is a 256-byte block of information, apparently a remnant of the CP/M operating system, that was adopted to assist in porting CP/M programs to the DOS environment[Dun88b]. When a program is loaded into memory, a PSP is built on the first 256 bytes of the allocated memory block. The fields of the PSP are shown in Figure B-1.

Segment offset	Description
00h	terminate vector: interrupt 20h (transfer to DOS)
02h	last segment allocated
04h	reserved
05h	call vector function: far call to DOS's function request handler
0Ah	copy of the parent's program termination handler vector
0Eh	copy of the parent's control-c/control-break handler vector
12h	copy of the parent's critical error handler vector
16H	reserved
2Ch	address of the first paragraph of the DOS environment
2Eh	reserved
50h	interrupt 21h, return far (`retf`) instruction
53h	reserved
5Ch	first parameter from the command line
6Ch	second parameter from the command line
80h	command tail; used as a buffer

Figure B-1: PSP Fields

The terminate vector (offset 00h of the PSP) used to be the warm boot/terminate (WBOOT) vector under CP/M. The call vector function (offset 05h of the PSP) used to be the basic disk operating system (BDOS) vector under CP/M.

Appendix C

Executable File Format

The DOS operating system supports two different types of executable files: **.exe** and **.com** files. The former allows for large programs and multiple segments to be used in memory, the latter for small programs that fit into one segment (i.e. 64Kb maximum) [Dun88b].

C.1 .exe Files

The **.exe** file consists of a header and a load module, as shown in Figure C-1. The file header consists of 28 bytes of fixed formatted area, and a relocation table which varies in size. The load module is a fully linked image of the program; there is no information on how to separate segments in the module since DOS ignores how the program is segmented.

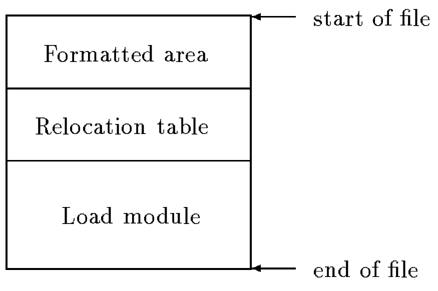


Figure C-1: Structure of an **.exe** File

The structure of the header's formatted area is shown in Figure C-2. The size of a page is 512 bytes, and the size of a paragraph is 16 bytes. The program image size is calculated from the value in the formatted area as the difference between the file size and the header size. The file size is given by the number of file pages (rounded up) and the size in bytes of the last page.

The relocation table is a list of pointers to words within the load module that must be adjusted. These words are adjusted by adding the start segment address where the program is to be loaded. Pointers in this table are stored as 2 words relative to the start of the load module.

Bytes	Description
00-01h	**.exe** signature (4Dh, 5Ah)
02-03h	number of bytes in the last page
04-05h	number of pages (rounded up)
06-07h	number of entries in the relocation table
08-09h	number of paragraphs in the header
0A-0Bh	minimum number of paragraphs required for data and stack
0C-0Dh	maximum number of memory paragraphs
0E-0Fh	pre-relocated initial `ss` value
10-11h	initial `sp` value (absolute value)
12-13h	complemented checksum (1's complement)
14-15h	initial `ip` value
16-17h	pre-relocated initial value of `cs`
18-19h	relocation table offset
1A-1B	overlay number (default: 0000h)

Figure C-2: Fixed Formatted Area

C.2 .com Files

A **.com** file is an image program without a header (i.e. equivalent to the load module of an **.exe** file), hence the program is loaded into memory "as is". As opposed to **.exe** programs, **.com** programs can only use one segment (up to 64Kb). These programs were designed to transport programs from CP/M into the DOS environment.

Appendix D

Low-level to High-level Icode Mapping

The mapping between low-level and high-level Icodes is shown in the following pages (Figure D-1). A dash (-) in the high-level Icode column means that there is no high-level counter part to the low-level icode, an asterisk (*) means that the low-level Icode forms part of a high-level instruction only when in an idiom, an f means that an Icode flag is set and the instruction is not considered any further, a cc means that the low-level instruction sets a condition code, it does not have a high-level counterpart, and is eliminated by condition code propagation, and an n means that the low-level Icode instruction was not considered in the analysis. Instructions marked with an n deal with machine string instructions, and were not considered in the analysis performed by **dcc**.

The initial mapping of low-level to high-level Icodes is expressed in terms of registers. Further data flow analysis on the Icodes transforms these instructions into expressions that do not make use of temporary registers, only variables and register variables (if any).

Low-level Icode	High-level Icode
iAAA	-
iAAD	-
iAAM	-
iAAS	-
iADC	*
iADD	`asgn` (`+`)
iAND	`asgn` (`&`)
iBOUND	f
iCALL	`call`
iCALLF	`call`
iCLC	cc
iCLD	cc
iCLI	-
iCMC	cc
iCMP	cc
iCMPS	n
iREPNE_CMPS	n
iREPE_CMPS	n
iDAA	-
iDAS	-
iDEC	`asgn` (`- 1`)
iDIV	`asgn` (`/`)
iENTER	f
iESC	f
iHLT	-
iIDIV	`asgn` (`/`)
iIMUL	`asgn` (*)
iIN	-
iINC	`asgn` (`+ 1`)
iINS	-
iREP_INS	-
iINT	-
iINTO	-
iIRET	-
iJB	`jcond` (`<`)
iJBE	`jcond` (`<=`)
iJAE	`jcond` (`>=`)
iJA	`jcond` (`>`)
iJE	`jcond` (`==`)
iJNE	`jcond` (`<>`)

Figure D-1: Icode Opcodes

Low-level Icode	High-level Icode
iJL	`jcond` (<)
iJGE	`jcond` (>=)
iJLE	`jcond` (<=)
iJG	`jcond` (>)
iJS	`jcond` (> 0)
iJNS	`jcond` (< 0)
iJO	-
iJNO	-
iJP	-
iJNP	-
iJCXZ	`jcond` (cx == 0)
iJNCXZ	`jcond` (cx <> 0)
iJMP	`jmp`
iJMPF	`jmp`
iLAHF	-
iLDS	`asgn` (far pointer)
iLEA	`asgn` (near pointer)
iLEAVE	`ret`
iLES	`asgn` (far pointer)
iLOCK	-
iLODS	n
iREP_LODS	n
iMOV	`asgn` (=)
iMOVS	n
iREP_MOVS	n
iMOD	`asgn` (%)
iMUL	`asgn` (*)
iNEG	`asgn` (-)
iNOT	!
iNOP	-
iOR	`asgn` (|)
iOUT	-
iOUTS	-
iREP_OUTS	-
iPOP	`pop`
iPOPA	-
iPOPF	-
iPUSH	`push`
iPUSHA	-
iPUSHF	-

Figure D-1: Icode Opcodes – Continued

Low-level Icode	High-level Icode
iRCL	*
iRCR	*
iREPE	n
iREPNE	n
iRET	`ret`
iRETF	`ret`
iROL	*
iROR	*
iSAHF	-
iSAR	*
iSHL	`asgn` (`<<`)
iSHR	`asgn` (`>>`)
iSBB	*
iSCAS	n
iREPNE_SCAS	n
iREPE_SCAS	n
iSIGNEX	`asgn` (`=`)
iSTC	cc
iSTD	cc
iSTI	-
iSTOS	n
iREP_STOS	n
iSUB	`asgn` (`-`)
iTEST	cc
iWAIT	f
iXCHG	`asgn` (uses `tmp`)
iXLAT	-
iXOR	`asgn` (`^`)

Figure D-1: Icode Opcodes – Continued

Appendix E

Comments and Error Messages displayed by dcc

dcc displays a series of comments in the output C and assembler files, on information collected during the analysis of each subroutine. This information is displayed before each subroutine. The following comments are supported by **dcc**:

- "Takes %d bytes of parameters."
- "Uses register arguments:" (and lists the registers and the formal argument name).
- "Takes no parameters."
- "Runtime support routine of the compiler."
- "High-level language prologue code."
- "Untranslatable routine. Assembler provided."
- "Return value in register %s." (register(s) provided).
- "Pascal calling convention."
- "C calling convention."
- "Unknown calling convention."
- "Incomplete due to an unstranslatable opcode"
- "Incomplete due to an indirect jump"
- "Indirect call procedure."
- "Contains self-modifying code."
- "Contains coprocessor instructions."
- "Irreducible control flow graph."

Assembler subroutines are also commented, as well as all DOS kernel services; interrupts 20h to 2Fh. Appendix F contains a list of all DOS interrupts supported by **dcc**.

dcc also displays two different types of errors: fatal and non fatal errors. Fatal errors terminate the execution of **dcc**, displaying the error with enough information to determine what happened. Non fatal errors do not cause **dcc** to terminate, and are treated as warnings to the user.

The fatal errors supported by **dcc** are:

- "Invalid option -%c."
- "Usage: dcc [-a1a2mpsvV][-o asmfile] DOS_executable"
- "New EXE format not supported."
- "Cannot open file %s."
- "Error while reading file %s."
- "Invalid instruction %02X at location %06lX."
- "Don't understand 80386 instruction %02X at location %06lX."
- "Instruction at location %06lX goes beyond loaded image."
- "malloc of %ld bytes failed."
- "Failed to find a basic block for jump to %ld in subroutine %s."
- "Basic Block is a synthetic jump."
- "Failed to find a basic block for interval."
- "Definition not found for condition code usage at opcode %d."

The non fatal errors supported by **dcc** are:

- "Segment override with no memory operand at location %06lX."
- "REP prefix without a string instruction at location %06lX."
- "Conditional jump use, definition not supported at opcode %d."
- "Definition-use not supported. Definition opcode = %d, use opcode = %d."
- "Failed to construct do..while() condition."
- "Failed to construct while() condition."

Appendix F

DOS Interrupts

The DOS kernel provides services to application programs via software interrupts 20h..2Fh. Interrupt 21h deals with character input/output, files, records, directory operations, disk, processes, memory management, network functions, and miscellaneous system functions; the function number is held in register `ah`. Figure F-1 lists the different interrupts provided by DOS [Dun88a]. These interrupts are commented by **dcc** when producing the disassembly of a subroutine.

Interrupt	Function	Function name
20h		Terminate process
21h	0h	Terminate process
21h	1h	Character input with echo
21h	2h	Character output
21h	3h	Auxiliary input
21h	4h	Auxiliary output
21h	5h	Printer output
21h	6h	Direct console input/output
21h	7h	Unfiltered character input without echo
21h	8h	Character input without echo
21h	9h	Display string
21h	Ah	Buffered keyboard input
21h	Bh	Check input status
21h	Ch	Flush input buffer and then input
21h	Dh	Disk reset
21h	Eh	Select disk
21h	Fh	Open file
21h	10h	Close file
21h	11h	Find first file
21h	12h	Find next file
21h	13h	Delete file
21h	14h	Sequential read
21h	15h	Sequential write
21h	16h	Create file
21h	17h	Rename file
21h	18h	Reserved
21h	19h	Get current disk
21h	1Ah	Set DTA address
21h	1Bh	Get default drive data
21h	1Ch	Get drive data
21h	1Dh	Reserved
21h	1Eh	Reserved
21h	1Fh	Reserved
21h	20h	Reserved
21h	21h	Random read
21h	22h	Random write
21h	23h	Get file size
21h	24h	Set relative record number
21h	25h	Set interrupt vector
21h	26h	Create new PSP
21h	27h	Random block read
21h	28h	Random block write

Figure F-1: DOS Interrupts

Interrupt	Function	Function name
21h	29h	Parse filename
21h	2Ah	Get date
21h	2Bh	Set date
21h	2Ch	Get time
21h	2Dh	Set time
21h	2Eh	Set verify flag
21h	2Fh	Get DTA address
21h	30h	Get DOS version number
21h	31h	Terminate and stay resident
21h	32h	Reserved
21h	33h	Get or set break flag
21h	34h	Reserved
21h	35h	Get interrupt vector
21h	36h	Get drive allocation info
21h	37h	Reserved
21h	38h	Get or set country info
21h	39h	Create directory
21h	3Ah	Delete directory
21h	3Bh	Set current directory
21h	3Ch	Create file
21h	3Dh	Open file
21h	3Eh	Close file
21h	3Fh	Read file or device
21h	40h	Write file or device
21h	41h	Delete file
21h	42h	Set file pointer
21h	43h	Get or set file attributes
21h	44h	IOCTL (input/output control)
21h	45h	Duplicate handle
21h	46h	Redirect handle
21h	47h	Get current directory
21h	48h	Alloate memory block
21h	49h	Release memory block
21h	4Ah	Resize memory block
21h	4Bh	Execute program (exec)
21h	4Ch	Terminate process with return code
21h	4Dh	Get return code
21h	4Eh	Find first file
21h	4Fh	Find next file
21h	50h	Reserved
21h	51h	Reserved
21h	52h	Reserved
21h	53h	Reserved

Figure F-1: DOS Interrupts – Continued

Interrupt	Function	Function name
21h	54h	Get verify flag
21h	55h	Reserved
21h	56h	Rename file
21h	57h	Get or set file date and time
21h	58h	Get or set allocation strategy
21h	59h	Get extended error info
21h	5Ah	Create temporary file
21h	5Bh	Create new file
21h	5Ch	Lock or unlock file region
21h	5Dh	Reserved
21h	5Eh	Get machine name
21h	5Fh	Device redirection
21h	60h	Reserved
21h	61h	Reserved
21h	62h	Get PSP address
21h	63h	Get DBCS lead byte table
21h	64h	Reserved
21h	65h	Get extended country info
21h	66h	Get or set code page
21h	67h	Set handle count
21h	68h	Commit file
21h	69h	Reserved
21h	6Ah	Reserved
21h	6Bh	Reserved
21h	6Ch	Extended open file
22h		Terminate handler address
23h		Ctrl-C handler address
24h		Critical-error handler address
25h		Absolute disk read
26h		Absolute disk write
27h		Terminate and stay resident
28h		Reserved
29h		Reserved
2Ah		Reserved
2Bh		Reserved
2Ch		Reserved
2Dh		Reserved
2Eh		Reserved
2Fh	1h	Print spooler
2Fh	2h	Assign
2Fh	10h	Share
2Fh	B7h	Append

Figure F-1: DOS Interrupts – Continued

Bibliography

[AC72] F.E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report RC 3923 (No. 17789), IBM, Thomas J. Watson Research Center, Yorktown Heights, New York, July 1972.

[AC76] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[All70] F.E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.

[All72] F.E. Allen. A basis for program optimization. In *Proc. IFIP Congress*, pages 385–390, Amsterdam, Holland, 1972. North-Holland Pub.Co.

[All74] F.E. Allen. Interprocedural data flow analysis. In *Proc. IFIP Congress*, pages 398–402, Amsterdam, Holland, 1974. North-Holland Pub.Co.

[AM71] E. Ashcroft and Z. Manna. The translation of 'go to' programs to 'while' programs. Technical report, Stanford University, Department of Computer Science, 1971.

[ASU86a] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[ASU86b] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 10, pages 585–722. Addison-Wesley Publishing Company, 1986.

[Bak77] B.S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[Bar74] P. Barbe. The Piler system of computer program translation. Technical report, Probe Consultants Inc., September 1974.

[BB91] J. Bowen and P. Breuer. Decompilation techniques. Internal to ESPRIT REDO project no. 2487 2487-TN-PRG-1065 Version 1.2, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, March 1991.

[BB92] P.T. Breuer and J.P. Bowen. Decompilation: The enumeration of types and grammars. Technical Report PRG-TR-11-92, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, 1992.

[BB93] P.T. Breuer and J.P. Bowen. Decompilation: the enumeration of types and grammars. To appear in Transaction of Programming Languages and Systems, 1993.

[BB94] P.T. Breuer and J.P. Bowen. Generating decompilers. To appear in Information and Software Technology Journal, 1994.

[BBL91] J.P. Bowen, P.T. Breuer, and K.C. Lano. The REDO project: Final report. Technical Report PRG-TR-23-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, December 1991.

[BBL93] J. Bowen, P. Breuer, and K. Lano. A compendium of formal techniques for software maintenance. *Software Engineering Journal*, pages 253–262, September 1993.

[BJ66] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[Bor92] Borland. *Borland C++ Version 3.1 - User's Guide.* Borland International, 1800 Green Hills Road, Scotts Valley, CA 95066, 1992.

[Bow91] J. Bowen. From programs to object code using logic and logic programming. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, Workshops in Computing, pages 173–192, Dagstuhl, Germany, May 1991. Springer Verlag.

[Bow93] J. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, 5(4):205–234, 1993.

[BP79] D. Balbinot and L. Petrone. Decompilation of Polish code in Basic. *Rivista di Informatica*, 9(4):329–335, October 1979.

[BP81] M.N. Bert and L. Petrone. Decompiling context-free languages from their Polish-like representations. pages 35–57, 1981.

[Bri81] D.L. Brinkley. Intercomputer transportation of assembly language software through decompilation. Technical report, Naval Underwater Systems Center, October 1981.

[Bri93] Brillo brillig. *EXE*, 8(5):6, October 1993.

[BZ80] A.L. Baker and S.H. Zweben. A comparison of measures of control flow complexity. *IEEE Transactions on Software Engineering*, SE-6(6):506–512, November 1980.

[Cal] C.A. Calkins. Masterful disassembler. Public domain software. Anonymous ftp oak.oakland.edu:SimTel/msdos/disasm/md86.zip.

[CCHK90] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[CG93] C. Cifuentes and K.J. Gough. A methodology for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informática*, pages 257–266, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informática.

[CG94] C. Cifuentes and K.J Gough. Decompilation of binary programs. Technical Report 3/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994. (To appear in 1995, Software – Practice & Experience).

[Chr80] W. Christensen. Resource for the 8086. Public domain software. Anonymous ftp oak.oakland.edu:SimTel/msdos/disasm/res86.zip, 1980. Translated to the 8086 by Larry Etienne.

[Cif93] C. Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informática*, pages 267–276, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informática.

[Cif94a] C. Cifuentes. Interprocedural data flow decompilation. Technical Report 4/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994. (Submitted to Journal of Programming Languages).

[Cif94b] C. Cifuentes. Structuring decompiled graphs. Technical Report 5/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994. (Submitted to The Computer Journal).

[Cob93] Cobol decompiler. *Midrange Systems*, 6(13):66, July 1993.

[Coc70] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–25, July 1970.

[Col81] J.W. Perry Cole. *ANSI Fortran IV, A structured programming approach.* WM.C.Brown, 2460 Kerper Boulevard, Dubuque, Iowa 52001, 8 edition, 1981.

[Com91] V Communications. *Sourcer - Commenting Disassembler, 8088 to 80486 Instruction Set Support.* V Communications, Inc, 4320 Stevens Creek Blvd., Suite 275, San Jose, CA 95129, 1991.

[Coo67] D.C. Cooper. Böhm and Jacopini's reduction of flow charts. *Communications of the ACM*, 10(8):463,473, August 1967.

[Coo83] D. Cooper. *Standard Pascal, User Reference Manual.* W.W.Norton & Company, 500 Fifth Avenue, New York, N.Y. 10110, 1 edition, 1983.

[Dav58] M. Davis. *Computability and Unsolvability*, chapter 5, pages 69–70. McGraw-Hill, 1958.

[DoD83] American National Standards Institute, Inc., Department of Defense, OUSD(R&E), Washington, DC 20301, USA. *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983*, February 1983.

[DS82] L.M. Dorsey and S.Y. Su. The decompilation of COBOL-DML programs for the purpose of program conversion. In *Proceedings of COMPSAC 82. IEEE Computer Society's Sixth International Computer Software and Applications Conference*, pages 495–504, Chicago, USA, November 1982. IEEE.

[Dud82] R. Dudley. A recursive decompiler. *FORTH Dimensions*, 4(2):28, Jul–Aug 1982.

[Dun88a] R. Duncan. *Advanced MSDOS programming*. Microsoft Press, 16011 NE 36th Way, Box 97017, Redmond, Washington 98073-9717, 2 edition, 1988.

[Dun88b] R. Duncan. *The MSDOS Encyclopedia*, chapter 4, pages 107–147. Microsoft Press, 1988.

[DZ89] D.M. Dejean and G.W. Zobrist. A definition optimization technique used in a code translation algorithm. *Communications of the ACM*, 32(1):94–104, January 1989.

[EH93] A.M. Erosa and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. Technical Report ACAPS Technical Memo 76, School of Computer Science, McGill University, 3480 University St, Montreal, Canada H3A 2A7, September 1993.

[Emm94] M. Van Emmerik. Signatures for library functions in executable files. Technical Report 2/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994.

[FJ88a] C.N. Fischer and R.J. LeBlanc Jr. *Crafting a Compiler*. Benjamin Cummings, 2727 Sand Hill Road, Menlo Park, California 94025, 1988.

[FJ88b] C.N. Fischer and R.J. LeBlanc Jr. *Crafting a Compiler*, chapter 16, pages 609–680. Benjamin Cummings, 2727 Sand Hill Road, Menlo Park, California 94025, 1988.

[Flu89] B.H. Flusche. The art of disassembly; getting to the source of the problem when the object's the data. *Micro Cornucopia*, (46):8–14, March – April 1989.

[Fre91] P. French. Private communication. Email, 1991.

[Fri74] F.L. Friedman. *Decompilation and the Transfer of Mini-Computer Operating Systems*. PhD dissertation, Purdue University, Computer Science, August 1974.

[FZ91] C. Fuan and L. Zongtian. C function recognition technique and its implementation in 8086 C decompiling system. *Mini-Micro Systems*, 12(11):33–40,47, 1991.

[FZL93] C. Fuan, L. Zongtian, and L. Li. Design and implementation techniques of the 8086 C decompiling system. *Mini-Micro Systems*, 14(4):10–18,31, 1993.

[Gai65] R.S. Gaines. On the translation of machine language programs. *Communications of the ACM*, 8(12):737–741, December 1965.

[Gar88] P.D. Garnett. Selective disassembly: a first step towareds developing a virus filter. *Fourth Aerospace Computing Security Appl Conference*, pages 2–6, December 1988.

[GCC$^+$92] K.J. Gough, C. Cifuentes, D. Corney, J. Hynd, and P. Kolb. An experiment in mixed compilation/interpretation. In G.K.Gupta and C.D.Keen, editors, *Proceedings of the Fifteenth Australian Computer Science Conference (ACSC-15)*, pages 315–327, University of Tasmania, Hobart, Australia, 29-31 January 1992. Australian Computer Society.

[GD83] J. Gersbach and J. Damke. Asmgen, version 2.01. Public domain software. Anonymous ftp oak.oakland.edu:SimTel/msdos/disasm/asmgen3.zip, 1983.

[GL82] L. Goldschlager and A. Lister. *Computer Science: A modern introduction.* Prentice-Hall International, 1982.

[Gou88] K.J. Gough. *Syntax Analysis and Software Tools.* Addison Wesley Publishing Company, Reading, U.K., 1988.

[Gou93] K.J. Gough. Private communication, 1993.

[Gut90] S. Guthery. exe2c. News item in comp.compilers USENET newsgroup, 30 Apr 1990.

[Gut91a] S. Guthery. exe2c. News item in comp.compilers USENET newsgroup, 23 Apr 1991.

[Gut91b] S. Guthery. Private communication. Austin Code Works, 11100 Leafwood Lane, Austin, TX 78750-3587, 14 Dec 1991.

[Hal62] M.H. Halstead. *Machine-independent computer programming*, chapter 11, pages 143–150. Spartan Books, 1962.

[Hal67] M.H. Halstead. Machine independence and third generation computers. In *Proceedings SJCC (Sprint Joint Computer Conference)*, pages 587–592, 1967.

[Hal70] M.H. Halstead. Using the computer for program conversion. *Datamation*, pages 125–129, May 1970.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc, 52 Vanderbilt Avenue, New York, New York 10017, 1977.

[HH73] B.C. Housel and M.H. Halstead. A methodology for machine language decompilation. Technical Report RJ 1316 (#20557), Purdue University, Department of Computer Science, December 1973.

[HHB$^+$93] P. Heiser, D. Hanson, C. Berntson, K. Everett, and A. Schwartz. Feedback. *Data Based Advisor*, 11(6):14–15, June 1993.

[HK92] M.W. Hall and K. Kennedy. Efficient call graph analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[HM79] R.N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[HM84] N.L. Hills and D. Moines. Revisited: Recursive decompiler. *FORTH Dimensions*, 5(6):16–18, Mar–Apr 1984.

[Hol73] C.R. Hollander. *Decompilation of Object Programs.* PhD dissertation, Stanford University, Computer Science, January 1973.

[Hoo91] S.T. Hood. Decompiling with definite clause grammars. Technical Report ERL-0571-RR, Electronics Research Laboratory, DSTO Australia, PO Box 1600, Salisbury, South Australia 5108, September 1991.

[Hop78] G.L. Hopwood. *Decompilation.* PhD dissertation, University of California, Irvine, Computer Science, 1978.

[Hou73] B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages.* PhD dissertation, Purdue University, Computer Science, August 1973.

[HU72] M.S. Hecht and J.D. Ullman. Flow graph reducibility. *SIAM Journal of Computing*, 1(2):188–202, June 1972.

[HU74] M.S. Hecht and J.D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[HU75] M. Hecht and J. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal of Computing*, 4(4):519–532, December 1975.

[HZY91] L. Hungmong, L. Zongtian, and Z. Yifen. Design and implementation of the intermediate language in a PC decompiler system. *Mini-Micro Systems*, 12(2):23–28,46, 1991.

[Int86] Intel. *iAPX 86/88, 186/188 User's Manual.* Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051, 1986.

[Int87] Intel. *80286 and 80287 Programmer's Reference Manual.* Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051, 1987.

[KF71] D.E. Knuth and R.W. Floyd. Notes on avoiding go to statements. *Information Processing Letters*, 1(1):23–31, 1971.

[Knu74] D.E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, December 1974.

[Kos74] S.R. Kosaraju. Analysis of structured programs. *Journal of Computer and System Sciences*, 9(3):232–255, 1974.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc, Englewood Cliffs, N.J., 2 edition, 1988.

[KW82] R.H. Katz and E. Wong. Decompiling CODASYL DML into relational queries. *ACM Transactions on Database Systems*, 7(1):1–23, March 1982.

[LG86] Y.C. Liu and G.A. Gibson. *Microcomputer Systems: the 8086/8088 family.* Prentice-Hall International, New Jersey, 2 edition, 1986.

[Lic85] U. Lichtblau. Decompilation of control structures by means of graph transformations. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Berlin, 1985.

[Lic91] U. Lichtblau. Recognizing rooted context-free flowgraph languages in polynomial time. In G. Rozenberg H. Ehrig, H.J. Kreowski, editor, *Graph Grammars and their application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 538–548. Springer-Verlag, 1991.

[LM75] H.F. Ledgard and M. Marcotty. A genealogy of control structures. *Communications of the ACM*, 18(11):629–639, November 1975.

[Mak90] O.J. Makela. Intelligent disassembler, version 1.2. Public domain software. Anonymous ftp oak.oakland.edu:SimTel/msdos/disasm/id12.zip, 1990.

[May88] W. May. A simple decompiler. *Dr.Dobb's Journal*, pages 50–52, June 1988.

[Mic87] Microsoft. *Mixed-Language Programming Guide*. Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717, 1987.

[MZBR85] W.K. Marshall, G.W. Zobrist, W. Bach, and A. Richardson. A functional mapping for a microprocessor system simulation. In Proceedings of the 1985 IEEE Microprocessor Forum (Atlantic City, N.J., Apr.2-4). IEEE Piscataway, N.J. pp.37-39, 1985.

[Oul82] G. Oulsnam. Unravelling unstructured programs. *The Computer Journal*, 25(3):379–387, 1982.

[Out93] Tools and utilities. *DBMS*, 6(7):63–92, June 1993.

[PLA91] Programming Languages and Systems Group – Queensland University of Technology, GPO Box 2434, Brisbane, QLD 4001, Australia. *Gardens Point Modula-2*, 1991.

[PW93] D.J. Pavey and L.A. Winsborrow. Demonstrating equivalence of source code and PROM contents. *The Computer Language*, 36(7):654–667, 1993.

[Ram88] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.

[Ray89] E.S. Raymond. Plum-hall benchmarks. URL: ftp//plaza.aarnet.edu.au/usenet/comp.sources.unix/volume20/plum-benchmarks.gz, 1989.

[Reu88] J. Reuter. URL: ftp//cs.washington.edu/ pub/decomp.tar.z. Public domain software, 1988.

[Reu91] J. Reuter. Private communication. Email, 1991.

[Ryd79] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–226, May 1979.

[Sas66] W.A. Sassaman. A computer program to translate machine language into fortran. In *Proceedings SJCC*, pages 235–239, 1966.

[SCK+93] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[Sha80] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

[Sof88] RH Factor Software. Bubble chamber installation. Public domain software, beta release. Anonymous ftp oak.oakland.edu:SimTel/msdos/disasm/bubble.zip, 1988.

[SW74] V. Schneider and G. Winiger. Translation grammars for compilation and decompilation. *BIT*, 14:78–86, 1974.

[SW93] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[Tar74] R.E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, pages 355–365, September 1974.

[Unc93] Your guide to clipper add-ons. *Data Based Advisor*, 11(8):47–67, August 1993.

[Val93] Slicing thru source code. *Data Based Advisor*, 11(3):28, March 1993.

[WG85] M.H. Williams and G.Chen. Restructuring pascal programs containing goto statements. *The Computer Journal*, 28(2):134–137, 1985.

[Wil77] M.H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal*, 20(1):45–50, 1977.

[Wir85] N. Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin Heidelberg, 3 edition, 1985.

[WO78] M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured form. *The Computer Journal*, 21(2):161–167, 1978.

[Wor78] D.A. Workman. Language design using decompilation. Technical report, University of Central Florida, December 1978.

[Wor91] Austin Code Works. exe2c. β release, 1991. Email: info@acw.com.

[Yoo85] C.W. Yoo. An approach to the transportation of computer software. *Information Processing Letters*, 21:153–157, September 1985.

[Zan85] J.R. Van Zandt. Interactive 8086 disassembler, version 2.21. Public domain software. Anonymous ftp oak.oakland.edu:SimTel/msdos/disasm/dis86221.zip, 1985.