Draft: Natural Language Processing for the Working Programmer

Harm Brouwer
Universität des Saarlandes
43 PUBLICATIONS **430** CITATIONS

Natural Language Processing for the Working Programmer

Daniël de Kok

Harm Brouwer

Natural Language Processing for the Working Programmer

Daniël de Kok
Harm Brouwer

License

Table of Contents

List of Figures

List of Tables

List of Equations

Preface

Acknowledgements

We would like to thank:

- SyncRO Soft Ltd. for their <oXygen/> XML Editor [http://www.oxygenxml.com/], which made writing of this book very pleasant.

- The Computational Linguistics group [http://www.rug.nl/let/onderzoek/onderzoekinstituten/clcg/onderzoek/compuling/index] at the Center of Language and Cognition Groningen [http://www.rug.nl/let/onderzoek/onderzoekinstituten/clcg/index] for their encouragement and support. But most of all, for introducing us to natural language processing in the first place.

Chapter 1. Introduction

Welcome

The Internet and the World Wide Web have changed mankind, forever. It is to early too tell, but their impact may be as great as the combustion engine or the introduction of electric devices. The Internet gave universal access to information, not just information that broadcasters or newspapers thought that was important, but information that interests the 'websurfer'. However the Internet is not a one-way street, every Internet user is also a producer: people make websites, maintain blogs, post tweets, and socialize via social networks.

Since a substantial part of the world has Internet access, and every user is also a producer, there is an enormous amount of information available. Some of it is of peer-reviewed and of a high quality, most of it is unchecked and biased. Still, every single piece of information can contain valuable information. For instance, as a vacuum cleaner producer, you might think that social media are not so interesting. However, the contrary is true: between billions of messages there may be hundreds expressing sentiment about your product. Such messages can answer questions about how your brand is conceived, what problems people commonly have with your product, etc.

Obviously, it is out of anyone's reach to manually analyze a significant portion of the information that is available on the Internet. The amount is just too overwhelming. Classic data analysis tools may not suffice either, most of the information is seemingly unstructured, and consist of blobs of natural language sentences. However, language is also structure. It is just not the kind of structure that computers can normally deal with. Computers deal with neat XML files, fragments of JSON, or comma separated values. No, it is the structure that we humans use to convey and transfer meaning. This book is about that type of information. We will go into many of the techniques that so-called *computational linguists* use to analyze the structure of human language, and transform it into a form that computers work with.

What is natural language processing?

Stub

What is Haskell?

Haskell is a static, pure, lazy, functional language. Gee, that sounds an awful lot like buzzword lingo! That may well be, but these properties make Haskell a very effective language, and sometimes a bit odd. These properties are also opposite to most mainstream languages. For instance, Java is static, impure, eager, and not functional, Ruby is dynamic, impure, eager, and only functional at times. While it is hard to give an accurate definition of every characteristic, we will try anyway.

Functional: Haskell puts an emphasis on functions and treats computation as the evaluation of functions. This in contrast to so-called imperative languages, that specify an order of instructions. As such, Haskell functions very often resemble mathematical functions closely. Being functional also has practical implications. For instance, iteration is accomplished by means of recursion, and functions are also values and can be passed to other functions.

Pure: Haskell is a pure language, in that functions do not have side-effects. This means that existing values cannot be changed, since changing data would be a side-effect of a function. It also guarantees that the evaluation of a function will always result in the same value given the same function arguments.

Lazy: As a lazy language, Haskell only evaluates expressions when needed. Say you just implemented a function that gives a list of all prime numbers. In a strict language, the function that makes the list will never terminate (since there are always more prime numbers). Haskell, on the contrary, will only evaluate this function when and as much as necessary. As long as you take a finite number of primes from the list, the program will happily terminate.

Static: Haskell programs are compiled before they can run. This means that the Haskell compiler will catch many errors for you at compile-time, rather than finding them when your program is used in production. Additionally, Haskell does *type-inference*. This means that the Haskell compiler can find out the types of values most of the times, and you do not need to type-annotate every value.

If you have prior programming experience in an imperative or eager language, Haskell can feel a bit odd in the beginning. Don't worry, you will feel warm and fuzzy eventually!

You may ask why we chose Haskell as the main programming language for this book, rather than a more mainstream language. During our own experiences developing natural language processing tools, we noticed that very many natural language processing tasks are relatively straightforward data transformations. Haskell is a language that is exceptionally good at data transformations. First of all, because it has *higher order functions* (functions that take functions as an argument) that traverse lists, sets, mappings, etc. Second, Haskell makes it easy to construct more complex transformations out of simple transformations.

Although this book does not provide a full introduction to the Haskell programming language, we try to cover Haskell concepts extensively when required. If you require more background on certain concepts, we recommend you to consult the book *Learn Haskell for Great Good! [http://learnyouahaskell.com/]*

What you need

To work with this book, you need the Haskell Platform and a text editor. The Haskell Platform is available for Mac OS X, Windows, and Linux at: http://hackage.haskell.org/platform/ Download the package for your platform, and install it.

If you do not use one of these platforms, not all is lost. First of all, you should download the ghc Haskell compiler. If you operating system provides a ports system or package manager, use it to locate and install a ghc package. If your operating system does not provide a port or package for ghc, you can still try to download a binary distribution from the GHC website [http://www.haskell.org/ghc/]. Once you have installed and set up ghc, install the packages of the Haskell Platform.

For the text editor, pick any editor you find comfortable, as long as it saves files as plain text. Programming Haskell becomes more comfortable if you have an editor with syntax highlighting and interpreter integration. Your authors prefer the Emacs editor with *haskell-mode*. *haskell-mode* provides good support for highlighting, and Haskell code formatting. Besides that, Emacs allows you to run the **ghci** Haskell interpreter within the editor.

Some examples in the book use corpora. These corpora are available at: http://nlpwp.org/

Ready, set, go!

You will probably want to get acquainted with the **ghci** Haskell interpreter. It allows you to try Haskell expressions and get immediate feedback. On UNIX systems, type **ghci -XNoMonomorphismRestriction** in a terminal to launch the interpreter. You will be greeted with a prompt that resembles the following:

```
$ ghci -XNoMonomorphismRestriction
GHCi, version 7.0.2: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

On Windows, choose *Start -> All Programs -> Haskell Platform -> WinGHCi*. The first time, you should set the *NoMonomorphismRestriction* option that is required for some examples. Do this by going to the following item: *File > Options > GHCi Startup*. Then append -*XNoMonomorphismRestriction* in the field (make sure that there is a space before this addition). Click *OK*, and restart WinGHCi.

Chapter 2. Words

Introduction

Words are the most fundamental building blocks of our language. Although they may look simple on the surface, they are very ingenious devices that pack not only meaning, but also grammatical information. For our purposes, we will say that a word consists of a *stem* and *affixes*. Let's look at three simple sentences:

- I **walk**.
- John **walk**s.
- Jack **walk**ed.

All three sentences contain some 'form' of *walk*. We say that these instances are all *inflections* of the verb walk. The part of the inflections that is shared (*walk*) is what we call the *stem*. The parts that are not common are named *affixes*. We inflect verbs to indicate tense (present, past, etc.), the person of the verb's subject (first, second, and third), and the number (singular or plural). The affix *s* in John walk*s*, for instance, tells us (in combination with the subject John) that the verb *walk* is in present tense, third person singular.

Other types of words have inflections as well. For example, we inflect nouns to distinguish singular and plural:

I saw one **duck**. I saw two **duck**s.

Up to this point, we have just seen one kind of affix: one that is glued to the end of the word. There are actually many types of affixes. For now, you should only know about two:

- Prefix: appears in front of the stem. For example, **un**believable.
- Suffix: appears after the stem. For example, duck**s**.

Now, with that out of the way, let's get some work done.

Playing with words

Written words consist of characters. We can write down characters in Haskell with single quotes. If you type in a character in *ghci*, it will simply echo back the character:

```
Prelude> 'h'
'h'
```

This is all that *ghci* does, it evaluates whatever you type. A character evaluates to... a character. We confirm that Haskell agrees with us that this actually a character by asking the type with *:type* or its shorthand *:t*:

```
Prelude> :type 'h'
'h' :: Char
```

Great. Haskell indeed confirms that 'h' is a character, or in Haskell's words: that 'h' is of type *Char*. Not all that practical with the small amount of single-lettered words in English though. Rather than a single character, we want a sequence of characters. Not surprisingly, Haskell has a data types to build sequences. The most commonly used sequence is the list. You can have lists of many things: lists of groceries, lists of planets, but also lists of characters. We can make a literal list in Haskell by enumerating its elements, separated by commas and surrounded by square brackets. For instance, the list *1, 2, 3, 4, 5* is written as *[1, 2, 3, 4, 5]*. Let's try to make a list of characters:

```
Prelude> ['h','e','l','l','o']
"hello"
```

Now we are getting somewhere! Let's look at the type of this list:

```
Prelude> :type ['h','e','l','l','o']
['h','e','l','l','o'] :: [Char]
```

Its type is *[Char]*, which should be read as 'list of characters'. Such a list of characters is known as *a string* Of course, writing down words in this manner is not very convenient. Fortunately, as the evaluation of the second to last example already suggests, there is a more convenient notation. We can represent strings by wrapping characters in double quotes:

```
Prelude> "hello"
"hello"
Prelude> :type "hello"
"hello" :: [Char]
```

We will take this opportunity to seriously demolish some words, but all with the noble cause of learning some commonly-used Haskell list functions. The first function `length` returns the length of a list:

```
Prelude> length "hello"
5
Prelude> length [1,2,3]
3
```

To get a better impression of functions, it is often useful to look at its type:

```
Prelude> :type length
length :: [a] -> Int
```

That's one heck of a type! Basically, it says 'give me a list of something (denoted by the *a* between the list brackets), then I will give you an Int'. In these so-called *type signatures*, letters that are not capitalized are generic, meaning that they can be of some unspecified type. That is, *[a]* is a list with elements of some type. **But:** all elements should be of the same type. An *Int* is an integral number: a positive or negative whole number.

Two other basic list functions are `head` and `tail`. `head` returns the first element of a list, `tail` everything but the first element:

```
Prelude> head "hello"
'h'
Prelude> tail "hello"
"ello"
```

The type of head is the following:

```
Prelude> :type head
head :: [a] -> a
```

Hey, two *a*'s! Equipped with the knowledge we have, we know that `head` is a function that takes a list of something, and gives back something. But there is an additional constraint here: although *a* is some type, all *a*'s have to be the same type. So, applying `head` to a list of numbers gives a number, applying `head` to a list of characters gives a character, etc.

In analogy, the type of `tail` should now be easy to understand:

```
Prelude> :type tail
tail :: [a] -> [a]
```

We apply `tail` to a list of some type, and get back a list with the same type.

Finally, the last function for now is `reverse`. We have to admit presenting this function with a bit of joy, since it will allow us to write our first little useful Haskell program. As expected, `reverse` reverses the elements of a list:

```
Prelude> reverse "hello"
"olleh"
```

Olé! And another one:

```
Prelude> reverse "level"
"level"
```

Hold on there! We bumped into a *palindrome*: a word that is read the same way, no matter whether it is read forward or backward. Now, suppose we would like to write our own function to determine whether a word is a palindrome. We first need to make a slightly more formal definition of a palindrome: a word is a palindrome if it is equal to its reverse. In Haskell we can compare values using the == operator:

```
Prelude> "hello" == "hello"
True
Prelude> "hello" == "olleh"
False
```

Such a comparison evaluates to *True* if both values are equal, or to *False* in case they are not. *True* and *False* are the only values of the *Bool* type. Since `reverse` also returns a value, nothing holds us from using it in comparisons:

```
Prelude> "hello" == reverse "hello"
False
Prelude> "level" == reverse "level"
True
```

The test that we devised for detecting palindromes seems to work. But it is a lot of typing. Luckily, we can generalize this into a function. Let's replace both words by the symbolic name *word* (but don't execute this in **ghci** yet, since it does not know this symbolic name):

```
word == reverse word
```

And as a next step, Let's do some magic:

```
Prelude> let palindrome word = word == reverse word
```

This defines the function `palindrome` taking one argument, and binds this argument to the symbolic name *word*. To this function we assign the expression *word == reverse word*. Play a little with this function to be convinced that it actually works. Some examples:

```
Prelude> palindrome "hello"
False
Prelude> palindrome "level"
True
Prelude> palindrome "racecar"
True
```

If this function is still a mystery to you, it may be useful to write down the application of the function stepwise for a word that is not a palindrome:

```
palindrome "hello"
palindrome "hello" = "hello" == reverse "hello"
palindrome "hello" = "hello" == "olleh"
palindrome "hello" = False
```

and a word that *is* a palindrome:

```
palindrome "racecar"
palindrome "racecar" = "racecar" == reverse "racecar"
palindrome "racecar" = "racecar" == "racecar"
palindrome "racecar" = True
```

Congratulations, you have made your first function, which is in essence a small program!

From words to sentences

So far, we have looked at words in isolation. However, in language, words are often combined to form a higher level of meaning representation: a sentence. Provided what we have learned about representing words in Haskell, the step towards representing sentences should be a minor one. We could, for example, represent sentences in the exactly the same way we represented words:

```
Prelude> "The cat is on the mat."
"The cat is on the mat."
```

That's fine for a beginning, although not so convenient. Let us see why. Assume we ask you to give us the first word of a sentence. In the previous section, we learned that `head` can be used to get the first element of a list. Let's try to apply that here:

```
Prelude> head "The cat is on the mat."
'T'
```

As you probably expected, that didn't work. We represented a sentence as a list of characters (a string), and hence asking for the first element will give the first character. But wait! What if we represented a sentence as a list of words?

```
Prelude> ["The", "cat", "is", "on", "the", "mat", "."]
["The","cat","is","on","the","mat","."]
Prelude> :type ["The", "cat", "is", "on", "the", "mat", "."]
["The", "cat", "is", "on", "the", "mat", "."] :: [[Char]]
```

Nifty! We just constructed a list, of a list of characters. Though, you may wonder why we made the punctuation at the end of the sentence a separate "word". Well, this is mostly a pragmatic choice, because gluing this punctuation sign to *mat* does not really form a word either. Having the period sign separate is more practical for future processing. Hence, formally we say that a sentence consists of tokens, where a token can be a word, a number, and a punctuation sign.

Rinse and repeat:

```
Prelude> head ["The", "cat", "is", "on", "the", "mat", "."]
"The"
```

Since a word is also a list, we can apply a function to words as well. For example, we can get the first character of the first word by applying `head`, to the `head` of a sentence:

```
Prelude> head (head ["The", "cat", "is", "on", "the", "mat", "."])
'T'
```

Note that we need parenthesis to force Haskell to evaluate the part in parentheses first. If we do not enforce this order of evaluation, Haskell will try to evaluate *head head* first, which makes no sense. Remember that `head` requires a list as its argument, and `head` is not a list.

Now that we know how to represent sentence, this is a good time to try to write yet another small program. This time, we will write a function to compute the average token length in a corpus (a collection of texts). Since we did not look at real corpora yet, pick any sentence you like as *My Little Corpus*™. The authors will use *"Oh, no, flying pink ponies!"* The average token length is the sum of the lengths of all tokens, divided by the total number of tokens in the corpus. So, stepwise, we have to:

1. Get the length of each token in the corpus.
2. Sum the lengths of the tokens.

3. Divide the sum by the length of the corpus.

You know how to get the in characters length of a single token:

```
Prelude> length "flying"
6
```

Since you are lazy, you are not going to apply `length` to every token in the corpus manually. Instead we want tell Haskell "Hey Haskell! Please apply this length function to each element of the list." It turns out that Haskell has a function to do this which is called `map`. Time to inspect `map`:

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
```

And we are in for another surprise. The most surprising element is probably the first element in the type signature, *(a -> b)*. Also surprising is that we now see three types, *(a -> b)*, *[a]* and *[b]*. The latter is simple: this function takes two arguments, *(a -> b)* and *[a]*, and returns *[b]*. *(a -> b)* as the notation suggests, is a function taking an *a* and returning a *b*. So, `map` is actually a function that takes a function as its argument, or in functional programming-speak: a *higher order* function.

So, `map` is a function that takes a function that maps from *a* to *b*, takes a list of *a*s, and returns a list of *b*s. That looks a suspicious lot like what we want! We have a list of tokens represented as strings, the function length that takes a list and returns its length as an integer, and we want to have a list of integers representing the lengths. Looks like we have a winner!

```
Prelude> map length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies","!"]
[2,1,2,1,6,1,4,6,1]
```

We have now completed our first step: we have the length of each token in the corpus. Next, we have to sum the lengths that we have just retrieved. Fortunately, Haskell has a `sum` function:

```
Prelude> :type sum
sum :: (Num a) => [a] -> a
```

This function takes a list of *a*s, and returns an *a*. But where did the *(Num a)* => come from? Well, *Num* is a so-called *typeclass*. A type can belong to one or more of such typeclasses. But belonging to a typeclass does not come without cost. In fact, it requires that certain functions need to be defined for types that belong to it. For instance, the typeclass *Num* is a typeclass for numbers, which requires amongst others, functions that define addition or subtraction. Coming back to the type signature, `sum` will sum a list of *a*s, but not just any *a*s, only those that belong to the typeclass *Num*. And after all, this makes sense, doesn't it? We cannot sum strings or planets, but we can sum numbers. In fact, we can only sum numbers.

After this solemn introduction into typeclasses, feel free to take a cup of tea (or coffee), and try step two:

```
Prelude> :{
sum (map length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"])
:}
24
```

By now, you will probably smell victory. The only step that remains is to divide the sum by the length of the sentence using the division operator (*/*):

```
Prelude> :{
sum (map length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"]) /
  length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"]
:}

<interactive>:1:0:
    No instance for (Fractional Int)
      arising from a use of `/' at <interactive>:1:0-136
    Possible fix: add an instance declaration for (Fractional Int)
    In the expression:
          sum (map length ["Oh", ",", "no", ",", ....])
        / length ["Oh", ",", "no", ",", ....]
    In the definition of `it':
        it = sum (map length ["Oh", ",", "no", ....])
           / length ["Oh", ",", "no", ....]
```

And we have... Failure! I hope you poured yourself a cup of herb tea! (again alternatively: espresso!) While this is all a bit cryptic, the second line (*No instance for (Fractional Int)*) gives some idea where the trouble stems from. *Fractional* is typeclass for fractional numbers, and Haskell complains that Int is not defined to be of the typeclass *Fractional*. This sounds obvious, since an integer is not a fractional number. In other words, Haskell is trying to tell us that there is an *Int* in some place where it expected a type belonging to the typeclass *Fractional*. Since the division is the only new component, it is the first suspect of the breakdown:

```
Prelude> :type (/)
(/) :: (Fractional a) => a -> a -> a
```

First off, notice that we have put the division operator in parentheses. We have done this because the division operator is used as a so-called *infix function*: it is a function that is put between its arguments (like *1.0 / 2.0*). By putting an infix operator in parentheses, you are stating that you would like to use it as a regular function. This means you can do things like this:

```
Prelude> (/) 1.0 2.0
0.5
```

Anyway, the verdict of the type signature of `(/)` is clear, it requires two arguments that belong to the *Fractional* typeclass. The sum and length that we calculated clearly do not belong to this typeclass, since they are of the type *Int*:

```
Prelude> :{
:type
  sum (map length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"])
:}
sum (map length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"])
  :: Int
Prelude> :{
:type
  length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"]
:}
length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"]
  :: Int
```

Fortunately, Haskell provides the function `fromIntegral` that converts an integer to any kind of number. Add `fromIntegral`, and you surely do get the average token length of the corpus:

```
Prelude> :{
fromIntegral
  (sum (map length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"])) /
  fromIntegral (length ["Oh", ",", "no", ",", "flying", ",", "pink", "ponies", "!"])
:}
2.6666666666666665
```

Well, that was a bumpier ride than you might have expected. Don't worry! During our first forays into Haskell, we were convinced that we were too stupid for this too (and here we are writing a book). However, after more practice, you will learn that Haskell is actually a very simple and logical language.

Maybe it will feel more like a victory after generalizing this to a function. You can follow the same pattern as in the palindrome example: replace the sentence with a symbolic name and transform it into a function:

```
Prelude> :{
let averageLength l =
  fromIntegral (sum (map length l)) / fromIntegral (length l)
:}
Prelude> :{
averageLength ["Oh", ",", "no", ",", "flying", ",", "pink" ,"ponies", "!"]
:}
2.6666666666666665
```

Congratulations, you just wrote your second function! But wait, you actually accomplished more than you may expect. Check the type signature of `averageLength`.

```
Prelude> :type averageLength
averageLength :: (Fractional b) => [[a]] -> b
```

You made your first weird type signature. Show it off to your colleagues, significant other, or dog. `averageLength` is a function that takes a list of a list of *a*, and returns a *b* that belongs to the *Fractional* typeclass. But wait, *a* can be anything, right? What happens if we apply this function to a list of sentences?

```
Prelude> averageLength [["I", "like", "Haskell", "."],
  ["Ruby", "rocks", "too", "."],
  ["Who", "needs", "Java", "?"]]
4.0
```

Woo! That's the average sentence length, expressed in number of words. It turns out that, although we set out to make a function to calculate the average token length, we wrote a function that calculates the average length of lists in a list (e.g., characters in words, words in sentences, or sentences in a text). This happens very often when you write Haskell programs: lots of functions are generic and can be reused for other tasks.

A note on tokenization

When dealing with real-world text, it is usually not neatly split in sentences and tokens. For example, consider this book - punctuation is usually glued to words. These processes, sentence splitting and tokenization may seem trivial, unfortunately they are not. Consider the following sentence:

E.g. Jack doesn't have 19.99 to spend.

If we simply perform sentence splitting on periods (.), we will find four sentences:

1. *E.*

2. *g.*

3. *Jack doesn't have 19.*

4. *99 to spend.*

Of course, it is just one sentence. Similar problems arise during punctuation: how do we know that *E.g.* and *19.99* should not be split? And how about *doesn't*, which should probably be split as *does n't* or *does not*? Tokenization can be performed accurately, but it requires techniques that you will see in later chapters. So don't worry, we will get back to proper tokenization later on. We promise!

Of course, up to the point where we handle tokenization, we need material to work on. To make life easier for you, the material for the first chapters of the book is pre-tokenized in a plain-text file using two simple rules:

1. One sentence per line.

2. Tokens are separated by a space.

To convert a text file to a Haskell representation, sentence splitting is a matter of splitting by line, and tokenization a matter of splitting by space. Have a look at the following example:

```
Prelude> "This is Jack .\nHe is a Haskeller ."
"This is Jack .\nHe is a Haskeller ."
```

This is exactly the representation that we will be using for our textual data. As you can see, the tokens are separated by spaces. Both sentences are separated using a newline. When writing down a string literally, you can insert a newline using *\n*.

Haskell provides a `lines` function to split up a string by line. Not surprisingly, this function accepts a string as its first argument, and will return a list of strings:

```
Prelude> :type lines
lines :: String -> [String]
Prelude> lines "This is Jack .\nHe is a Haskeller ."
["This is Jack .","He is a Haskeller ."]
```

That was easy! Now to the actual tokenization. For all sentences, we have a string representing the sentence. We want to split this string on the space character. Haskell also has a function to do this, named `words`. `words` is nearly the same function as `lines`, except that it splits on spaces rather than newlines:

```
Prelude> words "This is Jack ."
["This","is","Jack","."]
```

That will do, but we have to apply this to every sentence in the list of sentences. Recall that we can use the `map` function we have seen earlier to apply the `words` function to each element of the list of (untokenized) sentences:

```
Prelude> map words (lines "This is Jack .\nHe is a Haskeller .")
[["This","is","Jack","."],["He","is","a","Haskeller","."]]
```

Allright! That will do the job. We know how to turn this into a full-fledged function:

```
Prelude> let splitTokenize text = map words (lines text)
Prelude> splitTokenize "This is Jack .\nHe is a Haskeller ."
[["This","is","Jack","."],["He","is","a","Haskeller","."]]
```

This is a good moment to beautify this function a bit. To make it simpler, we first need to get rid of the parentheses. We used the parentheses to tell Haskell that it should evaluate *lines text* first, because it would otherwise try to map over the function `lines`, which would fail, because it is not a list. Very often, you will encounter function applications of the form *f(g(x))*, or *f(g(h(x)))*, etc. Haskell provides the *(.)* function to combine such function applications. So, *f(g(x))* can be rewritten to *(f . g) x* (apply function *f* to the outcome of *g(x)*) and *f(g(h(x)))* as *(f . g . h) x* (apply function *f* to the outcome of a function *g*, which is in turn applied to the outcome of *h(x))*. As you can see, this so-called *function composition* makes things much easier to read. We can now rewrite our tokenization function by using function composition:

```
Prelude> let splitTokenize text = (map words . lines) text
```

This states that we apply *map words* to the outcome *lines text*. This may not yet seem so interesting. However, it allows us to make yet another simplification step. Consider the type of the `map` function:

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
```

`map` takes a function, and a list, and returns a list. Now we will do something that may look weird, but is very common in functional programming.

```
Prelude> :type map words
map words :: [String] -> [[String]]
```

Applying `map` to just one argument will give... another function! In fact, what we just did is bind only the first argument of the map function, leaving the second unbound. This gives a novel map function that only takes one argument, as it has its first argument implicitly bound to the function words. This process is called *currying* (indeed, named after the mathematician Haskell Curry) in functional programming slang.

If we look back at our `splitTokenize` function, and look up the type of *map words . lines*, we see that it is a function that takes a *String* and returns a list of a list of strings:

```
Prelude> :type map words . lines
map words . lines :: String -> [[String]]
```

In our function body, we apply this function to the argument *text*. Of course, this is not really necessary, because *map words . lines* already defines our function (as we have shown above). We just need to bind this to the name `splitTokenize`. Consequently the function can once more be simplified:

```
Prelude> let splitTokenize = map words . lines
splitTokenize :: String -> [[String]]
Prelude> splitTokenize "This is Jack .\nHe is a Haskeller ."
[["This","is","Jack","."],["He","is","a","Haskeller","."]]
```

Word lists

In the following two sections, we will introduce two prototypical tasks related to words. The first is to make a word (or actually token) list, the second task is making a word frequency list.

A word list is a very simple data structure: it is just a list of *unique* words or tokens that occur in a text. Our corpus is also just a list of words, but since it contains duplicates, it is not a word list. The obvious method to make a word list is to go through a corpus word by word, and adding words that we have not yet seen to a second list. This requires some functions we haven't seen yet:

- Adding an element to a list.
- Checking whether an element is (or is not) in a list.
- Constructing a list while traversing another list.

We like easy things first, so let's start with the first item: adding an element to a list. We have seen the `head` function before that chops of the head of the list and returns it. But we can also do the reverse: take a list and give it a new head. The old head then becomes the head of the tail (are you still following?). In Haskell, we can do this using the `(:)` function:

```
Prelude> 2 : [3,4,5]
[2,3,4,5]
```

Ain't that great? We can also add a head, and yet another:

```
Prelude> 1 : 2 : [3,4,5]
[1,2,3,4,5]
```

What if we do not have an element yet? Add the head to the empty list (*[]*):

```
Prelude> "Hi" : []
["Hi"]
```

With that covered, the next thing we need to be able to do is checking whether some element belongs to a list. We can do this using the `elem` function. It takes an element as its first argument, and a list as its second. It will return a Bool of the value *True* if the element was in the list, or *False* otherwise. For example:

```
Prelude> elem 2 [1,2,3,4,5]
True
Prelude> elem 6 [1,2,3,4,5]
```

```
False
```

The function `notElem` is exactly the inverse of `elem`, and returns *True* if an element is not in the list, and *False* otherwise:

```
Prelude> notElem "foo" ["foo","bar","baz"]
False
Prelude> notElem "pony" ["foo","bar","baz"]
True
```

Ok, so we want to add an element to a list if, but only if, it is true that it is not yet a member of that list. Or in other words, the addition is conditional. Haskell provides a set of keywords to model conditionals, if..then..else. The structure is like this:

```
if expr then a else b
```

This whole structure itself is an expression. This expression evaluates to *a* if *expr* evaluates to *True* or to *b* if *expr* evaluates to False. To give a working, but useless example:

```
Prelude> if 1 == 2 then "cuckoo" else "egg"
"egg"
Prelude> if 1 == 1 then "cuckoo" else "egg"
"cuckoo"
```

This looks exactly like what we need. Just fill in the blanks:

```
Prelude> if elem "foo" ["foo","bar","baz"] then ["foo","bar","baz"]
  else "foo" : ["foo", "bar", "baz"]
["foo","bar","baz"]
Prelude> if elem "pony" ["foo","bar","baz"] then ["foo","bar","baz"]
  else "pony" : ["foo", "bar", "baz"]
["pony","foo","bar","baz"]
```

That's a bit contrived, but (as you hopefully see) not if we rewrite it to a function:

```
Prelude> let elemOrAdd e l = if elem e l then l else e:l
Prelude> elemOrAdd "foo" ["foo", "bar", "baz"]
["foo","bar","baz"]
Prelude> elemOrAdd "pony" ["foo", "bar", "baz"]
["pony","foo","bar","baz"]
```

Good, now we need to apply this to all words in a text, starting with an empty list. Haskell provides a function to do this, but brace yourself, the first time it may look a bit 'difficult'. It is named `foldl` (a so-called) left fold. A left fold traverses a list from head to tail, applying a function to each element, just like `map`. However, the difference is that it can, but does not necessarily return a list. As such, it is a generalization of the `map` function. As usual, you can inspect the type signature to see the arguments of `foldl`:

```
Prelude> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Now consider this example using `foldl`:

```
Prelude> foldl (+) 0 [1,2,3,4,5]
15
```

Stepwise, this fold is executed in the following manner:

```
foldl (+) 0 [1,2,3,4,5]
foldl (+) ((0)+1) [2,3,4,5]
foldl (+) (((0)+1)+2) [3,4,5]
foldl (+) ((((0)+1)+2)+3) [4,5]
foldl (+) (((((0)+1)+2)+3)+4) [5]
foldl (+) ((((((0)+1)+2)+3)+4)+5)) []
((((((0)+1)+2)+3)+4)+5))
```

So, it works by applying a function to some initial argument (*0* in this case) as its first argument, and the first element of the list as its second argument. When processing the second element of the list, this expression is then the first argument of the function, and the second element is the second argument, etc. The first argument of the function that is applied is also called the *accumulator*, since it accumulates results up till that point.

This could also work for our `elemOrAdd` function. Unfortunately, `elemOrAdd` requires the accumulator as the second argument, and the function passed to `foldl` as the first argument. Compare the type signatures:

```
Prelude> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
Prelude> :type elemOrAdd
elemOrAdd :: (Eq a) => a -> [a] -> [a]
```

In the function that is the first argument to `foldl`, the return type is the same as the type of the first argument. In the case of `elemOrAdd`, the type of the second argument corresponds to that of the first. Of course, an easy 'hack' to solve this, is to redefine elemOrAdd, switching its arguments, and plug it into foldl:

```
Prelude> let elemOrAdd l e = if elem e l then l else e:l
```

Now, since we are building a list, we use the empty list (*[]*) as the initial accumulator for this fold:

```
Prelude> foldl elemOrAdd [] ["blue", "blue", "red", "blue", "red"]
["red","blue"]
```

That looks good! Stepwise, the fold works like this:

```
foldl elemOrAdd [] ["blue", "blue", "red", "blue", "red"]
foldl elemOrAdd ("blue":([])) ["blue", "blue", "red", "blue", "red"]
foldl elemOrAdd ("blue":([])) ["blue", "red", "blue", "red"]
foldl elemOrAdd ("blue":([])) ["red", "blue", "red"]
foldl elemOrAdd ("red":("blue":([]))) ["blue", "red"]
foldl elemOrAdd ("red":("blue":([]))) ["red"]
foldl elemOrAdd ("red":("blue":([]))) []
("red":("blue":([])))
["red","blue"]
```

Now we wrap it up in another function, and you have constructed two functions that, together, make word lists:

```
Prelude> let wordList = foldl elemOrAdd []
Prelude> wordList ["blue", "blue", "red", "blue", "red"]
["red","blue"]
```

While our little word list function works fine on small texts, it will not be very efficient for big corpora. The reason is simple - suppose that we have already found 100,000 different tokens. For every word, it would have to check the list of 100,000 tokens. There is no other way of doing this than to traverse the list word by word. Or, on average, we compare a token to 100,000 / 2 = 50,000 elements in the list. As a computer scientist would say: `elemOrAdd` works in linear time, its processing time is linear to the number of different tokens that were seen.

This is a typical case of picking the wrong data structure for the task. But for illustrative purposes, using lists was nice and simple. However, since you are a working programmer, you want workable solutions. Bring in the sets! A set is, like the mathematical set, a collection that does not contain duplicate elements. That's good, because a word list does not contain duplicate elements. Silly us, we were fooled by the word *list*. What we actually want to build is a word set. In fact, it is just for historical purposes that it is called a word list.

Beside the uniqueness of elements, another nice property of sets, as they are normally implemented, is that set membership can be checked rather quickly. In the sets that we will use, membership checking is in logarithmic time. Or in other words, if comparison took one second, we would on average need 50,000 seconds to search the list mentioned earlier, but only *log(100,000)* or approximately 11.5 seconds to check whether the element is in a set. Talking about optimizations!

Haskell provides set functionality, but not in the so-called *Prelude*. *Prelude* is a module that contains functions. The *Prelude* module is always loaded, so its functions are always available (unless you explicitly ask Haskell to hide them). The functions `map`, `head`, `tail`, and `length`, for instance, are defined in the *Prelude*. Functions for set manipulation, on the other hand, are defined in a module named *Data.Set*. For the time being, we will access functions from modules by prefixing the name of the module. For instance, this will give us the empty set:

```
Prelude> Data.Set.empty
fromList []
```

Like a list, a set can contain elements of various types. We see this when inspecting the type signature of the `empty` function:

```
Prelude> :type Data.Set.empty
Data.Set.empty :: Data.Set.Set a
```

`empty` returns a Set of some type *a*. We can also construct a *Set* from a list using the `fromList` function:

```
Prelude> Data.Set.fromList [5,2,5,8,1,1,23]
fromList [1,2,5,8,23]
```

As you can see here, the set does not contain duplicates. Another nice property of Haskell sets is that they are ordered. We can also do the inverse, convert a set to a list using `toList`:

```
Prelude> Data.Set.toList (Data.Set.fromList [5,2,5,8,1,1,23])
[1,2,5,8,23]
```

Elements can be added to or removed from a *Set* using respectively the `insert` and `delete` functions. Both functions return a set with that element inserted or removed:

```
Prelude> Data.Set.insert 42 (Data.Set.fromList [5,2,5,8,1,1,23])
fromList [1,2,5,8,23,42]
Prelude> Data.Set.delete 5 (Data.Set.fromList [5,2,5,8,1,1,23])
fromList [1,2,8,23]
```

Finally, we can check whether some value is a member of a set by using the `member` function:

```
Prelude> Data.Set.member 23 (Data.Set.fromList [5,2,5,8,1,1,23])
True
Prelude> Data.Set.member 24 (Data.Set.fromList [5,2,5,8,1,1,23])
False
```

We have now seen enough to change our word list function. Rather than checking whether a value is in a list and adding it if not, we check whether it is in a *Set* and add it in when it is not:

```
Prelude> :{
let elemOrAdd s e =
  if Data.Set.member e s then s else Data.Set.insert e s
:}
Prelude> elemOrAdd (Data.Set.fromList [5,2,5,8,1,1,23]) 24
fromList [1,2,5,8,23,24]
```

That was simple. But it feels a weird, right? The most vital characteristic of a set is that it never contains duplicate elements, why do we need to check for duplicates? We don't. So, forget about `elemOrAdd`, we will only use *Data.Set.insert* from this point. Our objective now is to traverse a list of tokens, adding each token to a set, starting with the empty set. Our first take is this:

```
Prelude> let wordSet = foldl Data.Set.insert Data.Set.empty
```

However, this will not work. Remember that in the function we give to `foldl`, the accumulator has to be the first argument (the reason why we swapped the elements of our initial *elemOrAdd* function)? We are accumulating a *Set*, but the set is the second argument to *Data.Set.insert*. We will pull a little trick out of our hat.

```
Prelude> let wordSet = foldl (\s e -> Data.Set.insert e s) Data.Set.empty
```

You might be thinking "Oh, no, more syntax terror! Does it ever stop?" Actually, *(e -> Data.Set.insert e s)* is very familiar. You could see it as an inline function. In functional programming jargon, this is called a *lambda*. Check out the type signature of the lambda:

```
Prelude> :type (\s e -> Data.Set.insert e s)
(\s e -> Data.Set.insert e s)
  :: (Ord a) => Data.Set.Set a -> a -> Data.Set.Set a
```

It is just a function, it takes a set of some type *a*, a value of type *a*, and returns *a*. Additionally, a has the typeclass Ord, which means that some comparison operators should be defined for a. The lambda has two arguments that are bound to *s* and *e*. The function body comes after the arrow. To emphasize that this is just a function, the following functions are equivalent:

```
myFun = (\s e -> Data.Set.insert e s)
myFun s e = Data.Set.insert e ss
```

Back to our `wordSet` function. We used the lambda to swap the arguments of *Data.Set.insert*. *Data.Set.insert* takes a value and a set, our lambda takes a set and a value. The rest of the function follows the same pattern as *wordList*, except that we start with an empty set rather than an empty list. The function works as expected:

```
Prelude> wordSet ["blue", "blue", "red", "blue", "red"]
fromList ["blue","red"]
```

You have done it! You are now not only able to make a function that creates a word list, but also one that is performant.

Exercises

1. To measure the vocabulary of a writer, a so-called type-token ratio can be calculated. This is the number of distinct tokens occurring in a text (*types*) divided by the total number of tokens in that text.

   Equation 2.1. Type-token ratio

   For instance the phrase "to be or not to be" contains six tokens and four types (*to*, *be*, *or*, *not*). The type-token ratio of this phrase is $4 / 6 = 2 / 3$.

   Write a function that calculates the type-token ratio of a list of tokens. You can use the *Data.Set.size* function to get the number of elements in a set.

Storing functions in a file

Now that we are writing longer and longer functions, it becomes more convenient to define functions in a file rather than the **ghci** prompt. You can do this by creating a file using a plain-text editor with the *.hs* extension. Functions can be written down in the same manner as in *ghci*, but without the preceding *let* keyword. It is also highly recommended to add a type signature before the function. Haskell will check the function against the type signature, and report an error if they do not correspond. This will help you catch incorrect function definitions.

The `palindrome` function discussed earlier in this chapter can be written to a file like this:

```
palindrome :: (Eq a) => [a] -> Bool
palindrome word = word == reverse word
```

If you saved this file as *example.hs*, you can load it in **ghci** using the *:l* (shorthand for *:load*) command:

```
Prelude> :l example
[1 of 1] Compiling Main             ( example.hs, interpreted )
Ok, modules loaded: Main.
*Main> palindrome "racecar"
True
```

For code fragments that use a module other than the prelude, add an import statement at the top of the file. For example, the `wordSet` function from the previous section should be saved to a text file in the following manner:

```
import qualified Data.Set

wordSet :: Ord a => [a] -> Data.Set.Set a
wordSet = foldl (\s e -> Data.Set.insert e s) Data.Set.empty
```

From now on, we assume that examples are written to a text file, except when the *Prelude>* occurs in the example.

Word frequency lists

The word list function that we built in the previous section works is useful for various tasks, like calculating the type-token ratio for a text. For some other tasks this is not good enough - we want to be able to find out how often a word was used. We can expand a word list with frequencies to make a *word frequency list*.

To be able to store word frequencies, every word has to be associated with an integer. We could store such an association as a tuple. A tuple is a data type with a fixed number of elements and a fixed type for an element. Examples of tuples are:

- (1,2,3)
- ("hello","world")
- ("hello",1)

As you can see, they differ from lists in that they can have values of different types as elements. However, if you inspect the type signatures of these tuples, you will see that the length and type for each position is fixed:

```
Prelude> :type (1,2,3)
(1,2,3) :: (Num t, Num t1, Num t2) => (t, t1, t2)
Prelude> :type ("hello","world")
("hello","world") :: ([Char], [Char])
Prelude> :type ("hello",1)
("hello",1) :: (Num t) => ([Char], t)
```

To store frequencies, we could use a list of tuples of the type *[([Char], Int)]*. The phrase "to be or not to be" could be stored as

```
[("to",2),("be",2),("or",1),("not",1)]
```

However, this would be even less efficient than using lists for constructing word lists. First, like `elemOrAdd` we would potentially have to search the complete list to locate a word. Second, we would have to reconstruct the list up to the point of the element. In the `elemOrAdd` function we could just give the list a new head, but now we would have to replace the element to update the word frequency and add all preceding list items again. Since Haskell is a 'pure' language, we cannot modify existing values.

A more appropriate data type for this task is a map (not to be confused with the `map` function). A map maps a key to a value. In Haskell, maps are provided in the *Data.Map* module. Like sets, we can make an empty map:

```
Prelude> Data.Map.empty
fromList []
```

When you inspect the type signature of the empty map, you can see that it parametrizes over two types, a type for the key and a type for values:

```
Prelude> :type Data.Map.empty
Data.Map.empty :: Data.Map.Map k a
```

We can construct a *Map* from a list of binary tuples (tuples with two elements), where the first element of the tuple becomes the key, and the second the value:

```
Prelude> Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)]
fromList [("be",2),("not",1),("or",1),("to",2)]
Prelude> :type Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)]
Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)]
  :: (Num t) => Data.Map.Map [Char] t
```

This also binds the types for the map: we are mapping from keys of type string to values of type `t` that belongs to the *t* typeclass. No specific value for types is used (yet), because the numbers could be integers or fractionals.

The `insert` function is used to add a new mapping to the *Map*:

```
Prelude> :{
  Data.Map.insert "hello" 1
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
fromList [("be",2),("hello",1),("not",1),("or",1),("to",2)]
```

If a mapping with the given key already exists, the existing mapping is replaced:

```
Prelude> :{
  Data.Map.insert "be" 1
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
fromList [("be",1),("not",1),("or",1),("to",2)]
```

Looking up values is a bit peculiar. You can lookup a value with the `lookup` function. However, if you inspect the type signature, you will see that the value is not returned as is:

```
Prelude> :type Data.Map.lookup
Data.Map.lookup :: (Ord k) => k -> Data.Map.Map k a -> Maybe a
```

Rather than returning a value, it returns the value packed in some box called *Maybe*. *Maybe a* is a type that has just two possible so-called *constructors*, *Just a* or *Nothing*. You can put your own values in a *Maybe* box using the *Just a* constructor:

```
Prelude> Just 22
Just 22
Prelude> :type Just 22
Just 22 :: (Num t) => Maybe t
Prelude> Just [1,2,3,4,5]
Just [1,2,3,4,5]
Prelude> Just "stay calm"
Just "stay calm"
Prelude> :type Just "stay calm"
```

```
Just "stay calm" :: Maybe [Char]
```

You can also make a box that contains vast emptiness with the *Nothing* constructor:

```
Prelude> Nothing
Nothing
Prelude> :type Nothing
Nothing :: Maybe a
```

These boxes turn out to be pretty cool: you can use them to return something or nothing from functions, without resorting to all kinds of abominations as exceptions or null pointers (if you never heard of exceptions or pointers, do not worry, you have a life full of bliss). Since *Maybe* is so nice, the `lookup` function uses it. It will return the value packed with in a *Just* constructor if the key occurred in the map, or *Nothing* otherwise:

```
Prelude> :{
  Data.Map.lookup "to"
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
Just 2
Prelude> :{
  Data.Map.lookup "wrong"
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
Nothing
```

As for handling these values - we will come to that later. Mappings are deleted from a *Map* by key with the `delete` function. If a key did not occur in the *Map*, the original map is returned:

```
Prelude> :{
  Data.Map.delete "to"
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
fromList [("be",2),("not",1),("or",1)]
Prelude> :{
  Data.Map.delete "wrong"
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
fromList [("be",2),("not",1),("or",1),("to",2)]
```

Finally, a *Map* can be converted to a list using the `toList` function:

```
Prelude> :{
  Data.Map.toList
    (Data.Map.fromList [("to",2),("be",2),("or",1),("not",1)])
:}
[("be",2),("not",1),("or",1),("to",2)]
```

Alright. Back to our task at hand: constructing a word frequency list. As with word lists, we want to traverse a list of words, accumulating data. So, the use of `foldl` is appropriate for this task. During each folding step, we take the *Map* created in a previous step. We then lookup the value for the current step in the *Map*. If it does not exist, we add it to the *Map* giving it a frequency of one. Otherwise, we want to increase the frequency by one. The `countElem` function does this:

```
import qualified Data.Map

countElem :: (Ord k) => Data.Map.Map k Int -> k -> Data.Map.Map k Int
countElem m e = case (Data.Map.lookup e m) of
                  Just v  -> Data.Map.insert e (v + 1) m
                  Nothing -> Data.Map.insert e 1 m
```

This function introduces the *case* construct. Remember that `lookup` uses the nifty *Maybe* data type? The *case* construct allows us to select an expression based on a constructor. If `lookup` returned a value using the *Just* constructor, the key was in the *Map*. In this case, we bind the value to the name *v* and add a new value for this key. This value is the old value for this key incremented by one. If a value with the *Nothing* constructor was returned, the key was not in the *Map*. So, we will add it, and give it a (frequency) value of 1.

The `countElem` function works as intended:

```
*Main> foldl countElem Data.Map.empty ["to","be","or","not","to","be"]
fromList [("be",2),("not",1),("or",1),("to",2)]
```

While this was a nice exercise, the *Data.Map.insertWith* function can drastically shorten our function. This function uses an update function to update a value, or a specified value if the key is not present in the *Map*:

```
*Main> :t Data.Map.insertWith
Data.Map.insertWith
  :: Ord k =>
     (a -> a -> a) -> k -> a -> Data.Map.Map k a -> Data.Map.Map k a
```

The update function gets the specified value as its first argument, and the old value as its second argument. Using `insertWith`, we can shorten our function to:

```
countElem :: (Ord k) => Data.Map.Map k Int -> k -> Data.Map.Map k Int
countElem m e = Data.Map.insertWith (\n o -> n + o) e 1 m
```

If an element was not seen in the *Map* yet, a frequency of 1 will be inserted. If the element does occur as a key in the map, the lambda adds one to the old frequency. With `countElem` in our reach, we can define the `freqList` function:

```
freqList :: (Ord k) => [k] -> Data.Map.Map k Int
freqList = foldl countElem Data.Map.empty
```

Monads

In the next section you will see how to read real text corpora using the so-called *IO monad*. Before diving into the *IO monad*, we will give a short introduction to *monads*. In Haskell, it happens very often that you want to perform a series of computations on values that are wrapped using some type, such as Maybe or a list. For instance, suppose that you have a Map that maps a customer name to a customer number, and yet another Map that maps a customer number to a list of order numbers:

```
*Main> let customers = Data.Map.fromList [("Daniel de Kok", 1000),
  ("Harm Brouwer", 1001)]
*Main> let orders = Data.Map.fromList [(1001, [128])]
Prelude> Data.Map.lookup "Harm Brouwer" customers
Just 1001
Prelude> Data.Map.lookup 1001 orders
Just [128]
```

Now, we want to write a function that extracts a list orders for a given customer, wrapped in a Maybe, so that Nothing can be returned if the customer is not in the list or if the customer had no orders. Your first attempt will probably be along the following lines:

```
lookupOrder0 :: Data.Map.Map String Integer ->
                Data.Map.Map Integer [Integer] ->
                String -> Maybe [Integer]
lookupOrder0 customers orders customer =
    case Data.Map.lookup customer customers of
      Nothing -> Nothing
      Just customerId -> Data.Map.lookup customerId orders
```

This function works as intended:

```
*Main> lookupOrder0 customers orders "Jack Sparrow"
Nothing
```

```
*Main> lookupOrder0 customers orders "Daniel de Kok"
Nothing
*Main> lookupOrder0 customers orders "Harm Brouwer"
Just [128]
```

But *case* constructs will quickly start to stack up when more functions are called that return Maybe. For each Maybe we follow the same procedure: if the value is Nothing we end the computation with that value, if the value is Just x we call the next function that possibly uses `x` as an argument. This is where the Monad typeclass kicks in: all data types that are of the Monad typeclass implement the `(>>=)` function. This function *joins* computations resulting in that data type according to some logic. For instance, the Maybe monad combines expressions that return Maybe in such a way that if one expression returns Nothing, the whole joined expression also returns Nothing, just like our *case* construct in the example above. Consequently, we could rewrite the example above as:

```
lookupOrder1 :: Data.Map.Map String Integer ->
                Data.Map.Map Integer [Integer] ->
                String -> Maybe [Integer]
lookupOrder1 customers orders customer =
    Data.Map.lookup customer customers >>= (\m -> Data.Map.lookup m orders)
```

That surely shortened the function! But what does it do? Well, the function performs the following steps:

- `Data.Map.lookup customer customers`: Lookup `customer` in `customers`.
- `(>>=)`: If the expression on the left-hand returned Nothing, the value of the full expression, and consequently `lookupOrder1`, is Nothing. If the lookup returned a value of type Just Integer, extract the Integer from the Just constructor, and pass it as the argument of the next function.
- `(\m -> Data.Map.lookup m orders)`: Lookup the supplied argument in the `orders` Map. The result becomes the result of `lookupOrder1`. We had to use a lambda, to make the element to be looked up the first argument.

The type signature of (>>=) is also very illustrative:

```
*Main> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

The (>>=) simply unwraps a value from the monad and feeds it to a function that returns a value wrapped in the same monad. The `(>>=)` function has the freedom to perform any operation (including not calling (a -> mb)), as long as it returns a value of type m b.

The actual implementation of (>>=) for the Maybe monad is very simple:

```
(Just x) >>= k = k x
Nothing  >>= _ = Nothing
```

That's all! If the left-hand side expression evaluates to Just x, the expression on the left hand side is evaluated with `x` as its argument. If the left-hand side evaluates to Nothing, the right-end side is not evaluated, and the whole expression returns Nothing.

There is yet another function that is essential to monads named return, it does nothing else than wrapping a value in that monad. For instance, the `maybeBool` function wraps a Bool value in a Maybe monad:

```
maybeBool :: Bool -> Maybe Bool
maybeBool = return
```

This is `maybeBool` in action:

```
*Main> maybeBool True
Just True
*Main> maybeBool False
Just False
```

The implementation of `return` for the Maybe monad is trivial:

```
return = Just
```

Let's get back to our order lookup function. You may have noticed that the `(>>=)` function is somewhat imperative in nature: it chains a set of expressions, where the left-hand expression is evaluated before the right-hand. Due to this nature, Haskell has a *do notation* that resembles imperative programs. This is `lookupOrder1` using the do-notation:

```
lookupOrder2 :: Data.Map.Map String Integer ->
                Data.Map.Map Integer [Integer] ->
                String -> Maybe [Integer]
lookupOrder2 customers orders customer = do
    customerId <- Data.Map.lookup customer customers
    orders     <- Data.Map.lookup customerId orders
    return orders
```

You can see that we added the *do* keyword to start the do-notation. Every expression that follows can be seen as just one element in a sequence of (>>=) expressions. Consequently, each line is governed by the 'laws' of the monad. If the first lookup fails, no further evaluation is performed, and `lookupOrder2` will return Nothing. Otherwise, computation continues. The do-notation also allows the use of the backward arrow (<-) this arrow extracts a value from the monad, and binds it to a variable.

We can simplify lookupOrder2 further. The last lookup already returns a value wrapped in the Maybe monad, there is no need to extract it using <- and wrap it again with `return`:

```
lookupOrder :: Data.Map.Map String Integer ->
               Data.Map.Map Integer [Integer] ->
               String -> Maybe [Integer]
lookupOrder customers orders customer = do
    customerId <- Data.Map.lookup customer customers
    Data.Map.lookup customerId orders
```

Reading a text corpus

Up to this point we have been using very artificial text corpora. At most a few sentences. But you are in it for the real deal, right? Lucky you, we will use a real (like really real) corpus starting from this very moment. Of course, just to test functions we will start with small examples. But you will be able to apply your functions to real data.

So-called *I/O* (input/output) is a delicate matter in Haskell. The reason being that, as a pure functional language, it is not possible to modify expressions or values once they exist. This leads to an admirable quality of Haskell: given the same input, a function will always return the same output. Or in other words, a function does not have *side-effects*. Unlike most other languages, there is no state in a function that can change, so the output can also not change. If a function always evaluates to the same value given the same input, how can you have I/O? For example, suppose that we open a file, and use a function read a byte. And then we read yet another byte. The second byte may be a different one. The reading function has a side-effect: it increases the position within the file.

The Haskell developers have, clever as they are, found a solution to get Pandora's box into Haskell. I/O in Haskell is performed in the so-called *IO* monad. The IO monad is not very different from the Maybe monad, it implements the `(>>=)` and return functions as required for monads. However, there is a subtle, but very important difference. Remember that you can extract a value from a Maybe value using its Just constructor?

```
value = case someMaybe of
  Just x = x
```

The IO monad does not have a public constructor, so there is no way to pry a value out of an IO monad. If you read a file as a list of *Char*, it resides in the *IO* monad. You can apply any list function to this list, however, the result of the function that is applied to a list will also have to reside in the IO monad.

A value can never escape the IO monad. And this is how impure I/O is possible in Haskell without sacrificing purity of the language: impure data stays in the IO monad, and can never escape.

Now on to some real work. As said, functions that do IO return a value wrapped in IO. For instance, the `putStrLn` function returns an empty tuple packed in the *IO* monad:

```
Prelude> :type putStrLn "hello world!"
putStrLn "hello world!" :: IO ()
```

This is just a normal value, you can bind it to a name in **ghci**:

```
Prelude> let v = putStrLn "hello world!"
v :: IO ()
```

However, if we evaluate the value in **ghci**, it will execute this *I/O action*:

```
Prelude> v
hello world!
```

Of course, the same thing happens if we evaluate `putStrLn` directly:

```
Prelude> putStrLn "hello world!"
hello world!
```

Now, let's get to the interesting part: reading a file. In the files distributed with this book, you will find the file `brown.txt`. This file contains the Brown corpus, a corpus of written text of various kinds. The Brown corpus is already tokenized, which makes our life a bit easier. Ok, first we need to open the file using the `IO.openFile` function. `openFile` requires a filename and an I/O mode as its arguments, and it returns a handle packed in the *IO*monad:

```
Prelude> :type IO.openFile
IO.openFile
  :: FilePath
     -> GHC.IO.IOMode.IOMode
     -> IO GHC.IO.Handle.Types.Handle
```

We use *IO.ReadMode* to open `brown.txt` for reading:

```
Prelude> let h = IO.openFile "brown.txt" IO.ReadMode
Prelude> :type h
h :: IO GHC.IO.Handle.Types.Handle
```

The handle is bound to *h*, but still in the *IO* monad. It would be nicer if we can access that handle directly, but we told you that a value can never escape its monad. Surprisingly, we can extract the value from the *IO* monad in **ghci**. The reason that you can is that **ghci** lives in the *IO* monad itself. So, the value will still never leave *IO*. We can bind the value to a name (or pattern) using <-:

```
Prelude> h <- IO.openFile "brown.txt" IO.ReadMode
Prelude> :type h
h :: GHC.IO.Handle.Types.Handle
```

That gives us the handle, bound to *h*. The next function that we will use is *IO.hGetContents*, which returns unread data as a *String* wrapped in the *IO* monad:

```
Prelude> :type IO.hGetContents
IO.hGetContents :: GHC.IO.Handle.Types.Handle -> IO String
```

As we mentioned earlier, Haskell is a lazy language: expressions are only evaluated when necessary. The same thing applies to I/O: the relevant contents of a file are only read once you start extracting characters from the *String*. With some smart programming, it is not necessary for your program to read the whole file into memory, it will allocate and deallocate chunks of the file as they are used. Now, get the contents of the file:

```
Prelude> c <- IO.hGetContents h
Prelude> :type c
c :: String
```

We can apply the usual list functions to this *String*:

```
Prelude> head c
'T'
Prelude> length c
6157180
```

Since the file is sentence-splitted using newlines and tokenized using spaces, you can use the `lines` and `words` functions to apply sentence splitting and tokenization. For instance, the first word of the corpus is:

```
Prelude> head (words (head (lines c)))
"The"
```

The frequency of the word *the* is nicely wrapped in a *Just* constructor:

```
Prelude> Data.Map.lookup "the" (freqList (words c))
Just 62713
```

Congratulations! This was your first venture into the world of corpus statistics!

Todo: Zipfian distribution.

Chapter 3. N-grams

Introduction

In the previous chapter, we have looked at words, and the combination of words into a higher level of meaning representation: a sentence. As you might recall being told by your high school grammar teacher, not every random combination of words forms an grammatically acceptable sentence:

- Colorless green ideas sleep furiously
- Furiously sleep ideas green colorless
- Ideas furiously colorless sleep green

The sentence *Colorless green ideas sleep furiously* (made famous by the linguist Noam Chomsky), for instance, is grammatically perfectly acceptable, but of course entirely nonsensical (unless you ate wrong/weird mushrooms, that is). If you compare this sentence to the other two sentences, this grammaticality becomes evident. The sentence *Furiously sleep ideas green colorless* is grammatically unacceptable, and so is *Ideas furiously colorless sleep green*: these sentences do not play by the rules of the English language. In other words, the fact that languages have rules constraints the way in which words can be combined into an acceptable sentences.

Hey! That sounds good for us NLP programmers (we can almost hear you think), language plays by rules, computers work with rules, well, we're done, aren't we? We'll infer a set of rules, and there! we have ourselves *language model*. A model that describes how a language, say English, works and behaves. Well, not so fast buster! Although we will certainly discuss our share of such rule-based language models later on (in the chapter about parsing), the fact is that nature is simply not so simple. The rules by which a language plays are very complex, and no full set of rules to describe a language has ever been proposed. Bummer, isn't it? Lucky for us, there are simpler ways to obtain a language model, namely by exploiting the observation that words do not combine in a random order. That is, we can learn a lot from a word and its neighbors. Language models that exploit the ordering of words, are called *n-gram language models*, in which the *n* represents any integer greater than zero.

N-gram models can be imagined as placing a small window over a sentence or a text, in which only *n* words are visible at the same time. The simplest n-gram model is therefore a so-called *unigram* model. This is a model in which we only look at one word at a time. The sentence *Colorless green ideas sleep furiously*, for instance, contains five unigrams: “colorless”, “green”, “ideas”, “sleep”, and “furiously”. Of course, this is not very informative, as these are just the words that form the sentence. In fact, N-grams start to become interesting when *n* is two (a *bigram*) or greater. Let us start with bigrams.

Bigrams

An unigram can be thought of as a window placed over a text, such that we only look at one word at a time. In similar fashion, a bigram can be thought of as a window that shows two words at a time. The sentence *Colorless green ideas sleep furiously* contains four bigrams:

- Colorless, green
- green, ideas
- ideas, sleep
- sleep, furiously

To stick to our ‘window’ analogy, we could say that all bigrams of a sentence can be found by placing a window on its first two words, and by moving this window to the right one word at a time in a stepwise manner. We then repeat this procedure, until the window covers the last two words of a sentence. In fact, the same holds for unigrams and N-grams with *n* greater than two. So, say we have a body of text represented as a list of words or tokens (whatever you prefer). For the sake of legacy, we will stick to a list of tokens representing the sentence *Colorless green ideas sleep furiously*:

```
Prelude> ["Colorless", "green", "ideas", "sleep", "furiously"]
["Colorless","green","ideas","sleep","furiously"]
```

Hey! That looks like… indeed, that looks like a list of unigrams! Well, that was convenient. Unfortunately, things do not remain so simple if we move from unigrams to bigrams or *some-very-large-n-grams*. Bigrams and n-grams require us to construct 'windows' that cover more than one word at a time. In case of bigrams, for instance, this means that we would like to obtain a list of lists of two words (bigrams). Represented in such a way, the list of bigrams in the sentence *Colorless green ideas sleep furiously* would look like this:

```
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"]]
```

To arrive at such a list, we could start out with a list of words (yes indeed, the unigrams), and complete the following sequence of steps:

1. Place a window on the first bigram, and add it to our bigram list

2. Move the window one word to the right

3. Repeat from the first step, until the last bigram is stored

Provided these steps, we first need a way to place a window on the first bigram, that is, we need to isolate the first two items of the list of words. In its prelude, Haskell defines a function named *take* that seems to suit our needs:

```
Prelude> :type take
take :: Int -> [a] -> [a]
```

This function takes an *Integer* denoting *n* number of elements, and a list of some type *a*. Given these arguments, it returns the first *n* elements of a list of *a*s. Thus, passing it the number two and a list of words should give us... our first bigram:

```
Prelude> take 2 ["Colorless", "green", "ideas", "sleep", "furiously"]
["Colorless","green"]
```

Great! That worked out nice! Now from here on, the idea is to add this bigram to a list, and to move the window one word to the right, so that we obtain the second bigram. Let us first turn to the latter (as we will get the list part for free later on). How do we move the window one word to the right? That is, how do we extract the second and third word in the list, instead of the first and second? A possible would be to use Haskell's *!!* operator:

```
Prelude> :t (!!)
(!!) :: [a] -> Int -> a
```

This operator takes a list of *a*s, and returns the *n*th element;

```
Prelude> ["Colorless", "green", "ideas", "sleep", "furiously"] !! 1
"green"
Prelude> ["Colorless", "green", "ideas", "sleep", "furiously"] !! 2
"ideas"
```

Great, this gives us the two words that make up the second bigram. Now all we have to do is stuff them together in a list:

```
Prelude> ["Colorless", "green", "ideas", "sleep", "furiously"] !! 1 :
["Colorless", "green", "ideas", "sleep", "furiously"] !! 2 : []
["green","ideas"]
```

Well, this does the trick. However, it is not very convenient to wrap this up in a function, and moreover, this approach is not very Haskellish. In fact, there is a better and more elegant solution, namely to move the list instead of the window. Wait! What? Yes, move the list instead of the window. But how? Well, we could just look at the first and second word in the list again, after getting rid of the (previous) first word. In other words, we could look at the first two words of the tail of the list of words:

```
Prelude> take 2 (tail ["Colorless", "green", "ideas", "sleep", "furiously"])
["green","ideas"]
```

Now that looks Haskellish! What about the next bigram? and the one after that? Well, we could apply the same trick over and over again. We can look at the first two words of the tail of the tails of the list of words:

```
Prelude> take 2 (tail (tail ["Colorless", "green", "ideas", "sleep", "furiously"]))
["ideas","sleep"]
```

... and the tail of the tail of the tail of the list of words:

```
Prelude> take 2 (tail (tail (tail ["Colorless", "green", "ideas", "sleep", "furiously"])))
["sleep","furiously"]
```

In fact, that last step already gives us the last bigrams in the sentence *Colorless green ideas sleep furiously*. The last step would be to throw all these two word lists in a larger list, and we have ourselves a list of bigrams. However, whereas this is manageable by hand for this particular example, think about obtaining all the bigrams in the Brown corpus in this manner (gives you nightmares, doesn't it?). Indeed, we would rather like to wrap this approach up in a function that does all the hard word for us. Provided a list, this function should take its first two arguments, and then repetitively do this for the tail of this, and the tail of the tail of this list, and so forth. In other words, it should simply constantly take the first bigram of a list, and do the same for its tail:

```
bigram :: [a] -> [[a]]
bigram xs = take 2 xs : bigram (tail xs)
```

Wow! That almost looks like black magic, doesn't it? The type signature reveals that the function *bigram* takes a list of *a*s, and returns a list of list of *a*s. The latter could be a list of bigrams, so this looks promising. The function takes the first two elements of the list of *a*s, and places them in front of the result of applying the same function to the tail of the list of *a*s. Eehh.. what? Congratulations! You have just seen your first share of *recursion* magic (or madness). A recursive function is a function that calls itself, and whereas it might look dazzling on first sight, this function actually does nothing more than what we have done by hand in the above. It collects the first two elements of a list, and then does the same for the tail of this list. Moreover, it stuffs the two word lists in a larger list on the fly (we told you the list stuff would come in for free, didn't we?). But wait, will this work? Well, let us put it to a test:

```
Prelude> bigram ["Colorless", "green", "ideas", "sleep", "furiously"]
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"],
["furiously"],[],*** Exception: Prelude.tail: empty list
```

And the answer is... almost! The function gives us the four bigrams, but it seems to be too greedy: it does not stop looking for bigrams after collecting the last bigram in the list of words. But did we tell it when to stop then? Nope, we didn't. In fact, we have only specified a so-called *recursive step* of our recursive function. What we miss is what is called a *stop condition* (also known as a *base case*). In a recursive definition, a stop condition defines when a function should stop calling itself, that is, when our recursive problem is solved. In absence of a stop condition, a recursive function will keep calling itself for eternity. In fact, this explains above the error, we didn't specify a stop condition so the function will keep looking for bigrams for eternity. However, as the list of words is finite, the function will run into trouble when trying to look for bigrams in the tail of an empty list, and this is exactly what the exception tells us. So, how to fix it? Well, add a stop condition that specifies that we should stop looking for bigrams when the tail of a list contains only one item (as it is difficult to construct a bigram out of only one word). We could do this using an if..then..else structure:

```
bigram :: [a] -> [[a]]
bigram xs = if length(xs) >= 2
 then take 2 xs : bigram (tail xs)
 else []
```

This should solve our problems:

```
Prelude> bigram ["Colorless", "green", "ideas", "sleep", "furiously"]
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"]]
```

And, indeed it does. When there is only one word left in our list of words, the bigram function returns an empty list, and moreover, it will stop calling itself therewith ending the recursion. So, lets see how this works with an artificial example. First we will recursively apply the bigram function until it is applied to a list that has less than two elements:

```
Prelude> bigram [1,2,3,4]
bigram [1,2,3,4] = [1,2] : bigram (tail [1,2,3,4])
bigram [2,3,4] = [2,3] : bigram (tail [2,3,4])
bigram [3,4] = [3,4] : bigram (tail [3,4])
bigram [4] = []
```

Application of the bigram function to a list with less than two elements results in an empty list. Moreover, the bigram function will not be applied recursively again as we have reached our stop condition. Now, the only thing that remains is to *unwind* the recursion. That is, we have called the bigram function from within itself for three times, and as we have just found the result to its third and last self call, we can now reversely construct the result of the outermost function call:

```
bigram [3,4] = [3,4] : []
bigram [2,3,4] = [2,3] : [3,4] : []
bigram [1,2,3,4] = [1,2] : [2,3] : [3,4] : []
[[1,2],[2,3],[3,4]]
```

Great! Are you still with us? As L. Peter Deutsch put it: "to iterate is human, to recurse divine." Whereas recursive definitions may seem difficult on first sight, you will find they are very powerful once you get the hang of them. In fact, they are very common in Haskell, and this will certainly be the first of many to come in the course of this book. Lets stick to the bigram function a little longer, because whereas the above works, it is aesthetically unpleasing. That is, we used an if..then..else structure to define our stop condition, but Haskell provides a more elegant way to do this through so-called *pattern matching*. Pattern matching can be thought of as defining multiple definition of the same functions, each tailored and honed for a specific argument pattern. Provided an argument, Haskell will then pick the first matching definition of a function, and return the result its application. Hence, we can define patterns for the stop condition and recursive step as follows:

```
bigram :: [a] -> [[a]]
bigram [x] = []
bigram xs  = take 2 xs : bigram (tail xs)
```

The second line represents the stop condition, and the third the familiar recursive step. Provided the list of words in the sentence *Colorless green ideas sleep furiously*, Haskell will match this to the recursive step, and apply this definition of the function to the list. When the recursive step calls the bigram function with a list that contains only one word (indeed, the tail of the list containing the last bigram), Haskell will match this call with the stop condition. The result of this call will simply an empty list. Lets first proof that this indeed works:

```
Prelude> bigram ["Colorless", "green", "ideas", "sleep", "furiously"]
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"]]
```

It did! To make the working of the use of pattern matching more insightful we can again write out an artificial example in steps:

```
Prelude> bigram [1,2,3,4]
bigram [1,2,3,4] = [1,2] : bigram (tail [1,2,3,4])
bigram [2,3,4] = [2,3] : bigram (tail [2,3,4])
bigram [3,4] = [3,4] : bigram (tail [3,4])
bigram [4] = []
bigram [3,4] = [3,4] : []
bigram [2,3,4] = [2,3] : [3,4] : []
bigram [1,2,3,4] = [1,2] : [2,3] : [3,4] : []
[[1,2],[2,3],[3,4]]
```

Check? We are almost there now. There two things left that we should look at before we mark our function as production ready. The first is a tiny aesthetically unpleasing detail. In the pattern of our step condition we use the variable *x*, whereas we do not use this variable in the body of the function. It is therefore not necessary to bind the list element to this variable. Fortunately, Haskell provides a pattern that matches anything, without doing binding. This pattern is represented by an underscore. Using this underscore, we can patch up the aesthetics of our function:

```
bigram :: [a] -> [[a]]
bigram [_] = []
bigram xs  = take 2 xs : bigram (tail xs)
```

Secondly, our function fails if we apply it to an empty list:

```
Prelude> bigram []
[[]*** Exception: Prelude.tail: empty list
```

But hey! That error message looks familiar, doesn't it? Our function fails, again because we attempted to extract a bigram from the tail of an empty list. Indeed, an empty list does not match with the pattern of our stop condition, and therefore the recursive step is applied to it. We can solve this by adding a pattern for an empty list:

```
bigram :: [a] -> [[a]]
bigram []  = []
bigram [_] = []
bigram xs  = take 2 xs : bigram (tail xs)
```

This new pattern basically states that the list of a bigrams of an empty word list is in turn an empty list. This assures that our function will not fail when applied to an empty list:

```
Prelude> bigram []
[]
```

If you want to get really fancy, you could also use pattern matching to extract a bigram, rather than using `take`:

```
bigram' :: [a] -> [[a]]
bigram' (x:y:xs) = [x,y] : bigram xs
bigram' _        = []
```

Now, we only need to account for two patterns: the first pattern matches when the list has at least two elements. The second pattern matches the empty list and the list containing just one element.

Good, we are all set! We have our bigram function now... time for some applications of a bigram language model!

Exercises

1. A skip-bigram is any pair of words in sentence order. Write a function `skipBigrams` that extracts skip-bigrams from a sentence as a list of binary tuples, using explicit recursion. Running your function on *["Colorless", "green", "ideas", "sleep", "furiously"]* should give the following output:

   ```
   Prelude> skipBigrams ["Colorless", "green", "ideas", "sleep", "furiously"]
   [("Colorless","green"),("Colorless","ideas"),("Colorless","sleep"),
   ("Colorless","furiously"),("green","ideas"),("green","sleep"),
   ("green","furiously"),("ideas","sleep"),("ideas","furiously"),
   ```

```
    ("sleep","furiously")]
```

A few words on Pattern Matching

Stub

Collocations

A straightforward application of bigrams is the identification of so-called *collocations*. Recall that bigram language models exploit the observations that words do not simply combine in any random order, that is, word order is constraint by grammatical structure. However, some combinations of words are subject to an additional law of constraint. This law enforces a combination of two words to occur relatively more often together than in absence of each other. Such combinations are commonly known as collocations. Depending on the corpus, examples of collocations are:

- United States
- vice president
- chief executive

Corpus linguists study such collocations to answer interesting questions about the combinatory properties of words. An example of such a question concerns the combination of verbs and prepositions: does the verb to govern occur more often in combination with the preposition *by* than with the preposition *with*?.

In the present section, we will investigate collocations in the Brown corpus. But before we do so, we first turn to the question of how to identify collocations. A simple but effective approach to collocation identification is to compare the *observed* chance of observing a combination of two words to the *expected* chance. How does this work? Well, say we have a 1000 word corpus in which the word *vice* occurs 50 times, and the word *president* 100 times. In other words, the chance that a randomly picked word is the word *vice* is *p(vice)* = 50/1000 = 0.05. In similar fashion, the chance that randomly picked word is the word president is *p(president)* = 100/1000 = 0.1. Now what would be the chance of observing the combination *vice president* if the word *vice* and *president* were "unrelated"? Well, this would simply be the chance of observing the word *vice* multiplied by the chance of observing the word *president*. Thus, *p(vice president)* = 0.05 x 0.01 = 0.005. From our thousand word corpus, we can extract 1000 - 1 = 999 bigrams. Assume that the bigram *vice president* occurs 40 times, meaning that the chance of observing this combination in our corpus is *p(vice president)* = 40 / 999 = 0.04. This reveals the observed chance of observing the combination *vice president* is larger than the expected chance. In fact we can quantify this difference in observed and expected chance for any two words *W1* and *W2*:

Equation 3.1. Difference between observed and expected chance

The observed chance of observing the combination *vice president* is eight times larger than the expected chance of observing this combination. The difference between the observed and expected chance will be large for words that occur together a lot of times, whereas it will be small for words that also occur relatively often independent of each other.

Provided this measure of difference between the observed and expected chance, we can identify the strongest collocations in a corpus by means of three steps:

1. Extract all the bigrams from the corpus
2. Compute the difference between the observed and expected chance for each bigram

3. Rank the bigrams based on these differences

The bigrams with the highest difference between observed and expected chance reflect the strongest collocations. However, the difference between observed and expected chances might easily become very large. To condense these difference values, we can represent them in logarithmic space. By doing so, we have stumbled upon a very frequent used measure of association: the so-called Pointwise Mutual Information (PMI). The PMI value for the combination of the *vice president* is:

Equation 3.2. Pointwise mutual information

Provided this association measure, we can replace step two in three steps above with: compute the PMI between the obseved and expected chance for each bigram.

Now that we know how to identify collocations, we can apply our knowledge to the Brown corpus. First we have to read in the contents of this corpus like we learned in the previous chapter:

```
*Main> h <- IO.openFile "brown.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
```

Good! From here on, let us first obtain a list of bigrams for this corpus:

```
*Main> let bgs = bigrams (words c)
*Main> head bgs
["The","Fulton"]
```

As a sanity check, we could verify whether we indeed obtained all the bigrams in the corpus. For a corpus of *n* words, we expect *n-1* bigrams:

```
*Main> length (words c)
1165170
*Main> length bgs
1165169
```

That looks great! Next we need to determine the relative frequency of each of these bigrams in the corpus. That is, for each bigram we need to determine the observed chance of observing it. We could start by determining the frequency of each bigram. We can reuse the *freqList* function defined in the previous chapter to so:

```
*Main> Data.Map.lookup ["United","States"] (freqList bgs)
Just 392
```

Todo: finish this section

From bigrams to n-grams

While extracting collocations from the Brown corpus, we have seen how useful bigrams actually are. But at this point you may be clamoring for the extraction of collocations of three or more words. For this and many other tasks, it is useful to extract so-called *n-grams* for an arbitrary *n*. We can easily modify our definition of bigrams to extract n-grams a specified length. Rather than always `take`ing two elements, we make the number of items to take an argument to the function:

```
ngrams :: Int -> [a] -> [[a]]
ngrams 0 _  = []
ngrams _ [] = []
ngrams n xs
  | length ngram == n = ngram : ngrams n (tail xs)
  | otherwise         = []
  where
    ngram = take n xs
```

We also cannot use pattern matching to exclude the tail when it is shorter than *n*. Instead, we add a guard that ends the recursion if we cannot get the proper number of elements from the list. This function works as you would expect:

```
Prelude> ngrams 3 [1..10]
[[1,2,3],[2,3,4],[3,4,5],[4,5,6],
 [5,6,7],[6,7,8],[7,8,9],[8,9,10]]
Prelude> ngrams 8 [1..10]
[[1,2,3,4,5,6,7,8],[2,3,4,5,6,7,8,9],
 [3,4,5,6,7,8,9,10]]
```

Since this is barely worth a section, we will take this opportunity to show two other implementations of the `ngrams` function. The first will be more declarative than the definition above, the second will make use of a monad that we have not used yet: the list monad.

A declarative definition of ngrams

Some patterns emerge in the recursive definition of `ngrams` that correspond to functions in the *Data.List* module:

1. Every recursive call uses the tail of the list. In other words, we enumerate every tail of the list, including the complete list. The `Data.List.tails` function provides exactly this functionality.

2. We extract the first *n* elements from every tail. This is a mapping over the data that could be performed with the `map` function.

3. The guards in the recursive case amount to filtering lists that do not have length *n*. Such filtering can also be performed by the `filter` function.

Let's go through each of these patterns to compose a declarative definition of `ngrams`. First, we extract the tails from the list, using the `tails` function:

```
Prelude> import Data.List
Prelude Data.List> let sent = ["Colorless", "green", "ideas", "sleep", "furiously"]
Prelude Data.List> tails sent
[["Colorless","green","ideas","sleep","furiously"],
["green","ideas","sleep","furiously"],["ideas","sleep","furiously"],
["sleep","furiously"],["furiously"],[]]
```

This gives us a list of tails, including the complete sentence. Now, we `map take` over each tail to extract an n-gram. Since `take` requires two arguments, we use currying to bind the first argument. For now. we will use `take 2` to extract bigrams:

```
Prelude Data.List> map (take 2) $ tails sent
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"],["furio
```

This comes close to a list of bigrams, except that we have an empty list and a list with just one member dangling at the end. These anomalies are perfect candidates to be filtered out, so we use the `filter` function in conjunction with the `length` function to exclude any element that is not of the given length. To accomplish this, we apply some currying awesomeness. Remember that we can convert infix operators to prefix operators by adding parentheses:

```
Prelude Data.List> (==) 2 2
True
Prelude Data.List> (==) 2 3
False
```

This shows that == is just an ordinary function, that just happens to use the infix notation for convenience. Since this is an ordinary function, we can also apply currying:

```
Prelude Data.List> let isTwo = (==) 2
Prelude Data.List> isTwo 2
```

```
True
Prelude Data.List> isTwo 3
False
```

Ok, so we want to check whether a list has two elements, so we could just apply `isTwo` to the result of the `length` function:

```
Prelude Data.List> isTwo (length ["Colorless","green"])
True
Prelude Data.List> isTwo (length [])
False
```

Or, written as a function definition:

```
hasLengthTwo l = isTwo (length l)
```

Since this function follows the canonical form *f (g x)*, we can use function composition:

```
Prelude Data.List> let hasLengthTwo = isTwo . length
Prelude Data.List> hasLengthTwo ["Colorless","green"]
True
```

Our filtering expression, *(==) 2 . length*, turns out to be quite compact. Time to test this with our not-yet-correct list of bigrams:

```
Prelude Data.List> filter ((==) 2 . length) $ map (take 2) $ tails sent
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"]]
```

And this corresponds to the output we expected. So, we can now wrap this expression in a function, replacing *2* by *n*:

```
ngrams' :: Int -> [b] -> [[b]]
ngrams' n = filter ((==) n . length) . map (take n) . tails
```

This function is equivalent to `ngrams` for all given lists.

You may wonder why this exercise is worthwhile. The reason is that the declarativeness of `ngrams'` makes the function much easier to read. We can almost immediately see what this function does by reading its body right-to-left, while the recursive definition requires a closer look. You will notice that, as you get more familiar with Haskell, it will become easier to spot such patterns in functions.

A monadic definition of ngrams

As discussed in the previous chapter, each type that belongs to the `Monad` typeclass provides the `(>>=)` function to combine expressions resulting in that type. The list type also belongs to the monad type class. In GHCi, you can use the **:info** command to list the type classes to which a type belongs:

```
Prelude> :i []
data [] a = [] | a : [a]  -- Defined in GHC.Types
instance Eq a => Eq [a] -- Defined in GHC.Classes
instance Monad [] -- Defined in GHC.Base
instance Functor [] -- Defined in GHC.Base
instance Ord a => Ord [a] -- Defined in GHC.Classes
instance Read a => Read [a] -- Defined in GHC.Read
instance Show a => Show [a] -- Defined in GHC.Show
```

The third line of the output shows that lists belong to the `Monad` type class. But how does the `(>>=)` function combine expressions resulting in a list? A quick peek at its definition for the list type reveals this:

```
instance Monad [] where
  m >>= k = foldr ((++) . k) [] m
  [...]
```

So, the join operation takes a list *m*, applies a function `k` to each element and concatenates the results. Of course, this concatenation implies that `k` itself should evaluate to a list, making the type signature of *k* as follows: `k :: a -> [a]`

We will illustrate this with an example. Suppose that we would want to calculate the immediate predecessor and successor of every number in the list *[0..9]*. In this case, we could use the function `\x -> [x-1,x+1]` in the list monad:

```
Prelude> :{
do
  l  <- [0..9]
  ps <- (\x -> [x-1,x+2]) l
  return ps
:}
[-1,2,0,3,1,4,2,5,3,6,4,7,5,8,6,9,7,10,8,11]
```

First, the list is bound to *l*, then our predecessor/successor function is applied to *l*. Since we are using this function in the context of the list monad, the function is be applied to every member of *l*. The results of these applications is concatenated.

Note

Experimenting with list monads may give you results that may be surprising at first sight. For instance:

```
Prelude> :{
do
  l <- [0..9]
  m <- [42,11]
  return m
:}
[42,11,42,11,42,11,42,11,42,11,42,11,42,11,42,11,42,11,42,11]
```

Since *[42,11]* in *m <- [42,11]* does not use an argument, its corresponding function is `\_ -> [42,11]`. Since `foldr` still traverses the list bound to *l*, the monadic computation is equal to:

```
Prelude> foldr ((++) . (\_ -> [42,11])) [] [0..9]
[42,11,42,11,42,11,42,11,42,11,42,11,42,11,42,11,42,11,42,11]
```

We can also extract bigrams using the list monad. Given a list of tails, we could extract the first two words of each tail using `take`:

```
Prelude> import Data.List
Prelude Data.List> let sent = ["Colorless", "green", "ideas", "sleep", "furiously"]
Prelude Data.List> :{
do
  t <- tails sent
  l <- take 2 t
  return l
:}
["Colorless","green","green","ideas","ideas","sleep","sleep","furiously","furiously"]
```

That's close. However, since the list monad concatenates the results of every *take 2 t* expression, we cannot directly identify the n-grams anymore. This is easily remedied by wrapping the result of `take` in a list:

```
Prelude Data.List> :{
do
  t <- tails sent
  l <- [take 2 t]
  return l
:}
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"],["furio
```

Now we get the n-grams nicely as a list. However, as in previous definitions of `ngrams` we have to exclude lists that do not have the requested number of elements. We could, as we did previously, filter out these members using `filter`:

```
Prelude Data.List> :{
filter ((==) 2 . length) $ do
  t <- tails sent
  l <- [take 2 t]
  return l
:}
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"]]
```

But that would not be a very monadic way to perform this task. It would be nice if we could just choose elements to our liking. Such a (monadic) choice function exists, namely `Control.Monad.guard`:

```
Prelude Data.List> import Control.Monad
Prelude Data.List Control.Monad> :type guard
guard :: MonadPlus m => Bool -> m ()
```

`guard` is a function that takes a boolean, and returns something that is a `MonadPlus`. Whoa! For now, accept that the list type belongs to the `MonadPlus` type class (after importing *Control.Monad*). Instead of going into the working of `MonadPlus` now, we will perform a behavioral study of `guard`:

```
Prelude Data.List Control.Monad> :{
do
  l <- [0..9]
  guard (even l)
  return l
:}
[0,2,4,6,8]
```

Funky huh? We used `guard` to enumerate just those numbers from *[0..9]* that are even. Of course, we could as well use `guard` in our bigram extraction to filter lists that are not of a certain length:

```
Prelude Data.List Control.Monad> :{
do
  t <- tails sent
  l <- [take 2 t]
  guard (length l == 2)
  return l
:}
[["Colorless","green"],["green","ideas"],["ideas","sleep"],["sleep","furiously"]]
```

Ain't that beautiful? We applied a guard to pick just those elements that are of length *2*, or as you might as well say, we put a constraint on the list requiring elements to be of length *2*. We can easily transform this expression to a function, by making the n-gram length and the list arguments of that function:

```
ngrams'' :: Int -> [a] -> [[a]]
ngrams'' n l = do
  t <- tails l
  l <- [take n t]
  guard (length l == n)
  return l
```

As you can conclude from the previous sections, there is often more than one way to implement a function. In practice you will want to pick a declaration that is readable and performant. In this case, we think that the declarative definition of `ngrams` is the most preferable.

Exercises

1. Rewrite the `skipBigram` function discussed in the section called “Exercises” without explicit recursion, either by defining it more declaratively or using the list monad. Hint: make use of the `Data.List.zip` function.

Lazy and strict evaluation

You may have noticed that something curious goes on in Haskell. For instance, consider the following GHCi session:

```
Prelude> take 10 $ [0..]
[0,1,2,3,4,5,6,7,8,9]
```

The expression `[0..` is the list of numbers from zero to infinity. Obviously, it is impossible to store an infinite list in finite memory. Haskell does not apply some simple trick, since it also works in less trivial cases. For instance:

```
Prelude> take 10 $ filter even [0..]
[0,2,4,6,8,10,12,14,16,18]
```

This also works for your own predicates:

```
Prelude> let infinite n = n : infinite (n + 1)
Prelude> take 3 $ infinite 0
[0,1,2,3]
```

In most other programming languages, this computation will never terminate, since it will go into an infinite recursion. Haskell, however, won't. The reason is that Haskell uses *lazy evaluation* - an expression is only evaluated when necessary. For instance, taking three elements from `infinite` results in the following evaluations:

```
infinite 0
0 : infinite 1
0 : (1 : infinite 2)
0 : (1 : (2 : infinite 3))
0 : (1 : (2 : (3 : infinite 4)))
```

Once `take` has consumed enough elements from `infinite`, the tail of the list is the expression `infinite 4`. Since `take` does not need more elements, the tail is never evaluated. Lazy evaluation allows you to do clever tricks, such as defining infinite lists. The downside is that it is often hard to predict when an expression is evaluated, and what effect that has on performance of a program.

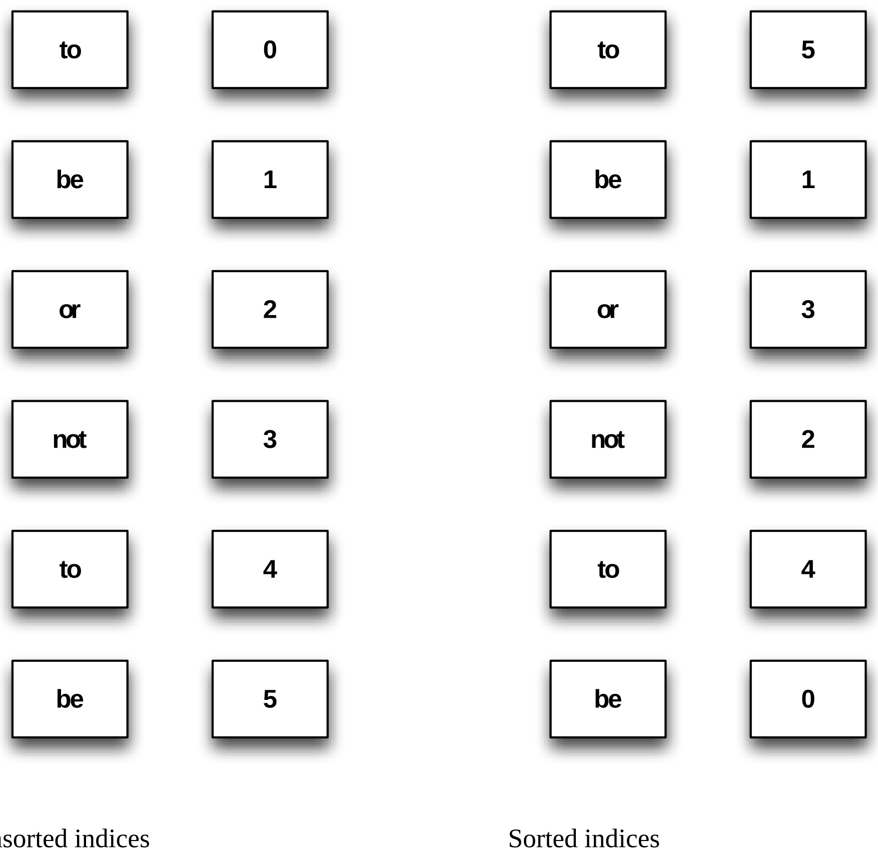
Todo: lazy evaluation and folds.

Suffix arrays

In this chapter, we have seen how you could extract an n-gram of a given *n* from a list of words, characters, groceries, or whatever you desire. You can also store n-gram frequencies in a Map, to build applications that quickly need the frequency (or probability) of an n-gram in a corpus. What if you would encounter an application where you need access to n-grams of any length? Any! From unigrams to 'almost the length of your corpus'-grams. Obviously, if your corpus contains *m* elements, storing frequencies of all 1..m-grams would make your program a memory hog.

Fortunately, it turns out that there is a simple and smart trick to do this, using a data structure called *suffix arrays*. First, we start with the corpus, and a parallel list or array where each element contains an index that can be seen as a pointer into the corpus. The left side of figure Figure 3.1, “Constructing a suffix array” shows the initial state for the phrase "to be or not to be". We then sort the array of indices by comparing the elements they point to. For instance, we could compare the element with index 2 ("or") and the element with index 3 ("not"). Since "not" is lexicographically ordered before "or", the list of indices should be sorted such that the element holding index 3 comes before 2. When two indices point to equal elements, e.g. 0 and 4 ("to"), we move on to the element that succeed both instances of "to", respectively "be" and "be". And we continue such comparisons recursively, until we find out that one n-gram is lexicographically sorted before the other (in this case, 4 should come before 0, since "to be" is lexicographically sorted before "to be or". The right side of figure Figure 3.1, “Constructing a suffix array” shows how the indices will be sorted after applying this sorting methodology.

Figure 3.1. Constructing a suffix array



After sorting the list of indices in this manner, the index list represents an ordered list of n-grams within the corpus. The length of the n-gram does not matter, since elements and their suffixes were compared until one element could be sorted lexicographically before the other. This ordering also implies that we can use a binary search to check whether an n-gram occurred in the corpus, and if so, how often. But more on that later...

Of course, as a working programmer you can't wait to fire up your text editor to implement suffix arrays. It turns out to be simpler than you might expect. But, we need to introduce another data type first, the vector. It is a data type that is comparable to arrays in other programming languages. Vectors allow for random access to array elements. So, if you want to access the *n*-th element of a vector, it can be accessed directly, rather than first traversing the *n-1* preceding elements as in a list. Vectors are provided in Haskell as a part of the vector package that can be installed using **cabal**. We can construct a Vector from a list and convert a Vector to a list:

```
Prelude> Data.Vector.fromList ["to","or","not","to","be"]
fromList ["to","or","not","to","be"] :: Data.Vector.Vector
Prelude> Data.Vector.toList $ Data.Vector.fromList ["to","or","not","to","be"]
["to","or","not","to","be"]
```

The `(!)` function is used to access an element:

```
Prelude> (Data.Vector.fromList ["to","or","not","to","be"]) Data.Vector.! 3
"to"
```

There's also a safe access function, `(!?)`, that wraps the element in a Maybe. Nothing is returned when you use an index that is 'outside' the vector:

```
Prelude> (Data.Vector.fromList ["to","or","not","to","be"]) Data.Vector.! 20
"*** Exception: ./Data/Vector/Generic.hs:222 ((!)): index out of bounds (20,5)
```

```
Prelude> (Data.Vector.fromList ["to","or","not","to","be"]) Data.Vector.!? 20
Nothing
Prelude> (Data.Vector.fromList ["to","or","not","to","be"]) Data.Vector.!? 3
Just "to"
```

That enough for now. The primary reason why Vector is a useful type here, is because we want random access to the corpus during the construction of the suffix array. After construction, it is also useful for most tasks to be able to access the indices randomly. Alright, first we create a data type for the suffix array:

```
import qualified Data.Vector as V

data SuffixArray a = SuffixArray (V.Vector a) (V.Vector Int)
                     deriving Show
```

It says exactly what we saw in the figure above: a suffix array consists of a data vector (in our case a corpus) and a vector of indices, respectively V.Vector a and V.Vector Int. Ideally, we would like to construct a suffix array from a list. However, to do this, we need a sorting function. The *Data.List* module contains the `sortBy` function that sorts a list according to some ordering function:

```
*Main> :type Data.List.sortBy
Data.List.sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

So, it takes a comparison function that should compare two elements, and that returns Ordering. Ordering is a data type that specifies... order. There are three constructors: LT, EQ, andGT, these constructors indicate respectively that the first argument is less than, equal to, or greater than the second argument.

We will use `sortBy` to sort the list of indices. Since the ordering of the indices is determined by elements of the data array, to which the indices refer, the comparison function that we provide for sorting the index array requires access to the data array. So, our function will compare (sub)vectors, indicated by their indices. This will work, since the Data.Vector data type is of the Ord type class, meaning that the operators necessary for comparisons are provided. Our comparison function can be written like this:

```
saCompare :: Ord a => (V.Vector a -> V.Vector a -> Ordering) ->
             V.Vector a -> Int -> Int -> Ordering
saCompare cmp d a b = cmp (V.drop a d) (V.drop b d)
```

To allow a user of our function to impose their own sorting order (maybe the want to make a reversibly offered suffix array), we `saCompare` requires a comparison function as its first argument. The second argument is the data vector, and the final two arguments are the indices to be compared. We can get the subvectors represented by the two indices by using the `Data.Vector.drop` function. Suppose, if we want the element at index two, we can just drop the first two arguments, since we start counting at zero. We then use the provided comparison function to compare the two subvectors.

Now we can create the function that actually creates a suffix array:

```
import qualified Data.List as L

suffixArrayBy :: Ord a => (V.Vector a -> V.Vector a -> Ordering) ->
                 V.Vector a -> SuffixArray a
suffixArrayBy cmp d = SuffixArray d (V.fromList srtIndex)
    where uppBound = V.length d - 1
          usrtIndex = [0..uppBound]
          srtIndex = L.sortBy (saCompare cmp d) usrtIndex
```

This function is fairly simple, first we create the unsorted list of indices and bind it to `usrtIndex`. We construct this list by using a range. A range contains the indicated lower bound and upper bound, and all integers in between;

```
*Main> [0..9]
[0,1,2,3,4,5,6,7,8,9,10]
```

We retrieve the upper bound using the `Data.Vector.length` function by subtracting one, since we are counting from zero. We then obtain the sorted index (`srtIndex`) by using the `Data.List.sortBy` function. This function takes a comparison function as its first argument and a list as its second argument:

```
*Main> :type Data.List.sortBy
Data.List.sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

We can just plug in our `saCompare` function, which we pass a comparison function, and the data vector. Finally, we use the SuffixArray constructor to construct a SuffixArray, converting the list of indices to a vector. For convenience, we can also add a function that uses Haskell's `compare` function that uses the default sorting order that is imposed by the Ord typeclass:

```
suffixArray :: Ord a => V.Vector a -> SuffixArray a
suffixArray = suffixArrayBy compare
```

Neat! But as you have noticed by now, every serious data type has `fromList` and `toList` functions, so ours should have those as well. `fromList` is really simple; we can already construct a suffix array from a Vector using the `suffixArray` function. So, we just need to convert a list to a Vector, and pass it to `suffixArray`:

```
fromList :: Ord a => [a] -> SuffixArray a
fromList = suffixArray . V.fromList
```

Easy huh? The `toList` is a bit more involved. First we have to decide what it should actually return. Providing the data vector as a list is not very useful, it's probably what someone started with. Returning a list of indices is more useful, but then we shift the burden off retrieving the n-grams that every index represents to the user of our suffix array. The most useful thing would be to return a list of all n-grams (of any length). So, for the phrase "to be or not to be", we want to return the following elements:

- ["be"]
- ["be","or","not","to","be"]
- ["not","to","be"]
- ["or","not","to","be"]
- ["to","be"]
- ["to","be","or","not","to","be"]

To achieve this, we need to extract the subvector for each index, in the order that the sorted vector of indices indicates. We can then convert each subvector to a list. We can use `Data.Vector.foldr` function to traverse the vector, constructing a list for each index. We will accumulate these lists in (yet another) list. Please welcome `toList`:

```
toList :: SuffixArray a -> [[a]]
toList (SuffixArray d i) = V.foldr vecAt [] i
    where vecAt idx l = V.toList (V.drop idx d) : l
```

The `vecAt` function extracts a subvector starting at index `idx`, converts it to a list. We form a new list, with the accumulator as the tail, and the newly constructed 'subvector list' as the head. We use `foldr` to ensure that the list that is being constructed is in the correct order - since the accumulator becomes the tail, a `foldl` would make the first subarray the last in the list. Time to play with our new data type a bit:

```
*Main> toList $ fromList ["to","be","or","not","to","be"]
[["be"],
["be","or","not","to","be"],
["not","to","be"],
["or","not","to","be"],
["to","be"],
["to","be","or","not","to","be"]]
```

Excellent, just as we want it: we get an ordered list of all n-grams in the corpus, for the maximum possible *n*. We can use this function to extract all bigrams:

```
*Main> filter ((== 2) . length) $ map (take 2) $ toList $ \
  fromList ["to","be","or","not","to","be"]
```

We extract the first two elements of each n-gram. This also gives us one unigram (the last token of the corpus), so we also have to filter the list for lists that contain two elements.

After some celebrations and a cup of tea, it is time to use suffix arrays to find the frequency of a word. To do this, we use a binary search. For quick accessibility, we create a function comparable to the `toList` method, but returning a Vector of Vector, rather than a list of list:

```
elems :: SuffixArray a -> V.Vector (V.Vector a)
elems (SuffixArray d i) = V.map vecAt i
    where vecAt idx = V.drop idx d
```

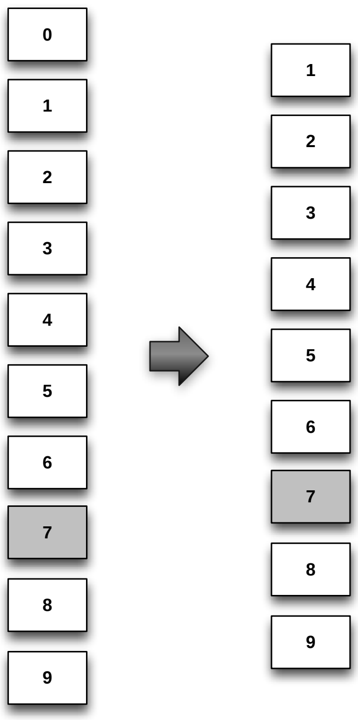
Note that we can use `Data.Vector.map` in this case, since it maps a function over all elements of vector, returning a vector:

```
*Main> :type Data.Vector.map
Data.Vector.map :: (a -> b) -> V.Vector a -> V.Vector b
```

Note: if you have a computer science background, you might want to skip the next paragraphs.

To be able to count the number of occurrences of an n-gram in the suffix array, we need to locate the n-gram in the suffix array first. We could just traverse the array from beginning to the end, comparing each element to the n-gram that we are looking for. However, this is not very inefficient. During every search step, we exclude just one element. For instance, if we have the numbers 0 to 9 and have to find the location of the number 7, the first search step would just exclude the number 0, leaving eight potential candidates (Figure 3.2, “Linear search step”).

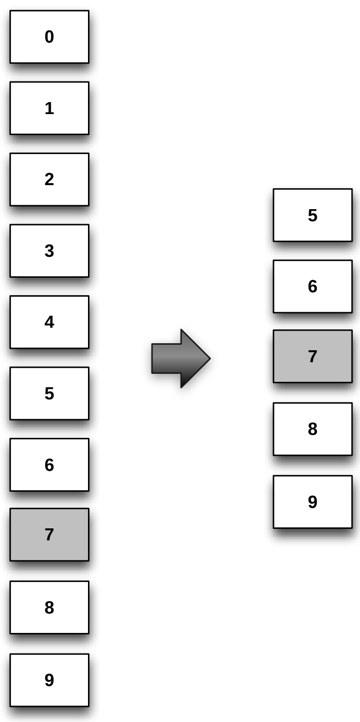
Figure 3.2. Linear search step



However, if we know that the vector of numbers is sorted, we can devise a more intelligent strategy. As a child, you probably played number guessing games. In one variant of the game, you would guess a number, and the person knowing the correct number would shout "smaller", "larger" or "correct". Being a smart kid, you would probably not start guessing at 1 if you had to guess a number between 1 and 100. Usually, you'd start somewhere halfway the range (say 50), and continue halfway the 1..50 or 51..100 range if the number was smaller or greater than 50.

The same trick can be applied when searching a sorted vector. If you compare a value to the element in the middle, you remove cut half of the search space (if initial guess was not correct). This procedure is called a *binary search*. For instance, Figure 3.3, “Binary search step” shows the first search step when applying a binary search to the example in Figure 3.2, “Linear search step”.

Figure 3.3. Binary search step



The performance of binary search compared to linear search should not be underestimated: the time of a linear search grows linearly with the number of elements (yes, we like pointing out the obvious), while time of a binary search grows logarithmically. Suppose that we have a sorted vector of 1048576 elements, a linear search would at most take 1048576 steps, while a binary search takes at most 20 steps. Pretty impressive right?

On to our binary search function. `binarySearchByBounded` finds the index of an element in a Vector, wrapped in Maybe. If the element has multiple occurrences in the Vector, just one index is returned. If the element is not in the Vector, Nothing is returned.

```
binarySearchByBounded :: (Ord a) => (a -> a -> Ordering) -> V.Vector a ->
                         a -> Int -> Int -> Maybe Int
binarySearchByBounded cmp v e lower upper
    | V.null v      = Nothing
    | upper < lower = Nothing
    | otherwise     = case cmp e (v V.! middle) of
                        LT -> binarySearchByBounded cmp v e lower (middle - 1)
                        EQ -> Just middle
                        GT -> binarySearchByBounded cmp v e (middle + 1) upper
    where middle    = (lower + upper) `div` 2
```

`binarySearchByBounded` takes a host of arguments: a comparison function, the (sorted) vector (`v`, the element to search for (`e`), and lower (`lower`) and upper bound (`upper`) indices of the search space.

The function works just like we described above. First we have to find the middle of the current search space, we do this by averaging the upper and lower bounds and binding it to `middle`. We then compare the element at index `middle` in the vector to `e` . If both are equal (EQ), then we are done searching, and return `Just middle` as the index. If `e` is smaller than (LT) the current element, we search in the lower half of the search space (`lower`..`middle`-1). If `e` is greater than (GT) the current element, we search in the upper half of the search space (`middle`+1..`upper`). If `e` does not occur in the search space, `upper` will become smaller than `lower` when we have exhausted the search space.

Let's define two convenience functions to make binary searches a bit simpler:

```
binarySearchBounded :: (Ord a) => V.Vector a -> a -> Int -> Int -> Maybe Int
binarySearchBounded = binarySearchByBounded compare

binarySearchBy :: (Ord a) => (a -> a -> Ordering) -> V.Vector a -> a ->
                  Maybe Int
binarySearchBy cmp v n = binarySearchByBounded cmp v n 0 (V.length v - 1)

binarySearch :: (Ord a) => V.Vector a -> a -> Maybe Int
binarySearch v e = binarySearchBounded v e 0 (V.length v - 1)
```

`binarySearchBounded` calls `binarySearchByBounded`, using Haskell's standard compare function. `binarySearchBy` calls `binarySearchByBounded`, binding the upper and lower bounds to the lowest index of the array (0) and the highest (the size of the Vector minus one). Finally, `binarySearch` combines the functionality of `binarySearchBounded` and `binarySearchBy`, Let's give the binary search functionality a try:

```
*Main> binarySearch (V.fromList [1,2,3,5,7,9]) 9
Just 1
*Main> binarySearch (V.fromList [1,2,3,5,7,9]) 10
Just 5
*Main> binarySearch (V.fromList [1,2,3,5,7,9]) 10
```

Great! Let's make a step in between, returning to suffix arrays. Say that you would want to write a `contains` function that returns True if an n-gram is in the suffix array, or False otherwise. Easy right? Your first attempt may be something like:

```
*Main> let corpus = ["to","be","or","not","to","be"]
*Main> binarySearch (elems $ fromList corpus) $ Data.Vector.fromList ["or","not", "to",
Just 3
```

Nice, right? But try this example:

```
*Main> binarySearch (elems $ fromList corpus) $ Data.Vector.fromList ["or","not"]
Nothing
```

You can almost hear the commentator of *Roger Wilco and the Time Rippers* in the background, right? Right! Of course, the element that we are looking for contains the n-gram of the maximum length ("or not to be"). That is why the first example worked, while the second did not. So, we have to apply the binary search to something that only contains bigrams in this case:

```
*Main> :{
*Main| binarySearch
*Main|   (Data.Vector.map (Data.Vector.take 2) $ elems $ fromList corpus) $
*Main|   Data.Vector.fromList ["or","not"]
*Main| :}
Just 3
```

That did the trick. Writing the contains function is now simple:

```
contains :: Ord a => SuffixArray a -> V.Vector a -> Bool
contains s e = case binarySearch (restrict eLen s) e of
                 Just _  -> True
                 Nothing -> False
    where eLen = V.length e
```

```
        restrict len = V.map (V.take len) . elems
```

To find the frequency of an element in a Vector, we have to do a bit more than locating one instance of that element. One first intuition could be to find the element, and scan upwards and downwards to find how many instances of the element there are in the Vector. However, there could be millions of such elements. Doing a linear search is, again, not very efficient. So, we should apply a binary search, but not just to find one instance of the element, but specifically the first and the last.

Such search functions are very comparable to the `binarySearchByBounds` function that we wrote earlier. Let's start with finding the first index in the Vector where a specified element occurs. Suppose that we do a binary search again: if the element in the middle of our search space is greater than the element, we want to continue searching in the lower half of the search space. If the element in the middle is smaller than the element, we want to continue searching in the upper half of the search space. If the middle is however equal to the element, we do not stop searching, but continue searching the lower half. We still keep the element that was equal in the search space, since it may have been the only instance of that element. This gives us the following `lowerBoundByBounds` function and corresponding helpers:

```
lowerBoundByBounds :: Ord a => (a -> a -> Ordering) -> V.Vector a -> a ->
                      Int -> Int -> Maybe Int
lowerBoundByBounds cmp v e lower upper
    | V.null v = Nothing
    | upper == lower = case cmp e (v V.! lower) of
                         EQ -> Just lower
                         _  -> Nothing
    | otherwise = case cmp e (v V.! middle) of
                    GT -> lowerBoundByBounds cmp v e (middle + 1) upper
                    _  -> lowerBoundByBounds cmp v e lower middle
    where middle = (lower + upper) `div` 2

lowerBoundBounds :: Ord a => V.Vector a -> a -> Int -> Int -> Maybe Int
lowerBoundBounds = lowerBoundByBounds compare

lowerBoundBy :: Ord a => (a -> a -> Ordering) -> V.Vector a -> a -> Maybe Int
lowerBoundBy cmp v e = lowerBoundByBounds cmp v e 0 (V.length v - 1)

lowerBound :: Ord a => V.Vector a -> a -> Maybe Int
lowerBound = lowerBoundBy compare
```

Searching the last index in the Vector where the element occurs, follows a comparable procedure. We search as normal, however if the element is equal to the middle we search the upper half of the search space including the element that we found to be equal. Give the floor to `upperBoundByBounds` and helpers:

```
upperBoundByBounds :: Ord a => (a -> a -> Ordering) -> V.Vector a -> a ->
                      Int -> Int -> Maybe Int
upperBoundByBounds cmp v e lower upper
    | V.null v       = Nothing
    | upper <= lower = case cmp e (v V.! lower) of
                         EQ -> Just lower
                         _  -> Nothing
    | otherwise      = case cmp e (v V.! middle) of
                         LT -> upperBoundByBounds cmp v e lower (middle - 1)
                         _  -> upperBoundByBounds cmp v e middle upper
    where middle     = ((lower + upper) `div` 2) + 1

upperBoundBounds :: Ord a => V.Vector a -> a -> Int -> Int -> Maybe Int
upperBoundBounds = upperBoundByBounds compare

upperBoundBy :: Ord a => (a -> a -> Ordering) -> V.Vector a -> a -> Maybe Int
upperBoundBy cmp v e = upperBoundByBounds cmp v e 0 (V.length v - 1)

upperBound :: Ord a => V.Vector a -> a -> Maybe Int
upperBound = upperBoundBy compare
```

Note that we add one to the middle in this case. This is to avoid landing in an infinite recursion when `middle` is `lower` plus one, and the element is larger than or equal to the element at `middle`. Under those circumstances, `lower` and `upper` would be unchanged in the next recursion.

Great. I guess you will now be able to write that function in terms of `lowerBoundByBounds` and `upperBoundByBounds`:

```
frequencyByBounds :: Ord a => (a -> a -> Ordering) -> V.Vector a -> a ->
                     Int -> Int -> Maybe Int
frequencyByBounds cmp v e lower upper = do
  lower <- lowerBoundByBounds cmp v e lower upper
  upper <- upperBoundByBounds cmp v e lower upper
  return $ upper - lower + 1

frequencyBy :: Ord a => (a -> a -> Ordering) -> V.Vector a -> a ->
               Maybe Int
frequencyBy cmp v e = frequencyByBounds cmp v e 0 (V.length v - 1)

frequencyBounds :: Ord a => V.Vector a -> a -> Int -> Int -> Maybe Int
frequencyBounds = frequencyByBounds compare

frequency :: Ord a => V.Vector a -> a -> Maybe Int
frequency = frequencyBy compare
```

This function works as expected:

```
*Main> frequency (V.fromList [1,3,3,4,7,7,7,10]) 7
Just 3
*Main> frequency (V.fromList [1,3,3,4,7,7,7,10]) 5
Nothing
```

We can use this with our suffix array now:

```
*Main> let corpus = ["to","be","or","not","to","be"]
*Main> let sa = fromList corpus
*Main> containsWithFreq sa $ Data.Vector.fromList ["not"]
Just 2
*Main> containsWithFreq sa $ Data.Vector.fromList ["not"]
Just 1
*Main> containsWithFreq sa $ Data.Vector.fromList ["jazz","is","not","dead"]
Nothing
*Main> containsWithFreq sa $ Data.Vector.fromList ["it","just","smells","funny"]
Nothing
```

Exercises

1. Write a function `mostFrequentNgram` with the following type signature:

   ```
   mostFrequentNgram :: Ord a => SuffixArray a -> Int -> Maybe (V.Vector a, Int)
   ```

   This function extracts the most frequent n-gram from a suffix array, where the suffix array and *n* are given as arguments. The function should continue a pair of the n-gram and the frequency as a typle wrapped in Maybe. If no n-gram could be extracted (for instance, because the suffix array contains to few elements), return Nothing.

2. Use `mostFrequentNgram` to find the most frequent bigram and trigram in the Brown corpus.

3. frequencyByBounds is not as efficient as it could be: it performs a search of the full Vector twice. A more efficient solution would be to narrow down the search space until the first match is found, and then using `lowerBoundByBounds` and `upperBoundByBounds` to search the lower and upper half of the search space. Modify `frequencyByBounds` to use this methodology.

Markov models

At the beginning of this chapter we mentioned that n-grams can be exploited to model language. While they may not be so apt as computational grammars, n-grams do encode some syntax albeit local. For instance, consider the following to phrases:

- the plan was
- * plan the was

The first phrase is clearly grammatical, while the second is not. We could neatly encode this using a syntax rule, but we could also count how often both combinations of words occur in a large text corpus. The first phrase is likely to occur a few times, while the second phrase is not likely to occur. Or more formally, the probability that we encounter *this plan was* occurs in a random text is higher than the probability that *plan this was* occurs:

Of course, we could also try to find the most grammatical of two sentences by comparing the probabilities of the sentences. So, if we have a sentence consisting of the words $w_{0..n}$ and a sentence consisting of the words $v_{0..m}$ that both aim to express the same meaning and the following is true:

We could conclude that the use of $w_{0..n}$ is preferred over $v_{0..m}$, since $w_{0..n}$ is either more grammatical or more fluent. So, how do we estimate the probability of such a sentence? Good question, at first sight it seems pretty easy. We simply count how often a sentence occurs in a text corpus, and divide it by the total number of sentences in a corpus:

Equation 3.3. Estimating the probability of a sentence

Here *C* is a counter function, and *N* is the total number of sentences in a corpus. While this is a theoretically sound method for estimating the probability, it does not work in practice. As ingenious as human language is, we can construct an infinite number of grammatical sentences. So, to be able to estimate the probability we would need an infinite text corpus, since not every grammatical sentence will occur in a finite corpus. Given that we only have a finite text corpus, we would simply give a probability of zero to many perfectly grammatical sentences. We encounter so-called *data sparseness*. This is nasty, because it interferes with our goal to compare the quality of sentences.

Fortunately for us, some smart people have thought about this problem, and came up with a pretty elegant solution (or `workaround' as we programmers like call it). To get to the solution, we have to make an intermediate step. This intermediate step does not immediately solve our problem, but sets the stage for the solution. We can decompose the probability of a sentence $p(w_{0..n})$ into a series of conditional probabilities:

Equation 3.4. The probability of a sentence as a Markov chain

Before this gets too confusing, let's write down how you would estimate the probability of the sentence *Colorless green ideas sleep furiously* in this manner: *p(Colorless) p(green|Colorless) p(ideas|Colorless green) p(sleep|Colorless green ideas) p(furiously|Colorless green ideas sleep)*.

Simple huh? Now, how do we estimate such a conditional probability? Formally, this is estimated in the following manner:

That is all nice and dandy, but as you may already see, this does not solve our problem with data sparseness. For if we want to estimate *p(furiously|Colorless green ideas sleep)*, we need counts of *Colorless green ideas sleep* and *Colorless green ideas sleep furiously*. Even if we decompose the probability of a sentence into conditional probabilities, we need counts for the complete sentence.

However, if we look at the conditional probability of a word, the following often holds:

Equation 3.5. Approximation using the Markov assumption

More formally, this is a process with the *Markov property*: prediction of the next *state* (word) is only dependent on the current state. Of course, we can easily calculate our revised conditional probability:

Equation 3.6. The conditional probability of a word using the Markov assumption

That spell worked! We only need counts of... unigrams (1-grams) and bigrams to estimate the conditional probability of each word. This is a *bigram language model*, which we can use to estimate to probability of a sentence:

Equation 3.7. The probability of a sentence using a bigram model

In practice it turns out that knowledge of previous states can help a bit in estimating the conditional probability of a word. However, if we increase the context too much, we run into the same data sparseness problems that we solved by drastically cutting the context. The consensus is that for most applications a trigram language model provides a good trade-off between data availability and estimator quality.

Implementation

The implementation of a bigram Markov model in Haskell should now be trivial. If we have a frequency map of unigrams and bigrams of the type (Ord a, Integral n) => Map [a] n, we could write a function that calculates , or more generally :

```
import qualified Data.Map as M
import Data.Maybe (fromMaybe)

pTransition :: (Ord a, Integral n, Fractional f) =>
  M.Map [a] n -> a -> a -> f
pTransition ngramFreqs state nextState = fromMaybe 0.0 $ do
  stateFreq <- M.lookup [state] ngramFreqs
  transFreq <- M.lookup [state, nextState] ngramFreqs
  return $ (fromIntegral transFreq) / (fromIntegral stateFreq)
```

Now we write a function that extracts all bigrams, calculates the transition probabilities and takes the product of the transition probabilities:

```
pMarkov :: (Ord a, Integral n, Fractional f) =>
  M.Map [a] n -> [a] -> f
pMarkov ngramFreqs =
  product . map (\[s1,s2] -> pTransition ngramFreqs s1 s2) . ngrams 2
```

This function is straightforward, except perhaps the `product` function. `product` calculates the product of a list:

```
Prelude> :type product
```

```
product :: Num a => [a] -> a
Prelude> product [1,2,3]
6
```

Chapter 4. Distance and similarity (proposed)

Chapter 5. Classification

Introduction

Many natural language processing tasks require classification, you want to find out to which class a particular instance belongs. To make this more concrete, we give three examples:

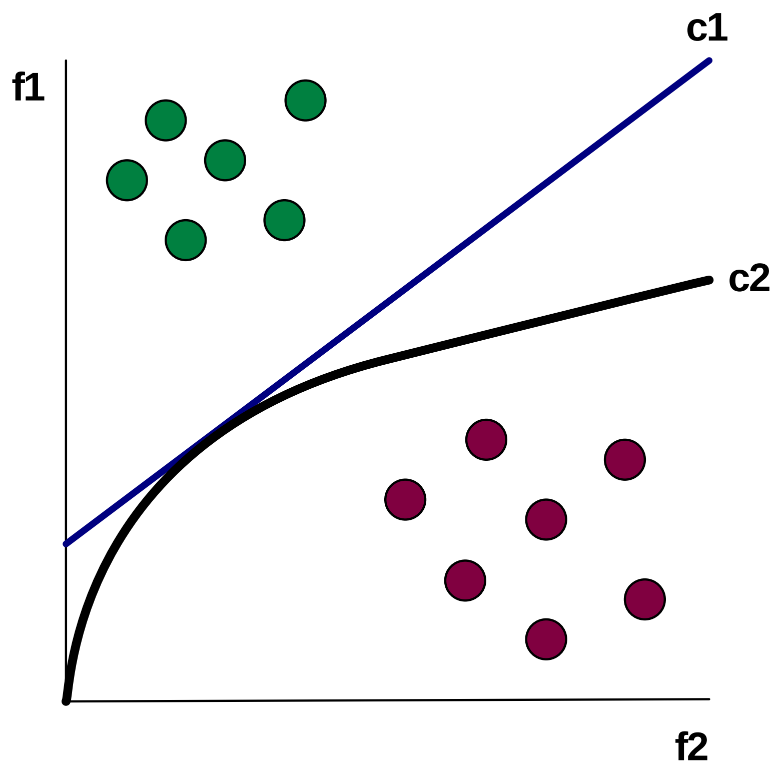
- **Authorship attribution**: suppose that you were given a text, and have to pick the correct author of the text from three proposed authors.
- **Part of speech tagging**: in part of speech tagging, words are classified morphosyntactically. For instance, we could classify the word 'loves' in the statement "John loves Mary" to be a verb.
- **Fluency ranking**: in natural language generation, we want to find out whether a sentence produced by a generation system is fluent or not fluent.

Such classifications can be made based on specific characteristics of the instance that we want to specify. These characteristics are called *features* in natural language processing jargon. Suppose that you were asked to determine the author of a text, and know that Jack tends to write short sentences, while Steven and Marie tend to write long sentences. Now, if you were given a text with mainly short sentences, who would you attribute the text to? Probably Jack, right? Average sentence length is one possible feature to classify the text by its author.

More formally speaking, we want to estimate $p(y|x)$, the probability of an *event* (classification), given a *context*. For instance, in authorship attribution, the classification being a specific author is an event, while the text is the context. In part of speech tagging, the classification of a word as verb is an event, the word and surrounding words are the context.

In this chapter, we will look at *linear classifiers*. A linear classification can make a classification based on a linear combination of features. To give an example, consider Figure 5.1, “Linear and non-linear classifiers”. Here we see two classes of objects, that can be separated using just two features (*f1* and *f2*). One class is tends to have high *f1* values, the other high *f2* values. The figure also shows two classifiers, *c1* and *c2*, that successfully separate both classes. *c1* is a linear classifier, as it is a linear combination of *f1* and *f2*. *c2*, on the other hand, is not a linear classifier: the effect of *f1* becomes weaker as *f2* increases.

Figure 5.1. Linear and non-linear classifiers



How do we find such functions? Doing it manually is not practical - realistic models for natural language processing classification use thousands to millions of features. Finding such functions is an art in itself, and is usually called *machine learning*. Machine learning methods learn such classifiers through training material. Machine learning methods are a topic by themselves, so in this chapter we will mainly look at the application of classifiers obtained through machine learning.

This may all seem somewhat abstract at this point, but things will get clearer as we dive into real classifiers. The thing to remember now is that we want to attach a particular class label to instances, based on features of that instance, and we will do this using linear classifiers.

Naive Bayes classification

Stub

Maximum entropy classification

Introduction

An important disadvantage of naive Bayes modelling is that it has strong feature independence assumptions. Often, it is not clear whether features are dependent, or you simply do not want to care. For instance, coming back to the task of authorship attribution. Suppose that you made a model that uses the average sentence length as a feature amongst others. Now you got a fantastic idea, you want to add some features modeling syntactic complexity of sentences in a text. Such features may add new cues to the model, but syntactic complexity also has a correlation with sentence length. In such situations, naive Bayes models may fail, since they see these features as independent contributors to a classification.

One class of models that do not assume independent features are maximum entropy models. We can almost hear you think "isn't classification supposed to minimize uncertainty"? That's a very good question, that we will come to in a moment. First, we have to ask a basic question: given a collection of training instances, what is a good model? Think about this for a moment, without diving into theory and technicalities.

The answer is pretty simple: since the training data is (supposed to be) a representative sample of reality, a good model would predict the training data. What does it mean to predict the training data? You may remember from high school math that you could calculate the expected value of a random variable given a *probability distribution*. For instance, if you play a coin tossing game, you can calculate the (average) profit or loss given enough tosses. Suppose that a friend has a biased coin, with *p(heads) = 0.7* and *p(tails) = 0.3*. Winning gives you Euro 1.50, when losing, you pay 1 Euro. The expected outcome of choosing tails is *0.7 * -1 + 0.3 * 1.50 # -0.25*. Not such a good bet, huh?

If we know the expected value from the observation of repeated coin flips (the training data), we can make a model that gives the same outcome. If we know the payments, finding the model analytically is trivial. What if we do the same for features? We can calculate the feature value in the training data:

Equation 5.1. Calculating the empirical value of a feature

It's easier than it looks: the empirical value of a feature f_i is the sum of the multiplication joint probability of a context and an event in the training data and the value of f_i for that context and event. We can also calculate the expected value of a feature f_i according to the conditional model p(y|x):

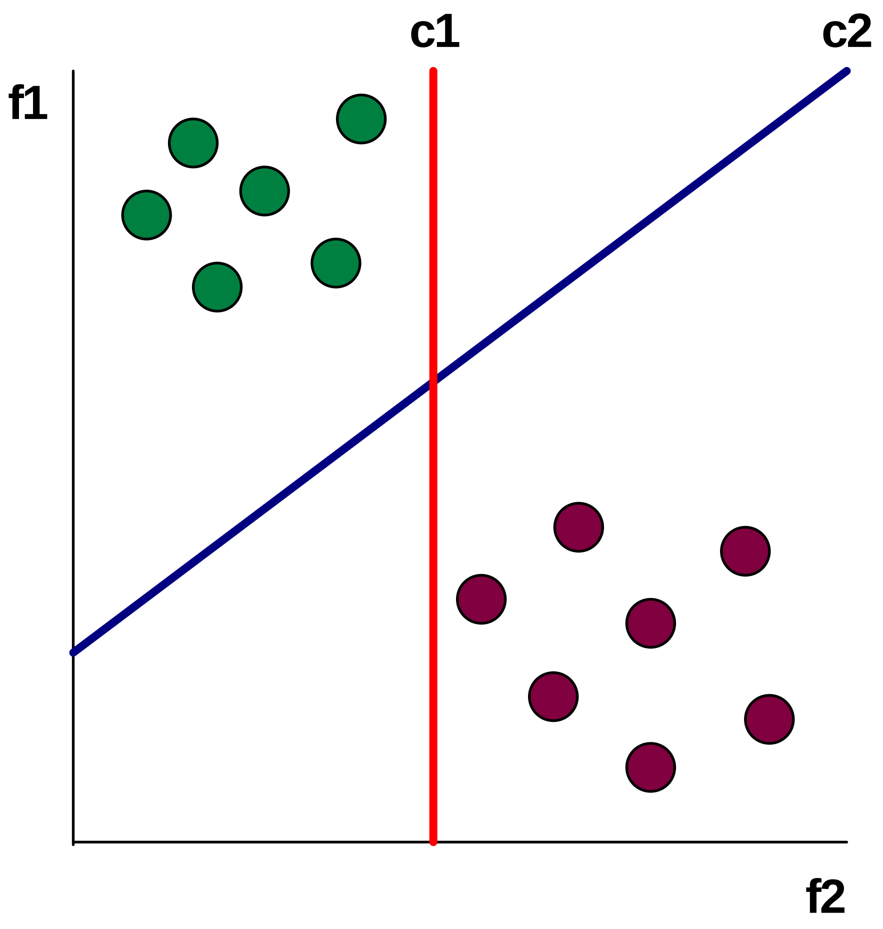
Equation 5.2. Calculating the expected value of a feature

Since , and the model only estimates the conditional probability *p(y|x)*, the probability of the context in the training data, , is used. To make the model predict the training data, a constraint is added for each feature f_i during the training of the model, such that:

Equation 5.3. Constraining the expected value to the empirical value

For any non-trivial model, the model that satisfies these constraints cannot be found analytically. In fact, there is normally even an infinite number of models. So, then the question becomes, which model do we use? Consider, for example, the classifiers in Figure 5.2, “Two competing models”. While both classifiers separate instances of both classes neatly, *c2* is the better classifier; it separates the classes with a wider margin than *c1*, and as such has more tolerance with respect to unseen instances that fall outside the current class boundaries. For instance, if an instance has a high value for *f1*, and a slightly higher value for *f2*, *c1* attribute this instance to the other class while there is no reason to believe that this is true. Or in other words, *c1* has a bias.

Figure 5.2. Two competing models



Now the idea is clear: we want a model that satisfies a set of constraints, but also has as few assumptions as possible.

Let's forget those constraints for a moment and get back to something simple, like coin flipping. We have two possible outcomes, head and tail. If we have no assumptions about the coin being biased and such, we (should) believe that the probability of getting head or tail is half-half. And what if we model dice roles? If we believe that the (cube) dice is not biased and no trickery is involved, the probability of each outcome should be 1/6th. In both cases, the outcomes have a *uniform distribution*, meaning that every outcome is equally probable. In the uniform distribution, the probability of a particular outcome is , where is the number of possible outcomes. In the uniform distribution, *uncertainty* is at its maximum. If we know that *p(tails) = 0.9*, we know pretty certain that a coin flip will result in *tails*. However, if *p(tails) = 0.5*, we are uncertain about the outcome. In fact, there is no possible distribution that has a higher uncertainty. A measure of uncertainty is entropy.

So our model should be uniform as possible, while still obeying the constraints that are imposed. We can find the most uniform model by maximizing entropy.

More to be done...

Chapter 6. Information retrieval (proposed)

Chapter 7. Part of speech tagging

Introduction

In the last episode, you have seen how n-gram language models can be used to model structure of language, purely based on words. In this chapter, we will make a further abstraction and will try to find proper *part of speech tags* (also named *morphosyntactic tags*) for words. Part of speech tags give relevant information about the role of a word in its narrow context. It may also provide information about the inflection of a word. POS tags are a valuable part of a language processing pipeline, they provide useful information to other components such as a parser or a named-entity recognizer.

There is no such thing as a standard set of part of speech tags (let's call them 'POS tags' from now on). Just like programming languages, text editors, and operating systems, the tag set that people use depends on the task at hand and taste. For our purposes, we will use the Brown tag set[1].

This is a sentence from the Brown corpus that is annotated with tags:

A/AT similar/JJ resolution/NN passed/VBD in/IN the/AT Senate/NN by/IN a/AT vote/NN of/IN 29-5/ CD ./.

The notation here is very simple: as our previous fragments of the Brown corpus the sentence is pre-tokenized. However, each word is amended by a POS tag that indicate the role of the world. For instance, the word 'a' is an *article*, 'similar' an *adjective*, and 'resolution' a *singular common noun*.

Corpora, such as the Brown corpus only provide POS tags for a small amount of sentences that occur in corpus. Being a working programmer, you will deal with new data that does not occur in the Brown corpus. Now, wouldn't it be nice to have a set of functions that could add POS tags to untagged data? Software that performs this task is called a POS tagger or morphosyntactic tagger, and this is exactly the thing we will build in this chapter.

Exercises

- In the data provided with this book, you will find the file `brown-pos-train.txt`. Open this file with a text file viewer or text editor, and look at the five first sentences. Try to find out what the tags mean using the description of the Brown tag set.

Frequency-based tagging

In one of the simplest forms of tagging, we just assign the most frequent POS tag for a token in the training data to a token in untagged data. That's right, the most frequent tag, because a token can have more than one tag. Consider the following two sentences:

- I wouldn't **trust** him.
- He put money in the family **trust**.

Both sentences contain the word 'trust'. However, 'trust' has different roles in different roles in both sentences. In the first sentence 'trust' is a verb, in the second sentence it is a noun. So, for many tokens we will have the choice of multiple tags. If we tag the token with the most frequent tag, we will frequently tag tokens incorrectly, but it is a first step.

To ease handling of tokens and tags, we will make type aliases for tokens and tags and define a new datatype for training instances, aptly named TrainingInstance:

[1]A full description of the Brown tag set can be found at: http://www.scs.leeds.ac.uk/ccalas/tagsets/brown.html

```
type Token = String
type Tag = String

data TrainingInstance = TrainingInstance Token Tag
                        deriving Show
```

The Token and Tag aliases will allow us to write clean function signatures. The TrainingInstance data type has only one constructor, TrainingInstance. The data type derives from the Show typeclass, which allows us to get a String representation of an instance[2]. We can use this constructor to create training instances:

```
*Main> TrainingInstance "the" "AT"
TrainingInstance "the" "AT"
*Main> TrainingInstance "pony" "NN"
TrainingInstance "pony" "NN"
```

Since our first POS tagger is trained purely on tokens and tags, and requires no sentential information, the corpus will be represented as a list of TrainingInstance. Since we can use the `words` function to tokenize the corpus, the task at hand is to convert a list of strings of the format "token/tag" to a list of TrainingInstance. This is done by splitting the String on the forward slash character (/). We can use the `break` function to break the string on the first element for which the supplied function is true. For instance:

```
Prelude> break (== '/') "the/AT"
("the","/AT")
```

This is a good start, we would only have to chop off the first character of the second element in the tuple. However, there is another problem: although a tag can never contain a slash, a token can. Consequently, we should break the string on the last slash, rather than the first. A cheap solution to this problem could be to reverse the string, applying `break`, and then reversing the results again. We will take a more sophisticated route, and write our own function:

```
rsplit :: Eq a => a -> [a] -> ([a], [a])
rsplit sep l = let (ps, xs, _) = rsplit_ sep l in
               (ps, xs)

rsplit_ :: Eq a => a -> [a] -> ([a], [a], Bool)
rsplit_ sep = foldr (splitFun sep) ([], [], False)
    where splitFun sep e (px, xs, True) = (e:px, xs, True)
          splitFun sep e (px, xs, False)
                   | e == sep = (px, xs, True)
                   | otherwise = (px, e:xs, False)
```

The core business happens in the `rsplit_` function, it splits a list in the part before the last instance of `sep` (the prefix) and the part after (the suffix). It does this by folding over the input list from right to left. The accumulator is a tuple that holds the prefix list, the suffix list, and a Bool indicating whether the separator was encountered. The function provided to the fold acts upon this Bool:

- If the Bool is True, the separator was seen, and the current element is added to the prefix list.
- If the Bool is False, the separator was not seen yet. If the current element is equal to the separator, the Bool is changed to True to indicate that all remaining elements should be added to the prefix list. Otherwise, the element is added to the suffix list.

`rsplit` is just a tiny wrapper around `rsplit_` that returns a binary tuple with just the prefix and suffix lists. The `rsplit` function works as intended:

```
*Main> rsplit '/' "the/AT"
("the","AT")
*Main> rsplit '/' "a/b/TEST"
```

[2]This String representation is also used by **ghci** to print the value of a TrainingInstance.

```
("a/b","TEST")
```

We are now able to get the necessary data out of a String containing a token and a tag. We can simply construct a training instance by converting the tuple:

```
toTrainingInstance :: String -> TrainingInstance
toTrainingInstance s = let (token, tag) = rsplit '/' s in
TrainingInstance token tag
```

Why not see how we are doing, and get the ten first training instances of the Brown corpus?

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> take 10 $ map toTrainingInstance $ words c
[TrainingInstance "The" "AT",TrainingInstance "Fulton" "NP",
  TrainingInstance "County" "NN",TrainingInstance "Grand" "JJ",
  TrainingInstance "Jury" "NN",TrainingInstance "said" "VBD",
  TrainingInstance "Friday" "NR",TrainingInstance "an" "AT",
  TrainingInstance "investigation" "NN",TrainingInstance "of" "IN"]
```

Alright! That's indeed our corpus in beautified format. The next step is to traverse this corpus, registering for each word with which tag it occurred (and how often). For this we write the `tokenTagFreq` function:

```
import qualified Data.List as L
import qualified Data.Map as M

tokenTagFreqs :: [TrainingInstance] -> M.Map Token (M.Map Tag Int)
tokenTagFreqs = L.foldl' countWord M.empty
    where
      countWord m (TrainingInstance token tag) =
          M.insertWith (countTag tag) token (M.singleton tag 1) m
      countTag tag _ old = M.insertWith
          (\newFreq oldFreq -> oldFreq + newFreq) tag 1 old
```

This function is very comparable to the `countElem` function we saw earlier, the primary difference being that we have to handle two levels of maps. Every Token in the first Map is associated with a value that is itself a Map that maps Tags to frequencies (Int). If we have not seen a particular Token yet, we will insert it to the map with the Token as the key, the value is a map with just one key/value: the Tag associated with the token and a frequency of one. If the Token was seen before, we will update the frequency of the associated Tag, setting it to one, if the Tag was never seen before with this token.

Let us test `tokenTagFreqs` on the first ten training instances as well:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> tokenTagFreqs $ take 10 $ map toTrainingInstance $ words c
fromList [("County",fromList [("NN",1)]),("Friday",fromList [("NR",1)]),
  ("Fulton",fromList [("NP",1)]),("Grand",fromList [("JJ",1)]),
  ("Jury",fromList [("NN",1)]),("The",fromList [("AT",1)]),
  ("an",fromList [("AT",1)]),("investigation",fromList [("NN",1)]),
  ("of",fromList [("IN",1)]),("said",fromList [("VBD",1)])]
```

It seems to work, but we cannot be sure until we have seen duplicates and ambiguous tokens. You may want to play a little with larger corpus samples or artificial training data to confirm that `tokenTagFreqs` works as intended.

The next thing we need for our first part of speech tagger is use the map defined by `tokenTagFreqs` to find the most frequent tag for a word. This is a typical mapping situation: for each key/value pair in the Map, we want to transform its value. The value was a Map, mapping Tag to Int, and we want the value to be a Tag, namely the most frequent Tag. There is also a `map` functions for Map:

```
*Main> :type Data.Map.map
Data.Map.map :: (a -> b) -> M.Map k a -> M.Map k b
```

`Data.Map.map` accepts some function to map every value in a Map to a new value. For getting the most frequent Tag, we have to fold over the inner map, storing the most frequent tag and its frequency in the accumulator. The Data.Map module provides the `foldlWithKey` to fold over keys and values:

```
*Main> :type Data.Map.foldlWithKey
Data.Map.foldlWithKey :: (b -> k -> a -> b) -> b -> M.Map k a -> b
```

This looks like the usual suspect, however, the folding function takes an additional parameter. The folding function has the current accumulator, the current key, and the associated value as its arguments. Using these building blocks, we can construct the `tokenMostFreqTag` function:

```
tokenMostFreqTag :: M.Map Token (M.Map Tag Int) -> M.Map Token Tag
tokenMostFreqTag = M.map mostFreqTag
    where
      mostFreqTag = fst . M.foldlWithKey maxTag ("NIL", 0)
      maxTag acc@(maxTag, maxFreq) tag freq
                 | freq > maxFreq = (tag, freq)
                 | otherwise = acc
```

The main function body uses `mostFreqTag` to get the most frequent tag for each token. `mostFreqTag` folds over all tokens and frequencies of a map associated with a token. The initial value of the accumulator is the dummy tag 'NIL'. The `maxTag` function that is used in the fold will replace the accumulator with the current tag and its frequency if its frequency is higher than the frequency of the tag in the accumulator. Otherwise, the tag in the accumulator is more frequent, and the accumulator is retained. After folding, we have the pair of the most frequent tag, and its frequency. We use the `fst` function to get the first element of this pair.

You can craft some examples to check whether `tokenMostFreqTag` works as intended. For example:

```
*Main> tokenMostFreqTag $ tokenTagFreqs [TrainingInstance "a" "A",
  TrainingInstance "a" "B", TrainingInstance "a" "A"]
fromList [("a","A")]
```

Combining `tokenTagFreqs` and `tokenMostFreqTag` we can make a simple function to train our first tagging model from a list of TrainingInstance:

```
trainFreqTagger :: [TrainingInstance] -> M.Map Token Tag
trainFreqTagger = tokenMostFreqTag . tokenTagFreqs
```

Next up is the actual tagger: it simply looks up a token, returning the most frequent tag of a token. Since not all tags may be known, you may want to decide how to handle unknown tags. For now, we will just return the `Maybe Tag` type, allowing us to return `Nothing` in the case we do not know how to tag a word. We will define the function `freqTagWord` as a simple wrapper around `Data.Map.lookup`:

```
freqTagWord :: M.Map Token Tag -> Token -> Maybe Tag
freqTagWord m t = M.lookup w t
```

We can now train our model from the Brown corpus, and tag some sentences:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let model = trainFreqTagger $ map toTrainingInstance $ words c
*Main> map (freqTagWord model) ["The","cat","is","on","the","mat","."]
[Just "AT",Just "NN",Just "BEZ",Just "IN",Just "AT",Just "NN",Just "."]
*Main> map (freqTagWord model) ["That's","right",",","the","mascara",
  "snake",".","Fast","and","bulbous",".","Also","a","tinned","teardrop","."]
[Just "DT+BEZ",Just "QL",Just ",",Just "AT",Just "NN",Just "NN",Just ".",
  Nothing,Just "CC",Nothing,Just ".",Just "RB",Just "AT",Nothing,
  Just "NN",Just "."]
```

Isn't that NLP for the working programmer? Not only did you learn about POS tagging, you built your own first POS tagger in just a few lines of Haskell code. In the next section we will be a bit more scientific, and focus on evaluation of taggers.

Evaluation

Now that you wrote your first tagger, the question is how well it work. Not only to show it off to your colleagues (it will do relatively well), but also to be able to see how future changes impact the performance of the tagger. To check the performance of the tagger, we will use an evaluation corpus. You should never evaluate a natural language processing component on the training data, because it is easy to perform well on seen data. Suppose that you wrote a tagger that just remembered the training corpus exactly. This tagger would tag every word correctly, but it will behave badly on unseen data.

For evaluating our taggers, we will use another set of sentences from the Brown corpus. These annotated sentences are provided in `brown-pos-test.txt`. Since file has the same format as `brown-pos-train.txt`, it can also be read as a list of TrainingInstance.

To evaluate a POS tagger, we will write a function that takes a tagging function (Word -> Maybe Tag) as its first argument and a training corpus as its second argument. It should then return a tuple with the total number of tokens in the corpus, the number of tokens that were tagged correctly, and the number of tokens for which the tagger did not provide an analysis (returned Nothing). This is the `evalTagger` function:

```
evalTagger tagFun = L.foldl' eval (0, 0, 0)
    where
      eval (n, c, u) (TrainingInstance token correctTag) =
          case tagFun token of
            Just tag -> if tag == correctTag then
                            (n+1, c+1, u)
                        else
                            (n+1, c, u)
            Nothing  -> (n+1, c, u+1)
```

The function is pretty simple, it folds over all instances in the evaluation data. The counts are incremented in the following manner:

- If the tagger returned a tag for the current token, we have two options:
  - The tagger picked the correct tag. We increment the number of tokens and the number of correct tags by one.
  - The tagger picked an incorrect tag. We only increment the number of tokens by one.
- The tagger returned no tag for the current token. We increment the number of tokens and the number of untagged tokens by one.

Time to evaluate your first tagger!

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let model = trainFreqTagger $ map toTrainingInstance $ words c
*Main> i <- IO.openFile "brown-pos-test.txt" IO.ReadMode
*Main> d <- IO.hGetContents i
*Main> evalTagger (freqTagWord model) $ map toTrainingInstance $ words d
(272901,239230,11536)
```

Those are quite impressive numbers for a first try, *239230 / 272901 * 100% = 87.66%* of the tokens were tagged and tagged correctly. Of the remaining 12.34% of the tokens, *11536 / 272901 * 100% = 4.23%* of the words were not known. This means that we tagged *239230 / (272901 - 11536) * 100% = 91.53%* of the words known to our model correctly.

To get an impression what these numbers actually mean, we will create a baseline. A baseline is a dumb model that indicates (more or less) the range we are working in. Our baseline will simply pick the most frequent tag for every token (as in, most frequent in the corpus, not for the token). We will generalize the function a bit, allowing us to specify the tag to be used:

```
baselineTagger :: Tag -> Token -> Maybe Tag
baselineTagger tag _ = Just tag
```

The most frequent tag in the Brown corpus is *NN*, the singular common noun. Let's evaluate the baseline tagger:

```
*Main> h <- IO.openFile "brown-pos-test.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> evalTagger (baselineTagger "NN") $ map toTrainingInstance $ words c
(272901,31815,0)
```

We sure do a lot better than this baseline at *31815 / 272901 * 100% = 11.66%*! What if we implement the same heuristic for unknown words in our frequency tagger? You may expect it to only correct a small proportion of unknown words, but trying never hurts. We add a function named `backOffTagger` that wraps a tagger, returning some default tag if the tagger failed to find a tag for a token:

```
backoffTagger :: (Token -> Maybe Tag) -> Tag -> Token -> Maybe Tag
backoffTagger f bt t = let pick = f t in
                       case pick of
                         Just tag -> Just tag
                         Nothing  -> Just bt
```

See how we can nicely cascade taggers by writing higher-order functions? We proceed to evaluate this tagger:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let model = trainFreqTagger $ map toTrainingInstance $ words c
*Main> i <- IO.openFile "brown-pos-test.txt" IO.ReadMode
*Main> d <- IO.hGetContents i
*Main> evalTagger (backoffTagger (freqTagWord model) "NN") $
  map toTrainingInstance $ words d
(272901,241590,0)
```

That did improve performance some. Of the 11536 tokens that we did not tag in the frequency-based tagger, we now tagged 2360 tokens correctly. This means that we tagged 20.46% of the unknown words correctly. This is almost double of the baseline, how is that possible? It turns out that of some classes of tokens, such as articles, prepositions, and tokens, you will never encounter new ones in unseen data. Unknown words are often nouns, verbs, and adjectives. Since nouns form a larger proposition of unknown words than all words, you will also get a better performance when guessing that a word is a singular common noun in unseen data.

Table 7.1, “Performance of the frequency tagger” summarizes the result so far. We have also added the score of the *oracle* this is the performance that you would attain if the tagger was omniscient. In this task, the *oracle* performs the task perfectly, but this is not true for every task.

Table 7.1. Performance of the frequency tagger

Tagger	Accuracy (%)
Baseline	11.66
Frequency-based	87.66
Frequency-based + backoff	88.53
Oracle	100.00

Transformation-based tagging

While the frequency-tagger that you developed over the last two sections was a good first attempt at POS tagging, the performance of taggers can be improved by taking context into account. To give an example, the token 'saving' is used as a verb most frequently. However, when the word is used as a noun, this can often be derived from the context, as in the following two sentences:

1. *The/AT weight/NN advantage/NN ,/, plus/CC greater/JJR durability/NN of/IN the/AT plastic/NN unit/NN ,/, yields/VBZ a/AT* ***saving/NN*** *of/IN about/RB one-fifth/NN in/IN shipping/VBG ./.*

2. *Its/PP$ elimination/NN would/MD result/VB in/IN the/AT* ***saving/NN*** *of/IN interest/ NN costs/ NNS ,/, heavy/JJ when/WRB short-term/NN money/NN rates/NNS are/BER high /JJ ,/, and/CC in/IN freedom/NN from/IN dependence/NN on/IN credit/NN which/WDT is/BEZ not/* always/RB available/JJ when/WRB needed/VBN most/RBT ./.*

In both cases, saving is preceded by an article and succeeded by a preposition. The context disambiguates what specific reading of the token 'saving' should be used.

We could manually inspect all errors in the training corpus after tagging it with the frequency-based tagger, and write rules that correct mistaggings. This has been done in the past, and can give a tremendous boost in performance. Unfortunately, finding such rules is very tedious work, and specific to one language and tag set. Fortunately, [bib-brill1992] has shown that such rules can be learnt automatically using so-called *transformation-based learning*. The learning procedure is straightforward:

1. Tag every token in the training corpus using the most frequent tag for a word.
2. Create rules from rule templates that correct incorrectly tagged words.
3. Count how many corrections were made and errors were introduced when each rule is applied to the corpus.
4. Select the best rule according to the following equation:

   Equation 7.1. Transformation rule selection criterion

5. Go to step 2, unless a threshold has been reached (e.g. rules do not give a net improvement).

The rule templates follow a very simple format. These are two examples from Brill's paper:

1. *old_tag new_tag NEXT-TAG tag*
2. *old_tag new_tag PREV-TAG to*

Two possible rules derived from these rule templates are:

1. TO IN NEXT-TAG AT
2. NN VB PREV-TAG TO

The first rule replaces the tag 'TO' (infinitival 'to') by 'IN' (preposition) if the next tag is 'AT' (article). The second rule, replaces the tag 'NN' (singular common noun) to 'VB' (verb, base) if the previous tag was 'TO' (infinitival 'to'). As you can immediately see, these are two very effective rules.

Since you already have a frequency-based tagger, you can already perform the first step of the learning procedure for transformation-based tagging. What we still need are rule templates, rule extractors, and a scoring function. For brevity, we will only focus on three tag-based templates, but after implementing the learning procedure, it should be fairly obvious how to had other contexts and integrating words in templates. The templates that we will create, will take be all variations on directly surrounding tags (previous tag, next tag, both surrounding tags). Thanks to Haskell's algebraic data types, we can easily model these templates:

```
data Replacement = Replacement Tag Tag
                   deriving (Eq, Ord, Show)
```

```
data TransformationRule =
      NextTagRule     Replacement Tag
    | PrevTagRule     Replacement Tag
    | SurroundTagRule Replacement Tag Tag
      deriving (Eq, Ord, Show)
```

To confirm that these templates are indeed working as expected, we can recreate the rules that were mentioned earlier:

```
*Main> NextTagRule (Replacement "TO" "IN") "AT"
NextTagRule (Replacement "TO" "IN") "AT"
*Main> PrevTagRule (Replacement "NN" "VB") "TO"
PrevTagRule (Replacement "NN" "VB") "TO"
```

Awesome! Now on to rule instantiation. We need to instantiate rules for tags that are incorrect, so ideally we have the corpus represented as a list of binary tuples, where the first element is the correct tag, and the second element the tag that is currently assigned by the tagger. For instance:

```
[("AT","AT"),("NN","VB"),("TO", "TO")]
```

This can simply be done by using Haskell's `zip` function, that 'zips' together two lists into one list of binary tuples:

```
*Main> :type zip
zip :: [a] -> [b] -> [(a, b)]
*Main> let correct = ["AT","NN","TO"]
*Main> let tagged = ["AT","VB","TO"]
*Main> zip correct tagged
[("AT","AT"),("NN","VB"),("TO","TO")]
```

However, using lists is not really practical in this case. By the way they are normally traversed, the current element is always the head, meaning that we do not readily have access to previous elements. But we no need to access previous elements for the PrevTagRule and SurroundTagRule templates. We can write our own function that keeps track of previous elements, but a package with such functionality, called ListZipper, is already available. After using **cabal** to install the ListZipper package, you will have access to the Data.List.Zipper module. A Zipper can be seen as a list that can be traversed in two directions. We can construct a Zipper from a list:

```
*Main> let taggingState = Data.List.Zipper.fromList $
  zip ["AT","NN","TO"]  ["AT","VB","TO"]
*Main> taggingState
Zip [] [("AT","AT"),("NN","VB"),("TO","TO")]
```

We can get the current element (the element the so-called *cursor* is pointing at) in the zipper using `Data.List.Zipper.cursor`:

```
*Main> Data.List.Zipper.cursor taggingState
("AT","AT")
```

We can move the cursor to the left (point to the previous element) with `Data.List.Zipper.left`, and to the right (point to the next element) with `Data.List.Zipper.right`:

```
*Main> Data.List.Zipper.right taggingState
Zip [("AT","AT")] [("NN","VB"),("TO","TO")]
*Main> Data.List.Zipper.cursor $ Data.List.Zipper.right taggingState
("NN","VB")
*Main> Data.List.Zipper.left $ Data.List.Zipper.right $ taggingState
Zip [] [("AT","AT"),("NN","VB"),("TO","TO")]
*Main> Data.List.Zipper.cursor $ Data.List.Zipper.left $
  Data.List.Zipper.right $ taggingState
("AT","AT")
```

This allows us to do the kind of maneuvering necessary to extract rules. The rule instantiations are modelled as functions, and are pretty simple: they just pick the information that is necessary out of their environment. We have to bit careful at the boundaries of the Zipper though: at the beginning of the Zipper only NextTagRule can extract the necessary information, and at the end of the Zipper this applies to PrevTagRule. To be able to handle such situations, we make the return type of the instantiation functions Maybe TransformationRule. Let's go through the instantiation functions one by one, starting with `instNextTagRule0` (we add the '0' suffix, since we will prettify these functions later):

```
import qualified Data.List.Zipper as Z

instNextTagRule0 :: Z.Zipper (Tag, Tag) -> Maybe TransformationRule
instNextTagRule0 z
    | Z.endp z = Nothing
    | Z.endp $ Z.right z = Nothing
    | otherwise = Just $ NextTagRule (Replacement incorrectTag correctTag) nextTag
    where (correctTag, incorrectTag) = Z.cursor z
          nextTag = snd $ Z.cursor $ Z.right z
```

When instantiating a rule from the current element in the Zipper, we have two problematic conditions to check for. The first is that the Zipper does not point to an element. This happens when we would traverse to the right when are already at the last element of the Zipper. In the second condition, we are actually at the last element of the Zipper. In this situation, we cannot extract the next Tag. For both conditions, we return Nothing. When these conditions do not hold, we can extract a NextTagRule. We do this by defining the replacement, replacing the incorrect tag by the correct one, and extracting the next tag. We can test this instantiation function, assuming that `taggingState` is defined as above:

```
*Main> instNextTagRule0 $ Data.List.Zipper.right taggingState
Just (NextTagRule (Replacement "VB" "NN") "TO")
```

The `instPrevTag0` function is almost similar, except that in the second condition returns Nothing if the current element is the first element of the Zipper. And, of course, we extract the previous tag rather than the next tag:

```
instPrevTagRule0 :: Z.Zipper (Tag, Tag) -> Maybe TransformationRule
instPrevTagRule0 z
    | Z.endp z = Nothing
    | Z.beginp z = Nothing
    | otherwise = Just $ PrevTagRule (Replacement incorrectTag correctTag) prevTag
    where (correctTag, incorrectTag) = Z.cursor z
          prevTag = snd $ Z.cursor $ Z.left z
```

Let's do a sanity check to be safe:

```
*Main> instPrevTagRule0 $ Data.List.Zipper.right taggingState
Just (PrevTagRule (Replacement "VB" "NN") "AT")
```

Finally, we write the `instSurroundTag0` function, which combines the functionality of `instNextTag0` and `instPrevTag0`:

```
instSurroundTagRule0 :: Z.Zipper (Tag, Tag) -> Maybe TransformationRule
instSurroundTagRule0 z
    | Z.endp z = Nothing
    | Z.beginp z = Nothing
    | Z.endp $ Z.right z = Nothing
    | otherwise = Just $ SurroundTagRule (Replacement incorrectTag correctTag)
                  prevTag nextTag
    where (correctTag, incorrectTag) = Z.cursor z
          prevTag = snd $ Z.cursor $ Z.left z
          nextTag = snd $ Z.cursor $ Z.right z
```

And this also works as intended:

```
*Main> instSurroundTagRule0 $ Data.List.Zipper.right taggingState
Just (SurroundTagRule (Replacement "VB" "NN") "AT" "TO")
```

We will make these functions simpler by making use of the Maybe monad. First, we define two functions to get the previous and the next element of the zipper, wrapped in Maybe. To accomplish this, we use the `safeCursor` function, which returns the element the cursor points at using Maybe. It will return value Nothing if the cursor points beyond the last element of the zipper.

```
rightCursor :: Z.Zipper a -> Maybe a
rightCursor = Z.safeCursor . Z.right

leftCursor :: Z.Zipper a -> Maybe a
leftCursor z = if Z.beginp z then
                   Nothing
               else
                   Z.safeCursor $ Z.left z
```

The `rightCursor` function is trivial. The `leftCursor` is a bit more complicated, since calling `left` on a Zipper with a cursor pointing at the first element, will return an equivalent Zipper. So, we return Nothing when we are pointing at the first element (and cannot move left).

In our previous implementations of the instantiation functions, we checked all failure conditions using guards. However, once we work with expressions evaluating to Maybe, we can use the Maybe monad instead. The Maybe monad represents computations that could fail (return Nothing), and a failure will be propagated (the monad will end in Nothing). The `return` function is used to pack the value of the final expression in a Maybe.

Using the Maybe monad, we can simplify the instantiation functions:

```
instNextTagRule :: Z.Zipper (Tag, Tag) -> Maybe TransformationRule
instNextTagRule z = do
    (_, next) <- rightCursor z
    (correct, incorrect) <- Z.safeCursor z
    return $ NextTagRule (Replacement incorrect correct) next

instPrevTagRule :: Z.Zipper (Tag, Tag) -> Maybe TransformationRule
instPrevTagRule z = do
    (_, prev)            <- leftCursor z
    (correct, incorrect) <- Z.safeCursor z
    return $ PrevTagRule (Replacement incorrect correct) prev


instSurroundTagRule :: Z.Zipper (Tag, Tag) -> Maybe TransformationRule
instSurroundTagRule z = do
    (_, next)            <- rightCursor z
    (_, prev)            <- leftCursor z
    (correct, incorrect) <- Z.safeCursor z
    return $ SurroundTagRule (Replacement incorrect correct) prev next
```

With the instantiation functions set in place, we can fold over the Zipper using `Data.List.Zipper.foldlz'`. This is a left fold with a strict accumulator. The folding function gets the accumulator as its first argument and the current Zipper (state) as its second:

```
*Main> :type Data.List.Zipper.foldlz'
Data.List.Zipper.foldlz'
  :: (b -> Z.Zipper a -> b) -> b -> Z.Zipper a -> b
```

Using this function, we write the `instRules0` function:

```
instRules0 :: [(Z.Zipper (Tag, Tag) -> Maybe TransformationRule)] ->
             Z.Zipper (Tag, Tag) -> S.Set TransformationRule
instRules0 funs = Z.foldlz' applyFuns S.empty
    where applyFuns s z
              | correct == proposed = s
              | otherwise = foldl (applyFun z) s funs
              where (correct, proposed) = Z.cursor z
                    applyFun z s f = case f z of
```

```
                                      Nothing -> s
                                      Just r -> S.insert r s
```

`instRules0` accepts a list of instantiation functions (`funs`) and a zipper, and returns a Set of instantiated rules. It folds over the zipper, applying all functions (`applyFuns`) to the current element. If the tag that is currently proposed is already correct, the Set is unchanged, because there is no transformation to be learnt. If the proposed tag differs from the correct tag, rules are instantiated by folding over the instantiation functions. Applying this to our little test data, shows that the function is operating correctly:

```
*Main> instRules0 [instNextTagRule, instPrevTagRule, instSurroundTagRule]
  taggingState
fromList [NextTagRule (Replacement "VB" "NN") "TO",
  PrevTagRule (Replacement "VB" "NN") "AT",
  SurroundTagRule (Replacement "VB" "NN") "AT" "TO"]
```

Now we have to massage the corpus and the proposed corpus to the correct format. The `initialLearningState` function extracts the list of correct tags from the corpus, and uses the word frequency tagger with 'NN' as the back-off for unknown words to get a list of proposed tags. Both lists are then zipped and the zipped list is converted to a Zipper:

```
initialLearningState :: [TrainingInstance] -> Z.Zipper (Tag, Tag)
initialLearningState train = Z.fromList $ zip (correct train) (proposed train)
    where proposed    = map tagger . trainTokens
          correct     = map (\(TrainingInstance _ tag) -> tag)
          tagger      = DM.fromJust . backoffTagger (freqTagWord model) "NN"
          trainTokens = map (\(TrainingInstance token _) -> token)
          model       = trainFreqTagger train
```

Now we can use this function to create the initial state for the transformation-based learner, and extract all possible transformation rules:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let proposedRules = instRules0 [instNextTagRule, instPrevTagRule, instSurroundTagRule] $
  initialLearningState $ map toTrainingInstance $ words c
*Main> Data.Set.size proposedRules
18992
```

Good, this allows us to find all possible correction rules. We could now calculate the scores for all rules. But the rule selection can be made somewhat more efficient. Each rule instantiation was actually an instance of a correct rule application. If we register the correct counts, we can start with scoring the most promising rules first. Once the score of a rule is higher than the correct count of the next rule, we have found the most effective rule. We can modify `instRules0` to do this:

```
instRules :: [(Z.Zipper (Tag, Tag) -> Maybe TransformationRule)] ->
             Z.Zipper (Tag, Tag) -> M.Map TransformationRule Int
instRules funs = Z.foldlz' applyFuns M.empty
    where applyFuns m z
              | correct == proposed = m
              | otherwise = foldl (applyFun z) m funs
              where (correct, proposed) = Z.cursor z
                    applyFun z m f = case f z of
                                       Nothing -> m
                                       Just r -> M.insertWith' (+) r 1 m
```

We then use this frequency map to create a list of rules sorted by frequency:

```
sortRules :: M.Map TransformationRule Int -> [(TransformationRule, Int)]
sortRules = L.sortBy (\(_,a) (_,b) -> compare b a) . M.toList
```

`Data.List.sortBy` sorts a list according to some comparison function. We use the stock `compare` function:

```
Prelude> :type compare
compare :: (Ord a) => a -> a -> Ordering
Prelude> compare 1 2
LT
Prelude> compare 2 2
EQ
Prelude> compare 2 1
GT
```

The compare function returns a value of the Ordering type, returning LT, EQ, GT, depending on whether the first argument is smaller than, equal to, or larger than the second argument. In `sortRules`, we use a lambda to get the second tuple element (representing a frequency). We also swap the arguments to `compare` function to get a reverse ordering, making larger elements come first.

Let's get some immediate gratification by extracting the ten rules with the most corrections:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let proposedRules = instRules [instNextTagRule, instPrevTagRule, instSurroundTagRule] $
  initialLearningState $ map toTrainingInstance $ words c
*Main> take 10 $ sortRules proposedRules
[(NextTagRule (Replacement "TO" "IN") "AT",3471),
  (PrevTagRule (Replacement "TO" "IN") "NN",2459),
  (PrevTagRule (Replacement "NN" "VB") "TO",1690),
  (NextTagRule (Replacement "VBN" "VBD") "AT",1154),
  (PrevTagRule (Replacement "TO" "IN") "VBN",1088),
  (SurroundTagRule (Replacement "TO" "IN") "NN" "AT",1034),
  (NextTagRule (Replacement "TO" "IN") "NN",994),
  (NextTagRule (Replacement "NN" "VB") "AT",846),
  (PrevTagRule (Replacement "TO" "IN") "JJ",813),
  (NextTagRule (Replacement "VBD" "VBN") "IN",761)]
```

The next thing we need to be able to do is to evaluate a rule. However, we currently have no way to see whether a rule applies. To this end, we write the `ruleApplication` function, this function returns the replacement tag wrapped in Maybe's Just constructor. If the rule could not be applied to the current corpus element, Nothing is returned:

```
ruleApplication :: TransformationRule -> Z.Zipper (Tag, Tag) -> Maybe Tag
ruleApplication (NextTagRule (Replacement old new) next) z = do
  (_, proposed)     <- Z.safeCursor z
  (_, nextProposed) <- rightCursor z
  if proposed == old && nextProposed == next then Just new else Nothing
ruleApplication (PrevTagRule (Replacement old new) prev) z = do
  (_, proposed)     <- Z.safeCursor z
  (_, prevProposed) <- leftCursor z
  if proposed == old && prevProposed == prev then Just new else Nothing
ruleApplication (SurroundTagRule (Replacement old new) prev next) z = do
  (_, proposed)     <- Z.safeCursor z
  (_, nextProposed) <- rightCursor z
  (_, prevProposed) <- leftCursor z
  if proposed == old && prevProposed == prev &&
      nextProposed == next then Just new else Nothing
```

This function closely matches the instantiation functions, except that we check whether the context corresponds to the context specified by the rule. We can then apply a rule to every element in the Zipper, checking whether the change was correct when a Just value is returned:

```
scoreRule :: TransformationRule -> Z.Zipper (Tag, Tag) -> Int
scoreRule r z = nCorrect - nIncorrect
    where (nCorrect, nIncorrect) = scoreRule_ r z

scoreRule_ :: TransformationRule -> Z.Zipper (Tag, Tag) -> (Int, Int)
scoreRule_ r = Z.foldlz' (scoreElem r) (0, 0)
    where scoreElem r s@(nCorrect, nIncorrect) z =
              case ruleApplication r z of
                Just tag -> if tag == correct then
                                (nCorrect + 1, nIncorrect)
                            else
                                (nCorrect, nIncorrect + 1)
                Nothing  -> s
              where (correct, _) = Z.cursor z
```

The main action happens in the `scoreRule_` function. It traverses the zipper, applying the rule to each element. If the rule applies to an element, we check whether the application corrected the tag, and update the counts (`nCorrect` and `nIncorrect`) accordingly. If the rule does not apply, we keep the counts as they are. `scoreRule` is just a simple wrapper around `scoreRule_`, and subtracts the number of errors introduced from the number of corrections. You can try to apply this function to some rules, for instance the best rule of the initial ranking:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let learningState = initialLearningState $ map toTrainingInstance $ words c
*Main> let proposedRules = instRules
  [instNextTagRule, instPrevTagRule, instSurroundTagRule] learningState
*Main> head $ sortRules proposedRules
(NextTagRule (Replacement "TO" "IN") "AT",3471)
*Main> let (firstRule, _) = head $ sortRules proposedRules
*Main> scoreRule firstRule learningState
3470
```

So, given a set of rules, we have to select the best rule. We know that we have found the best rule when its score is higher than the number of corrections of the next rule in the sorted rule list. The `selectRule_` function does exactly this:

```
selectRule :: [(TransformationRule, Int)] -> Z.Zipper (Tag, Tag) ->
              (TransformationRule, Int)
selectRule ((rule, _):xs) z = selectRule_ xs z (rule, (scoreRule rule z))

selectRule_ :: [(TransformationRule, Int)] -> Z.Zipper (Tag, Tag) ->
              (TransformationRule, Int) -> (TransformationRule, Int)
selectRule_ [] _ best = best
selectRule_ ((rule, correct):xs) z best@(bestRule, bestScore) =
    if bestScore >= correct then
        best
    else
        if bestScore >= score then
            selectRule_ xs z best
        else
            selectRule_ xs z (rule, score)
    where score = scoreRule rule z
```

First we check whether the stopping condition is reached (`bestScore > correct`). If this is not the case, we have to decide whether the currently best rule is better than the current rule (`bestScore >= score`). If this is is not the case, the current rule becomes the best rule. Let us use this function to select the best rule:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let learningState = initialLearningState $ map toTrainingInstance $ words c
*Main> let proposedRules = instRules
  [instNextTagRule, instPrevTagRule, instSurroundTagRule] learningState
*Main> selectRule (sortRules proposedRules) learningState
(NextTagRule (Replacement "TO" "IN") "AT",3470)
```

Excellent! Our learner is now almost done! Once we have selected a rule, we need to update the training state, and then we can rinse and repeat until we are happy with the list of rules. First, we will make the `updateState` function to update the learning state:

```
updateState :: TransformationRule -> Z.Zipper (Tag, Tag) ->
               Z.Zipper (Tag, Tag)
updateState r = Z.fromList . reverse . Z.foldlz' (update r) []
    where update r xs z =
              case ruleApplication r z of
                Just tag -> (correct, tag):xs
                Nothing  -> e:xs
              where e@(correct, _) =  Z.cursor z
```

The updated state is created by copying the old state using a fold, replacing the proposed tag if a rule is applicable (returns Just tag). We used a strict left-fold for efficiency. Since we are building a list, the consequency is that we construct the list in reverse order. We then reverse the list, and construct a Zipper from this list. The next function, `transformationRules`, constructs the list of transformations:

```
transformationRules :: [(Z.Zipper (Tag, Tag) -> Maybe TransformationRule)] ->
                       Z.Zipper (Tag, Tag) -> [TransformationRule]
transformationRules funs state = bestRule:(transformationRules funs nextState)
    where (bestRule, _) = selectRule (sortRules $ instRules funs state) state
          nextState     = updateState bestRule state
```

This function is fairly simple: during each recursion we find the next best rule, and update the state accordingly. The rule becomes the head of the list that we are returning, and we call `transformationRules` recursively to construct the tail of the list. We have now completed our transformation-based learner! Time to extract some rules:

```
*Main> h <- IO.openFile "brown-pos-train.txt" IO.ReadMode
*Main> c <- IO.hGetContents h
*Main> let learningState = initialLearningState $ map toTrainingInstance $ words c
*Main> take 10 $ transformationRules [instNextTagRule, instPrevTagRule, instSurroundTagRule]
  learningState
[NextTagRule (Replacement "TO" "IN") "AT",
  PrevTagRule (Replacement "NN" "VB") "TO",
  NextTagRule (Replacement "TO" "IN") "NP",
  PrevTagRule (Replacement "VBN" "VBD") "PPS",
  PrevTagRule (Replacement "NN" "VB") "MD",
  NextTagRule (Replacement "TO" "IN") "PP$",
  PrevTagRule (Replacement "VBN" "VBD") "NP",
  PrevTagRule (Replacement "PPS" "PPO") "VB",
  NextTagRule (Replacement "TO" "IN") "JJ",
  NextTagRule (Replacement "TO" "IN") "NNS"]
```

Since we haven't optimized our implementation for instructional purposes, extracting the ten most effective rules can take a while. It is best to store the resulting the source file if you do not want to repeat this step:

```
tenBestRules :: [TransformationRule]
tenBestRules = [NextTagRule (Replacement "TO" "IN") "AT",
                PrevTagRule (Replacement "NN" "VB") "TO",
                NextTagRule (Replacement "TO" "IN") "NP",
                PrevTagRule (Replacement "VBN" "VBD") "PPS",
                PrevTagRule (Replacement "NN" "VB") "MD",
                NextTagRule (Replacement "TO" "IN") "PP$",
                PrevTagRule (Replacement "VBN" "VBD") "NP",
                PrevTagRule (Replacement "PPS" "PPO") "VB",
                NextTagRule (Replacement "TO" "IN") "JJ",
                NextTagRule (Replacement "TO" "IN") "NNS"]
```

The tagger itself recursively applies every rule to the proposed tags.

Exercises

1. Add two additional types of rules:

- PrevOneOrTwoTagRule: this rule is triggered when one of the last or second to last tags corresponds to the specified tag.

- PrevOneOrTwoTagRule: this rule is triggered when one of the two next tags corresponds to the specified tag.

Extract the ten best rules, adding these rule types.

2. Modify the examples so that rules can condition on words as well. Implement three rule types, CurWordRule, PrevWordRule, NextWordRule, that condition respectively on the current, previous and next word.

   Extract the ten best rules, adding these rule types.

Bibliography

[bib-brill1992] *A simple rule-based part of speech tagger*. E. Brill. 1992. Association for Computational Linguistics.

Chapter 8. Regular languages (proposed)

Chapter 9. Context-free grammars (Proposed)

Chapter 10. Performance and efficiency (proposed)