Chapter 5

Recursion and Induction

5.1 Introduction

In this set of lectures, I will talk about *recursive programming*, a programming technique you have seen before. I will introduce a style of programming, called *Functional Programming*, that is especially suited for describing recursion in computation and data structure. Functional programs are often significantly more compact, and easier to design and understand, than their imperative counterparts. I will show why induction is an essential tool in designing functional programs.

Haskell I will use a functional programming language, called `Haskell`. What follows is a very very small subset of Haskell; you should consult the references given at the end of this document for further details. A very good source is Thompson [51] which covers this material with careful attention to problems that students typically face. Another very good source is Richards [43], whose lecture slides are available online. The Haskell manual is available online at [21]; you should consult it as a reference, particularly its Prelude (haskell98-report/standard-prelude.html) for definitions of many built-in functions. However, the manual is not a good source for learning programming. I would recommend the book by Bird [6], which teaches a good deal of programming methodology. Unfortunately, the book does not quite use Haskell syntax, but a syntax quite close to it. Another good source is *A Gentle Introduction to Haskell* [22] which covers all the material taught here, and more, in its first 20 pages.

5.2 Preliminaries

5.2.1 Running Haskell programs from command line

The Haskell compiler is installed on all Sun and Linux machines in this department. To enter an interactive session for Haskell, type

```
ghci
```

The machine responds with something like

```
Prelude>
```

At this point, whenever it displays `zzz>` for some `zzz`, the machine is waiting for some response from you. You may type an expression and have its value displayed on your terminal, as follows.

```
Prelude> 3+4
7
Prelude> 2^15
32768
```

5.2.2 Loading Program Files

Typically, Haskell program files have the suffix `hs`. I load a file `Utilities.hs`, which I have stored in a directory called `haskell.dir`. My input and machine's output appear below.

```
Prelude> :l haskell.dir/Utilities.hs

[1 of 1] Compiling Utilities  ( haskell.dir/Utilities.hs, interpreted )
Ok, modules loaded: Utilities.
*Utilities Data.Char>
```

I have a program in that file to sort a list of numbers. So, now I may write

```
*Utilities Data.Char> sort[7, 5, 1, 9]
[1,5,7,9]
```

Let me create a file `337.hs` in which I will load all the definitions in this note. Each time you change a file, by adding or modifying definitions, you have to reload the file into ghci (a quick way to do this is to use the ghci command `:r` which reloads the last loaded file).

5.2.3 Comments

Any string following `--` in a line is considered a comment. So, you may write in a command line:

```
Prelude> 2^15   -- This is 2 raised to 15
```

or, in the text of a program

```
-- I am now going to write a function called "power."
-- The function is defined as follows:
--  It has 2 arguments and it returns
--  the first argument raised to the second argument.
```

There is a different way to write comments in a program that works better for longer comments, like the four lines I have written above: you can enclose a region within `{-` and `-}` to make the region a comment.

```
{- I am now going to write a function called "power."
   The function is defined as follows:
    It has 2 arguments and it returns
    the first argument raised to the second argument. -}
```

I prefer to put the end of the comment symbol, `-}`, in a line by itself.

5.2.4 Program Layout

Haskell uses line indentations in the program to delineate the scope of definitions. A definition is ended by a piece of text that is to the left (columnwise) of the start of its definition. Thus,

```
chCase c                                  -- change case
 | upper c   = uplow c
 | otherwise = lowup c
```

and

```
ch1Case c                                  -- change case
 | upper c   = uplow c
   | otherwise          =
 lowup c
```

are fine. But,

```
ch2Case c                                  -- change case
 | upper c   = uplow c
 | otherwise          =
lowup c
```

is not. The line `lowup c` is taken to be the start of another definition. In the last case, you will get an error message like

```
ERROR "337.hs":81 - Syntax error in expression (unexpected `;',
possibly due to bad layout)
```

The semicolon (`;`) plays an important role; it closes off a definition (implicitly, even if you have not used it). That is why you see `unexpected ';'` in the error message.

5.3 Primitive Data Types

Haskell has a number of built-in data types; we will use only integer (called *Int*), boolean (called *Bool*), character (called *Char*) and string (called *String*) types[1].

[1] Haskell supports two kinds of integers, `Integer` and `Int` data types. We will use only `Int`.

5.3.1 Integer

You can use the traditional integer constants and the usual operators for addition (`+`), subtraction (`-`) and multiplication (`*`). Unary minus sign is the usual `-`, but enclose a negative number within parentheses, as in `(-2)`; I will tell you why later. Division over integers is written in infix as `` `div` `` and it returns only the integer part of the quotient. (`` ` `` is the backquote symbol, usually it is the leftmost key in the top row of your keyboard.) For division over negative integers, the following rules are used: for positive x and y, `(-x) `div` (-y)` $=$ `x `div` y`, and `(-x) `div` y` $=$ `x `div` (-y)` $= -\lceil$`x/y`$\rceil$. Thus, `5 `div` 3` is `1` and `(-5) `div` 3` is `-2`. Exponentiation is the infix operator `^` so that `2^15` is `32768`. The remainder after division is given by the infix operator `` `rem` `` and the infix operator `` `mod` `` is for modulo; `x `mod` y` returns a value between 0 and $y - 1$, for positive y. Thus, `(-2) `rem` 3` is -2 and `(-2) `mod` 3` is 1.

Two other useful functions are `even` and `odd`, which return the appropriate boolean results about their integer arguments. Functions `max` and `min` take two arguments each and return the maximum and the minimum values, respectively. It is possible to write a function name followed by its arguments without any parentheses, as shown below; parentheses are needed only to enforce an evaluation order.

```
Prelude> max 2 5
5
```

The arithmetic relations are `< <= == /= > >=`. Each of these is a binary operator that returns a boolean result, `True` or `False`. Note that equality operator is written as `==` and inequality as `/=`. Unlike in C++, `3 + (5 >= 2)` is not a valid expression; Haskell does not specify how to add an integer to a boolean.

5.3.2 Boolean

There are the two boolean constants, written as `True` and `False`. The boolean operators are:

```
not -- for negation
&&  -- for and
||  -- for or
==  -- for equivalence
/=  -- for inequivalence (also called "exclusive or")
```

Here is a short session with ghci.

```
Prelude> (3 > 5) || (5 > 3)
True
Prelude> (3 > 3) || (3 < 3)
False
```

```
Prelude> (2 `mod` (-3)) == ((-2) `mod` 3)
False
Prelude> even 3 || odd 3
True
```

5.3.3 Character and String

A character is enclosed within single quotes and a string is enclosed within double quotes.

```
Prelude> 'a'
'a'
Prelude> "a b c"
"a b c"
Prelude> "a, b, c"
"a, b, c"
```

You can compare characters using arithmetic relations; the letters (characters in the Roman alphabet) are ordered in the usual fashion with the uppercase letters *smaller* than the corresponding lowercase letters. The expected ordering applies to the digits as well.

```
Prelude> 'a' < 'b'
True
Prelude> 'A' < 'a'
True
Prelude> '3' < '5'
True
```

There are two functions defined on characters, `ord` and `chr`. Function `ord(c)` returns the value of the character `c` in the internal coding table; it is a number between 0 and 255. Function `chr` converts a number between 0 and 255 to the corresponding character. Therefore, `chr(ord(c))=c`, for all characters `c`, and `ord(chr(i))=i`, for all `i`, $0 \leq$ `i` $<$ `256`. Note that all digits in the order `'0'` through `'9'`, all lowercase letters 'a' through 'z' and uppercase letters 'A' through 'Z' are contiguous in the table. The uppercase letters have smaller *ord* values than the lowercase ones.

To use `ord` and `chr` you first have to write the following command in the command line.

```
import Data.Char
```

The machine responds by

```
Prelude Data.Char>
```

```
Prelude Data.Char> ord('a')
97
Prelude Data.Char> chr(97)
'a'
Prelude Data.Char> ord(chr(103))
103
Prelude Data.Char> chr(255)
'\255'
Prelude Data.Char> (ord '9')- (ord '0')
9
Prelude Data.Char> (ord 'a')- (ord 'A')
32
```

A string is a *list* of characters; all the rules about lists, described later, apply to strings.

5.4 Writing Function Definitions

We cannot compute much by working with constants alone. We need to be able to define functions. The functions cannot be defined by interactive input. We need to keep a file in which we list all the definitions and load that file.

5.4.1 Function Parameters and Binding

Consider the expression

```
f 1+1
```

where `f` is a function of one argument. In normal mathematics, this will be an invalid expression, and if forced, you will interpret it as `f(1+1)`. In Haskell, this is a valid expression and it stands for `f(1)+1`. I give the binding rules below.

Infix and Prefix Operators I use the term *operator* to mean a function of two arguments. An *infix operator*, such as `+`, is written between its two arguments, whereas a prefix operator, such as `max`, precedes its arguments. You can convert an infix operator to a prefix operator by putting parentheses around the function name (or symbol). Thus, `(+) x y` is the same as `x + y`. You can convert from prefix to infix by putting backquotes around an operator, so `div 5 3` is the same as `5 ‘div‘ 3`.

Most built-in binary operators in Haskell that do not begin with a letter, such as `+`, `*`, `&&`, and `||`, are infix; `max`, `min`, `rem`, `div`, and `mod` are prefix. □

An expression consists of functions, binary infix operators and operands as in

```
-2 + sqr 9 + min 2 7 - 3
```

Here the first minus (called unary minus) is a prefix operator, `sqr` is a function of one argument, `+` is a binary operator, `min` is a function of two arguments, and the last minus is a binary infix operator.

Functions bind more tightly than infix operators. Function arguments are written immediately following the function name, and the right number of arguments are used up for each function, e.g., one for `sqr` and two for `min`. No parentheses are needed unless your arguments are themselves expressions. So, for a function `g` of two arguments, `g x y z` stands for `(g x y) z`. If you write `g f x y`, it will be interpreted as `(g f x) y`; so, if you have in mind the expression `g(f(x),y)`, write it as `g (f x) y` or `g(f(x),y)`. Now, `sqr 9 + min 2 7 - 3` is `(sqr 9) + (min 2 7) - 3`. As a good programming practice, do not ever write `f 1+1`; make your intentions clear by using parentheses, writing `(f 1)+1` or `f(1+1)`.

How do we read `sqr 9 + min 2 7` $\times$ `max 2 7`? After functions are bound to their arguments, we get `(sqr 9) + (min 2 7)` $\times$ `(max 2 7)`. That is, we are left with operators only, and the operators bind according to their binding powers. Since $\times$ has higher binding power than $+$, the expression is read as `(sqr 9) + ((min 2 7)` $\times$ `(max 2 7))`.

Operators of equal binding power usually associate to the left; so, `5 - 3 - 2` is `(5 - 3) - 2`, but this is not always true. Operators in Haskell are either (1) associative, so that the order does not matter, (2) left associative, as in binary minus shown above, or (3) right associative, as in `2 ^ 3 ^ 5`, which is `2 ^ (3 ^ 5)`. When in doubt, parenthesize.

In this connection, unary minus, as in `-2`, is particularly problematic. If you would like to apply `inc` to `-2`, don't write

```
inc -2
```

This will be interpreted as `(inc) - (2)`; you will get an error message. Write `inc (-2)`. And `-5 ‘div‘ 3` is `-(5 ‘div‘ 3)` which is`-1`, but `(-5) ‘div‘ 3` is `-2`.

Exercise 53

What is `max 2 3 + min 2 3`? □

Note on Terminology In computing, it is customary to say that a function "takes an argument" and "computes (or returns) a value". A function, being a concept, and not an artifact, cannot "do" anything; it simply has arguments and it has a value for each set of arguments. Yet, the computing terminology is so prevalent that I will use these phrases without apology in these notes.

5.4.2 Examples of Function Definitions

In its simplest form, a function is defined by: (1) writing the function name, (2) followed by its arguments, (3) then a "=", (4) followed by the body of the function definition.

Here are some simple function definitions. Note that I do not put any parentheses around the arguments, they are simply written in order and parentheses are put only to avoid ambiguity. We will discuss this matter in some detail later, in Section 5.4.1 (page 124).

Note: Parameters and arguments I will use these two terms synonymously. □

```
inc x = x+1                              -- increment x
imply p q = not p || q                   -- boolean implication
digit c =  ('0' <= c) && (c <= '9')      -- is c a digit?
```

We test some of our definitions now.

```
*Main> :l Teaching.dir/337.dir/HaskellFiles.dir/337.hs

*Main> inc 5
6
*Main> imply True False
False
*Main> digit '6'
True
*Main> digit 'a'
False
*Main> digit(chr(inc (ord '8')))
True
*Main> digit(chr(inc (ord '9')))
False
```

We can use other function names in a function definition.

We can define variables in the same way we define functions; a variable is a function without arguments. For example,

```
offset  = (ord 'a') - (ord 'A')
```

Unlike a variable in C++, this variable's value does not change during the program execution; we are really giving a name to a constant expression so that we can use this name for easy reference later.

Exercise 54

1. Write a function to test if its argument is a lowercase letter; write another to test if its argument is an uppercase letter.

2. Write a function to test if its argument, an integer, is divisible by 6.

3. Write a function whose argument is an uppercase letter and whose value is the corresponding lowercase letter.

4. Define a function whose argument is a digit, 0 through 9, and whose value is the corresponding character `'0'` through `'9'`.

5. Define a function `max3` whose arguments are three integers and whose value is their maximum. □

5.4.3 Conditionals

In traditional imperative programming, we use `if-then-else` to test some condition (i.e., a predicate) and perform calculations based on the test. Haskell also provides an `if-then-else` construct, but it is often more convenient to use a *conditional equation*, as shown below. The following function computes the absolute value of its integer argument.

```
absolute x
 | x >= 0 = x
 | x < 0  = -x
```

The entire definition is a *conditional equation* and it consists of two *clauses*. A clause starts with a bar (|), followed by a predicate (called a *guard*), an equals sign (=) and the expression denoting the function value for this case. The guards are evaluated in the order in which they are written (from top to bottom), and for the first guard that is true, its corresponding expression is evaluated. So, if you put `x <= 0` as the second guard in the example above, it will work too, but when `x = 0` the expression in the first equation will be evaluated and the result returned.

You can write `otherwise` for a guard, denoting a predicate that holds when none of the other guards hold. An `otherwise` guard appears only in the last equation. The same effect is achieved by writing `True` for the guard in the last equation. If no guard is `True` in a conditional equation, you will get a run-tme error message.

Given below is a function that converts the argument letter from upper to lowercase, or lower to uppercase, as is appropriate. Assume that we have already written two functions, `uplow` (to convert from upper to lowercase), `lowup` (to convert from lower to uppercase), and a function `upper` whose value is `True` iff its argument is an uppercase letter.

```
chCase c                  -- change case
 | upper c   = uplow c
 | otherwise = lowup c
```

The test with ghci gives:

```
*Main> chCase 'q'
'Q'
*Main> chCase 'Q'
'q'
```

Exercise 55

1. Define a function that returns `-1`, `0` or `+1`, depending on whether the argument is negative, zero or positive.

2. Define a function that takes three integer arguments, p, q and r. If these arguments are the lengths of the sides of a triangle, the function value is `True`; otherwise, it is `False`. Recall from geometry that every pair of values from p, q and r must sum to a value greater than the third one for these numbers to be the lengths of the sides of a triangle.

```
max3 p q r = max p (max q r)
triangle p q r = (p+q+r) > (2* (max3 p q r))
```

5.4.4 The `where` Clause

The following function has three arguments, x, y and z, and it determines if $x^2 + y^2 = z^2$.

```
pythagoras x y z = (x*x) + (y*y) == (z*z)
```

The definition would be simpler to read in the following form

```
pythagoras x y z = sqx + sqy == sqz
                   where
                    sqx = x*x
                    sqy = y*y
                    sqz = z*z
```

The `where` construct permits *local definitions*, i.e., defining variables (and functions) within a function definition. The variables `sqx`, `sqy` and `sqz` are undefined outside this definition.

We can do this example by using a local function to define squaring.

```
pythagoras x y z = sq x + sq y == sq z
                   where
                    sq p = p*p
```

5.4.5 Pattern Matching

Previously, we wrote a function like

```
imply p q = not p || q
```

as a single equation. We can also write it in the following form in which there are two equations.

```
imply False q = True
imply True  q = q
```

Observe that the equations use constants in the left side; these constants are called *literal parameters*. During function evaluation with a specific argument —say, `False True`— each of the equations are checked from top to bottom to find the first one where the given arguments match the *pattern* of the equation. For `imply False True`, the pattern given in the first equation matches, with `False` matching `False` and `q` matching `True`.

We can write an even more elaborate definition of `imply`:

```
imply False False = True
imply False True  = True
imply True  False = False
imply True  True  = True
```

The function evaluation is simply a table lookup in this case, proceeding sequentially from top to bottom.

Pattern matching has two important effects: (1) it is a convenient way of doing case discrimination without writing a sphagetti of if-then-else statements, and (2) it *binds* names to formal parameter values, i.e., assigns names to components of the data structure —`q` in the first example— which may be used in the function definition in the right side.

Pattern matching on integers can use simple arithmetic expressions, as shown below in the definition of the *successor* function.

```
suc 0 = 1
suc (n+1) = (suc n)+1
```

Asked to evaluate `suc 3`, the pattern in the second equation is found to match —with `n = 2`— and therefore, evaluation of `(suc 3)` is reduced to the evaluation of `(suc 2) + 1`.

Pattern matching can be applied in elaborate fashions, as we shall see later.

5.5 Recursive Programming

Recursive programming is closely tied to *problem decomposition*. In program design, it is common to divide a problem into a number of subproblems where each subproblem is easier to solve by some measure, and the solutions of the subproblems can be combined to yield a solution to the original problem. A subproblem is easier if its solution is known, or if it is an instance of the original problem, but over a smaller data set. For instance, if you have to sum twenty numbers, you may divide the task into four subtasks of summing five numbers each, and then add the four results. A different decomposition may (1) scan the numbers, putting negative numbers in one subset, discarding zeros and putting positive numbers in another subset, (2) sum the positive and negative subsets individually, and (3) add the two answers. In this case, the subproblem (1) is different in kind from the original problem.

In recursive programming, typically, a problem is decomposed into subproblems of the same kind, and we apply the same solution procedure to each of the

subproblems, further subdividing them. A recursive program has to specify the solutions for the very smallest cases, those which cannot be decomposed any further.

The theoretical justification of recursive programming is *mathematical induction*. In fact, recursion and induction are so closely linked that they are often mentioned in the same breath (see the title of this chapter); I believe we should have used a single term for this concept.

5.5.1 Computing Powers of 2

Compute $2^{\texttt{n}}$, for $\texttt{n} \geq 0$, using only doubling. As in typical induction, we consider two cases, a base value of `n` and the general case where `n` has larger values. Let us pick the base value of `n` to be `0`; then the function value is `1`. For `n+1` the function value is double the function value of `n`.

```
power2 0     = 1
power2 (n+1) = 2 * (power2 n)
```

How does the computer evaluate a call like `power2 3`? Here is a very rough sketch. The interpreter has an expression to evaluate at any time. It picks an operand (a subexpression) to reduce. If that operand is a constant, there is nothing to reduce. Otherwise, it has to compute a value by applying the definitions of the functions (operators) used in that expression. This is how the evaluation of such an operand proceeds. The evaluator matches the pattern in each equation of the appropriate function until a matching pattern is found. Then it replaces the matched portion with the right side of that equation. Let us see how it evaluates `power2 3`.

```
   power2 3
=  2 * (power2 2)             -- apply function definition on 3
=  2 * (2 * (power2 1))       -- apply function definition on 2
=  2 * (2 * (2 * (power2 0))) -- apply function definition on 1
=  2 * (2 * (2 * (1)))        -- apply function definition on 0
=  2 * (2 * (2))              -- apply definition of *
=  2 * (4)                    -- apply definition of *
=  8                          -- apply definition of *
```

What is important to note is that each recursive call is made to a strictly *smaller* argument, and there is a *smallest* argument for which the function value is explicitly specified. In this case, numbers are compared by their magnitudes, and the smallest number is `0`. You will get an error in evaluating `power2 (-1)`. We will see more general recursive schemes in which there may be several *base* cases, and the call structure is more elaborate, but the simple scheme described here, called *primitive recursion*, is applicable in a large number of cases.

5.5.2 Counting the 1s in a Binary Expansion

Next, let us program a function whose value is the number of 1s in the binary expansion of its argument, where we assume that the argument is a natural number. Imagine scanning the binary expansion of a number from right to left (i.e., from the lower order to the higher order bits) starting at the lowest bit; if the current bit is 0, then we ignore it and move to the next higher bit, and if it is 1, then we add 1 to a running count (which is initially 0) and move to the next higher bit. Checking the lowest bit can be accomplished by the functions `even` and `odd`. Each successive bit can be accessed via integer division by 2 (right shift).

```
count 0 = 0
count n
 | even n = count (n `div` 2)
 | odd  n = count (n `div` 2) + 1
```

Note on pattern matching It would have been nice if we could have written the second equation as follows.

```
count (2*t)     = count t
count (2*t + 1) = (count t) + 1
```

Unfortunately, Haskell does not allow such pattern matching.

5.5.3 Multiplication via Addition

Let us now implement multiplication using only addition. We make use of the identity $x * y = x * (y - 1) + x$. We require $y \geq 0$.

```
mlt x 0 = 0
mlt x y = (mlt x (y-1)) + x
```

The recursive call is made to a strictly smaller argument in each case. There are two arguments which are both numbers, and the second number is strictly decreasing in each call. The smallest value of the arguments is attained when the second argument is 0.

The multiplication algorithm has a running time roughly proportional to the magnitude of `y`, because each call decreases `y` by 1. We now present a far better algorithm. You should study it carefully because it introduces an important concept, *generalizing the function*. The idea is that we write a function to calculate something more general, and then we call this function with a restricted set of arguments to calculate our desired answer. Let us write a function, `quickMlt`, that computes `x*y + z` over its three arguments. We can then define

```
mlt x y = quickMlt x y 0
```

The reason we define `quickMlt` is that it is more efficient to compute than `mlt` defined earlier. We will use the following result from arithmetic.

$$x \times (2 \times t) + z \;=\; (2 \times x) \times t + z$$
$$x \times (2 \times t + 1) + z \;=\; (2 \times x) \times t + (x + z)$$

The resulting program is:

```
quickMlt x 0 z = z
quickMlt x  y  z
 | even y = quickMlt (2 * x ) (y `div` 2) z
 | odd  y = quickMlt (2 * x ) (y `div` 2) (x + z)
```

In each case, again the second argument is strictly decreasing. In fact, it is being halved, so the running time is proportional to $\log y$.

Exercise 56

Extend `quickMlt` to operate over arbitrary y, positive, negative and zero. □

Exercise 57

Use the strategy shown for multiplication to compute x^y. I suggest that you compute the more general function $z * x^y$. □

5.5.4 Fibonacci Numbers

The *Fibonacci sequence* (named after a famous Italian mathematician of the 10th century) is the sequence of integers whose first two terms are 0 and 1, and where each subsequent term is the sum of the previous two terms. So, the sequence starts out:

0 1 1 2 3 5 8 13 21 34 ...

Let us index the terms starting at 0, so the 0th fibonacci number is 0, the next one 1, and so forth. Our goal is to write a function that has argument n and returns the nth Fibonacci number. The style of programming applies to many other sequences in which each successive term is defined in terms of the previous ones. Note, particularly, the pattern matching applied in the last equation.

```
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Or, equivalently, we may write

```
fib n
 | n == 0    = 0
 | n == 1    = 1
 | otherwise = fib(n-1) + fib(n-2)
```

The first definition has three equations and the second has one conditional equation with three (guarded) clauses. Either definition works, but these programs are quite inefficient. Let us see how many times `fib` is called in computing `(fib 6)`, see Figure 5.1. Here each node is labeled with a number, the argument of a call on `fib`; the root node is labeled `6`. In computing `(fib 6)`, `(fib 4)` and `(fib 5)` have to be computed; so, the two children of `6` are `4` and `5`. In general, the children of node labeled `i+2` are `i` and `i+1`. As you can see, there is considerable recomputation; in fact, the computation time is proportional to the value being computed. (Note that `(fib 6) (fib 5) (fib 4) (fib 3) (fib 2) (fib 1)` are called 1 1 2 3 5 8 times, respectively, which is a part of the Fibonacci sequence). We will see a better strategy for computing Fibonacci numbers in the next section.

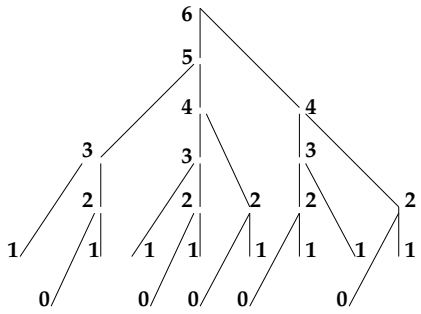

Figure 5.1: Call pattern in computing `(fib 6)`

5.5.5 Greatest Common Divisor

The greatest common divisor (gcd) of two positive integers is the largest positive integer that divides both m and n. (Prove the existence and uniqueness of gcd). Euclid gave an algorithm for computing gcd about 2,500 years ago, an algorithm that is still used today. Euclid's algorithm is as follows.

```
egcd m n
 | m == 0 = n
 | otherwise = egcd (n `mod` m) m
```

A simpler version of this algorithm, though not as efficient, is

```
gcd m n
 | m == n = m
 | m > n  = gcd (m - n) n
 | n > m  = gcd m (n - m)
```

This algorithm, essentially, computes the remainders, `(m ‘mod‘ n)` and `(n ‘mod‘ m)`, using repeated subtraction.

There is a modern version of the gcd algorithm, known as *binary gcd*. This algorithm uses multiplication and division by 2, which are implemented by shifts on binary numbers. The algorithm uses the following facts: (1) if `m == n`, then `gcd m n = m`, (2) if `m` and `n` are both even, say `2s` and `2t`, then `gcd m n = 2* (gcd s t)`, (3) if exactly one of `m` and `n`, say `m`, is even and equal to `2s`, then `gcd m n = gcd s n`, (5) if `m` and `n` are both odd and, say, `m > n`, then `gcd m n = gcd (m-n) n`. In this case, `m-n` is even whereas `n` is odd; so, `gcd m n = gcd (m-n) n = gcd ((m-n) ‘div‘ 2) n`.

```
bgcd m n
 | m == n               = m
 | (even m) && (even n) = 2 * (bgcd s t)
 | (even m) && (odd  n) =      bgcd s n
 | (odd  m) && (even n) =      bgcd m t
 | m > n                =      bgcd ((m-n) `div` 2) n
 | n > m                =      bgcd m ((n-m) `div` 2)
                         where s = m `div` 2
                               t = n `div` 2
```

We can estimate the running time (the number recursive calls) as a function of `x` and `y`, where `x` and `y` are arguments of `bgcd`. Note that the value of $\log_2$`x` + $\log_2$`y` decreases in each recursive call. So the execution time is at most logarithmic in the argument values.

Exercise 58

For positive integers m and n prove that

1. `gcd m n = gcd n m`
2. `gcd (m+n) n = gcd m n` □

5.6 Tuple

We have, so far, seen a few elementary data types. There are two important ways we can build larger structures using data from the elementary types—*tuple* and *list*. We cover tuple in this section.

Tuple is the Haskell's version of a *record*; we may put several kinds of data together and give it a name. In the simplest case, we put together two pieces of data and form a 2-tuple, also called a *pair*. Here are some examples.

```
(3,5)   ("Misra","Jayadev")  (True,3)  (False,0)
```

As you can see, the components may be of different types. Note that Haskell treats `(3)` and `3` alike, both are treated as numbers. Let us add the following definitions to our program file:

```
teacher = ("Misra","Jayadev")
uniqno = 59285
course = ("cs337",uniqno)
```

There are two predefined functions, *fst* and *snd*, that return the first and the second components of a pair, respectively.

```
*Main> fst(3,5)
3
*Main> snd teacher
"Jayadev"
*Main> snd course
59285
```

There is no restriction at all in Haskell about what you can have as the first and the second component of a tuple. In particular, we can create another tuple

```
hard = (teacher,course)
```

and extract its first and second components by

```
*Main> fst hard
("Misra","Jayadev")
*Main> snd hard
("cs337",59285)
```

Haskell allows you to create tuples with any number of components, but *fst* and *snd* are applicable only to a pair.

Revisiting the Fibonacci Computation As we saw in the Figure 5.1 of page 133, there is considerable recomputation in evaluating `fib n`, in general. Here, I sketch a strategy to eliminate the recomputation. We define a function, called `fibpair`, which has an argument `n`, and returns the pair `((fib n), (fib (n+1)))`, i.e., two consecutive elements of the Fibonacci sequence. This function can be computed efficiently, as shown below, and we may define `fib n = fst(fibpair n)`.

```
fibpair 0 = (0,1)
fibpair n = (y, x+y)
             where (x,y) = fibpair (n-1)
```

Exercise 59

1. What is the difference between `(3,4,5)` and `(3,(4,5))`?

2. A point in two dimensions is a pair of coordinates; assume that we are dealing with only integer coordinates. Write a function that takes two points as arguments and returns `True` iff either the $x-$coordinate or the $y-$coordinate of the points are equal. Here is what I expect (`ray` is the name of the function).

```
*Main> ray (3,5) (3,8)
True
*Main> ray (3,5) (2,5)
True
*Main> ray (3,5) (3,5)
True
*Main> ray (3,5) (2,8)
False
```

3. A line is given by a pair of distinct points (its end points). Define function `parallel` that has two lines as arguments and value `True` iff they are parallel. Recall from coordinate geometry that two lines are parallel if their slopes are equal, and the slope of a line is given by the difference of the $y-$coordinates of its two points divided by the difference of their $x-$coordinates. In order to avoid division by 0, avoid division altogether. Here is the result of some evaluations.

```
*Main> parallel ((3,5), (3,8)) ((3,5), (3,7))
True
*Main> parallel ((3,5), (4,8)) ((4,5), (3,7))
False
*Main> parallel ((3,5), (4,7)) ((2,9), (0,5))
True
```

Solution This program is due to Jeff Chang, class of Spring 2006.

```
   parallel ((a,b),(c,d)) ((u,v),(x,y))
 = (d-b) * (x-u) == (y - v) * (c - a)
```

4. The function `fibpair n` returns the pair `((fib n), (fib (n+1)))`. The computation of `(fib (n+1))` is unnecessary, since we are interested only in `fib n`. Redefine `fib` so that this additional computation is avoided.

Solution

```
fib 0 = 0
fib n = snd(fibpair (n-1))
```

5.7 Type

Every expression in Haskell has a *type*. The type may be specified by the programmer, or deduced by the interpreter. If you write `3+4`, the interpreter can deduce the type of the operands and the computed value to be integer (not quite, as you will see). When you define

```
imply p q = not p || q
digit c = ('0' <= c) && (c <= '9')
```

the interpreter can figure out that p and q in the first line are booleans (because || is applied only to booleans) and the result is also a boolean. In the second line, it deduces that c is a character because of the two comparisons in the right side, and that the value is boolean, from the types of the operands.

The type of an expression may be a primitive one: `Int`, `Bool`, `Char` or `String`, or a structured type, as explained below. You can ask to see the type of an expression by giving the command `:t`, as in the following.

```
*Main> :t ('0' <= '9')
'0' <= '9' :: Bool
```

The type of a tuple is a tuple of types, one entry for the type of each operand. In the following, `[Char]` denotes a string; I will explain why in the next section.

```
*Main> :t ("Misra","Jayadev")
("Misra","Jayadev") :: ([Char],[Char])
*Main> :t teacher
teacher :: ([Char],[Char])
*Main> :t course
course :: ([Char],Integer)
*Main> :t (teacher,course)
(teacher,course) :: (([Char],[Char]),([Char],Integer))
```

Each function has a type, namely, the types of its arguments in order followed by the type of the result, all separated by `->`.

```
*Main> :t imply
imply :: Bool -> Bool -> Bool
*Main> :t digit
digit :: Char -> Bool
```

Capitalizations for types Type names (e.g., `Int`, `Bool`) are always capitalized. The name of a function or parameter should never be capitalized.

5.7.1 Polymorphism

Haskell allows us to write functions whose arguments can be *any* type, or any type that satisfies some constraint. Consider the *identity* function:

```
identity x = x
```

This function's type is

```
*Main> :t identity
identity :: t -> t
```

That is for any type `t`, it accepts an argument of type `t` and returns a value of type `t`.

A less trivial example is a function whose argument is a pair and whose value is the same pair with its components exchanged.

```
exch (x,y) = (y,x)
```

Its type is as follows:

```
*Main> :t exch
exch :: (t, t1) -> (t1, t)
```

Here, `t` and `t1` are arbitrary types. So, `exch(3,5), exch (3,"misra"), exch ((2,'a'),5)` and `exch(exch ((2,'a'),5))` are all valid expressions. The interpreter chooses the most general type for a function so that the widest range of arguments would be accepted.

Now, consider a function whose argument is a pair and whose value is `True` iff the components of the pair are equal.

```
eqpair (x,y) = x == y
```

It is obvious that `eqpair (3,5)` makes sense, but not `eqpair(3,'j')`. We would expect the type of `eqpair` to be `(t,t) -> Bool`, but it is more subtle.

```
*Main> :t eqpair
eqpair :: (Eq t) => (t, t) -> Bool
```

This says that the type of `eqpair` is `(t, t) -> Bool`, *for any type* `t` *that belongs to the* `Eq` *class, i.e., types over which* `==` *is defined*. Otherwise, the test `==` in `eqpair` cannot be performed. Equality is not necessarily defined on all types, particularly on function types.

Finally, consider a function that sorts two numbers which are given as a pair.

```
sort (x,y)
 | x <= y = (x,y)
 | x >  y = (y,x)
```

The type of `sort` is

```
*Main> :t sort
sort :: (Ord t) => (t, t) -> (t, t)
```

It says that sort accepts any pair of elements of the same type, provided the type belongs to the `Ord` type class, i.e., there is an order relation defined over that type; `sort` returns a pair of the same type as its arguments. An order relation is defined over most of the primitive types. So we can do the following kinds of sorting. Note, particularly, the last example.

```
*Main> sort (5,2)
(2,5)
*Main> sort ('g','j')
('g','j')
*Main> sort ("Misra", "Jayadev")
("Jayadev","Misra")
*Main> sort (True, False)
(False,True)
*Main> sort ((5,3),(3,4))
((3,4),(5,3))
```

Polymorphism means that a function can accept and produce data of many different types. This allows us to define a single sorting function, for example, which can be applied in a very general fashion.

5.7.2 Type Classes

Haskell has an extensive type system, which we will not cover in this course. Beyond types are type classes, which provide a convenient treatment of overloading. A type class is a collection of types, each of which has a certain function (or set of functions) defined on it. Here are several examples of type classes: the `Eq` class consists of all types on which an equality operation is defined; the `Ord` class consists of all types on which an order relation is defined; the `Num` class consists of all types on which typical arithmetic operations (`+, *,` etc.) are defined. The following exchange is instructive.

```
*Main> :t 3
3 :: (Num t) => t
*Main> :t (3,5)
(3,5) :: (Num t, Num t1) => (t, t1)
```

Read the second line to mean 3 has the type `t`, where `t` is any type in the type class `Num`. The last line says that `(3,5)` has the type `(t, t1)`, where `t` and `t1` are arbitrary (and possibly equal) types in the type class `Num`. So, what is the type of `3+4`? It has the type of any member of the `Num` class.

```
*Main> :t 3+4
3 + 4 :: (Num t) => t
```

5.7.3 Type Violation

Since the interpreter can deduce the type of each expression, it can figure out if you have supplied the arguments of the right type for a function. If you provide invalid arguments, you will see something like this.

```
    *Main> digit 9

<interactive>:1:6:
    No instance for (Num Char)
      arising from the literal `9' at <interactive>:1:6
    Possible fix: add an instance declaration for (Num Char)
    In the first argument of `digit', namely `9'
    In the expression: digit 9
    In the definition of `it': it = digit 9
```

5.8 List

Each tuple has a bounded number of components —two each for `course` and `teacher` and two in `(teacher,course)`. In order to process larger amounts of data, where the number of data items may not be known a priori, we use the data structure *list*. A list consists of a finite sequence of items[2] **all of the same type**. Here are some lists.

```
    [1,3,5,7,9]   -- all odd numbers below 10
    [2,3,5,7]     -- all primes below 10
    [[2],[3],[5],[7]]   -- a list of lists
    [(3,5), (3,8), (3,5), (3,7)] -- a list of tuples
    [[(3,5), (3,8)], [(3,5), (3,7), (2,9)]] -- a list of list of tuples
    ['a','b','c'] -- a list of characters
    ["misra", "Jayadev"] ---- a list of strings
```

The following are not lists because not all their elements are of the same type.

```
    [[2],3,5,7]
    [(3,5), 8]
    [(3,5), (3,8,2)]
    ['J',"misra"]
```

The order and number of elements in a list matter. So,

`[2,3]` $\neq$ `[3,2]`
`[2]` $\neq$ `[2,2]`

[2]We deal with only finite lists in this note. Haskell permits definitions of infinite lists and computations on them, though only a finite portion can be computed in any invocation of a function.

5.8.1 The Type of a List

The type of any list is `[ItemType]` where `ItemType` is the type of one of its items. So, `[True]` is a list of booleans and so are `[True, False]` and `[True, False, True, False]`. Any function that accepts a list of booleans as arguments can process any of these three lists. Here are these and some more examples.

```
*Main> :t [True]
[True] :: [Bool]
*Main> :t [True, False]
[True,False] :: [Bool]
*Main> :t [(2,'c'), (3,'d')]
[(2,'c'), (3,'d')] :: (Num t) => [(t, Char)]
*Main> :t [[2],[3],[5],[7]]
[[2],[3],[5],[7]] :: (Num t) => [[t]]
*Main> :t [(3,5), (3,8), (3,5), (3,7)]
[(3,5),(3,8),(3,5),(3,7)] :: (Num t, Num t1) => [(t, t1)]
*Main> :t [[(3,5), (3,8)], [(3,5), (3,7), (2,9)]]
[[(3,5),(3,8)],[(3,5),(3,7),(2,9)]] :: (Num t, Num t1) => [[(t, t1)]]
*Main> :t ['a','b','c']
['a','b','c'] :: [Char]
```

A string is a list of characters, i.e., `[Char]`; each of its characters is taken to be a list item. Therefore, a list whose items are strings is a list of `[Char]`, or `[[Char]]`.

```
*Main> :t ["misra"]
["misra"] :: [[Char]]
```

Empty List A very special case is an empty list, one having no items. We write it as `[]`. It appears a great deal in programming. What is the type of `[]`?

```
*Main> :t []
[] :: [a]
```

This says that `[]` is a list of `a`, where `a` is *any* type. Therefore, `[]` can be given as argument wherever a list is expected.

5.8.2 The List Constructor *Cons*

There is one built-in operator that is used to construct a list element by element; it is pronounced *Cons* and is written as `:` (a colon). Consider the expression `x:xs`, where `x` is an item and `xs` is a list. The value of this expression is a list obtained by prepending `x` to `xs`. Note that `x` should have the same type as the items in `xs`. Here are some examples.

```
    *Main> 3: [2,1]
    [3,2,1]
    *Main> 3: []
    [3]
    *Main> 1: (2: (3: [])) --Study this one carefully.
    [1,2,3]
    *Main> 'j': "misra"
    "jmisra"
    *Main> "j": "misra"

<interactive>:1:5:
    Couldn't match expected type `[Char]' against inferred type `Char'
      Expected type: [[Char]]
      Inferred type: [Char]
    In the second argument of `(:)', namely `"misra"'
    In the expression: "j" : "misra"
```

5.8.3 Pattern Matching on Lists

When dealing with lists, we often need to handle the special case of the empty list in a different manner. Pattern matching can be applied very effectively in such situations.

Let us consider a function `len` on lists that returns the length of the argument list. We need to differentiate between two cases, as shown below.

```
len [] = ..
len (x:xs) = ..
```

The definition of this function spans more than one equation. During function evaluation with a specific argument —say, `[1,2,3]`— each of the equations is checked from top to bottom to find the first one where the given list matches the pattern of the argument. So, with `[1,2,3]`, the first equation does not match because the argument is not an empty list. The second equation matches because `x` matches with `1` and `xs` matches with `[2,3]`. Additionally, pattern matching assigns names to components of the data structure —`x` and `xs` in this example— which may then be used in the RHS of the function definition.

5.8.4 Recursive Programming on Lists

Let us try to complete the definition of the function `len` sketched above. Clearly, we expect an empty list to have length 0. The general case, below, should be studied very carefully.

```
len [] = 0
len (x:xs) = 1 + (len xs)
```

This style of programming on lists is very common: we define the function separately for empty and non-empty lists. Typically, the definition for non-empty list involves recursion. This style corresponds to induction in that empty list corresponds to the base case and the non-empty list to the inductive case.

Consider now a function that sums the elements of a list of integers. It follows the same pattern.

```
suml [] = 0
suml (x:xs) = x + (suml xs)
```

A function that multiplies the elements of a list of integers.

```
multl [] = 1
multl (x:xs) = x * (multl xs)
```

Next, we write a program for a function whose value is the maximum of a list of integers. Here it does not make much sense to talk about the maximum over an empty list.[3] So, our smallest list will have a single element, and here is how you pattern match for a single element.

```
maxl [x] = x
maxl (x:xs) = max x (maxl xs)
```

Exercise 60
Write a function that takes the conjunction (`&&`) of the elements of a list of booleans.

```
andl [] = True
andl (x:xs) = x && (andl xs)
```

So, we have

```
*Main> andl [True, True, 2 == 5]
False
```

□

Now consider a function whose value is not just one item but a list. The following function negates every entry of a list of booleans.

```
notl[] = []
notl (x:xs) = (not x) : (notl xs)
```

So,

```
*Main> notl [True, True, 2 == 5]
[False,False,True]
```

[3]But people do and they define it to be $-\infty$. The value $-\infty$ is approximated by the smallest value in type `Int` which is `minBound::Int`; similarly, $+\infty$ is approximated by the largest value, `maxBound::Int`.

The following function removes all negative numbers from the argument list.

```
negrem [] = []
negrem (x:xs)
 | x < 0     = negrem xs
 | otherwise = x : (negrem xs)
```

So,

```
*Main> negrem []
[]
*Main> negrem [2,-3,1]
[2,1]
*Main> negrem [-2,-3,-1]
[]
```

Pattern matching over a list may be quite involved. The following function, `divd`, partitions the elements of the argument list between two lists, putting the elements with even index in the first list and with odd index in the second list (list elements are numbered starting at 0). So, `divd [1,2,3]` is `([1,3],[2])` and `divd [1,2,3,4]` is `([1,3],[2,4])`. See Section 5.8.5 for another solution to this problem.

```
divd [] = ([], [])
divd (x: xs) = (x:ys, zs)
                where (zs,ys) = divd xs
```

We conclude this section with a small example that goes beyond "primitive recursion", i.e., recursion is applied not just on the tail of the list. The problem is to define a function `uniq` that returns the list of unique items from the argument list. So, `uniq[3, 2] = [3, 2]`, `uniq[3, 2, 2] = [3, 2]`, `uniq[3, 2, 3] = [3, 2]`.

```
uniq [] = []
uniq (x:xs) = x: (uniq(minus x xs))
              where
                    minus y [] = []
                    minus y (z: ys)
                     | y == z    = (minus y ys)
                     | otherwise = z: (minus y ys)
```

Note This program does not work if you try to evaluate `uniq []` on the command line. This has to do with type classes; the full explanation is beyond the scope of these notes.

Exercise 61

1. Define a function `unq` that takes two lists `xs` and `ys` as arguments. Assume that initially `ys` contains distinct elements. Function `unq` returns the list of unique elements from `xs` and `ys`. Define `uniq` using `unq`.

```
unq [] ys     = ys
unq (x:xs) ys
 | inq x ys   = unq xs ys -- inq x ys is: x in ys?
 | otherwise  = unq xs (x:ys)
                where
                 inq y []      = False
                 inq y (z: zs) = (y == z) || (inq y zs)
uniq xs = unq xs []
```

2. Define a function that creates a list of unique elements from a sorted list. So, a possible input is `[2,2,3,3,4]` and the corresponding output is `[2,3,4]`.

3. The *prefix sum* of a list of numbers is a list of equal length whose ith element is the sum of the first i items of the original list. So, the prefix sum of [3,1,7] is [3,4,11]. Write a linear-time algorithm to compute the prefix sum.
   Hint: Use function generalization.

```
ps xs = pt xs 0
        where pt [] c = []
              pt (x:xs) c = (c+x) : (pt xs (c+x))
```

5.8.5 Mutual Recursion

All of our examples so far have involved recursion in which a function calls itself. It is easy to extend this concept to a group of functions that call each other. To illustrate mutual recursion, I will consider the problem of partitioning a list; see page 144 for another solution to this problem.

It is required to create two lists of nearly equal size from a given list, *lis*. The order of items in *lis* is irrelevant, so the two created lists may contain elements from *lis* in arbitrary order. If *lis* has an even number of elements, say $2 \times n$, then each of the created lists has n elements, and if *lis* has $2 \times n + 1$ items, one of the lists has $n + 1$ elements and the other has n elements.

One possible solution for this problem is to determine the length of *lis* (you may use the built-in function `length`) and then march down *lis* half way, adding elements to one output list, and then continue to the end of *lis* adding items to the second output list. We adopt a simpler strategy. We march down *lis*, adding items alternately into the two output lists. We define two functions, `divide0` and `divide1` each of which partitions the argument list, `divide0` starts with prepending the first item of the argument into the first list, and `divide1` by prepending the first item to the second list. Here, `divide0` calls `divide1` and `divide1` calls `divide0`.

```
divide0 []      = ([],[])
divide0 (x: xs) = (x:f, s)
                   where (f,s) = divide1 xs

divide1 []      = ([],[])
divide1 (x: xs) = (f, x:s)
                   where (f,s) = divide0 xs
```

We then get,

```
*Main> divide0 [1,2,3]
([1,3],[2])
*Main> divide0 [1,2,3,4]
([1,3],[2,4])
```

Encoding finite state machines using mutual recursion I show a general strategy of encoding finite state machines using mutual recursion. The idea is to encode each state by defining a function, and each transition by calling the appropriate function.

First, consider a machine that accepts binary strings of even parity. Figure 5.2 is from Chapter 4.

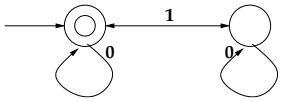

Figure 5.2: Accept strings of even parity

We encode this machine using function `zero` and `one` for the initial state and the other state. The input string is coded as argument of the functions, and the function values are boolean, `True` for acceptance and `False` for rejection. The machine is run on input string `s` by calling `zero(s)`.

```
zero [] = True
zero('0':xs) = zero(xs)
zero('1':xs) = one(xs)
one[] = False
one('0':xs) = one(xs)
one('1':xs) = zero(xs)
```

Next, consider a finite state transducer that accepts a string of symbols, and outputs the same string by (1) removing all white spaces in the beginning, (2) reducing all other blocks of white spaces (consecutive white spaces) to a single white space. Thus, the string (where - denotes a white space)

```
----Mary----had--a little---lamb-
```

is output as

```
Mary-had-a-little-lamb-
```

A machine to solve this problem is shown in Figure 5.3, where `n` denotes any symbol other than a white space

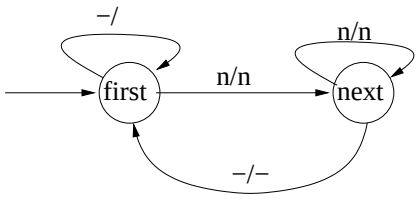

Figure 5.3: Remove unnecessary white space

We encode this machine as follows.

```
first [] = []
first(' ':xs) = first(xs)
first(x:xs) = x:next(xs)
next[] = []
next(' ':xs) = ' ':first(xs)
next(x:xs) = x:next(xs)
```

Exercise 62

Modify your design so that a trailing white space is not produced.

5.9 Examples of Programming with Lists

In this section, we take up more elaborate examples of list-based programming.

5.9.1 Some Useful List Operations

5.9.1.1 `snoc`

The list constructor `cons` of Section 5.8.2 (page 141) is used to add an item at the head of a list. The function `snoc`, defined below, adds an item at the "end" of a list.

```
snoc x []      = [x]
snoc x (y: xs) = y:(snoc x xs)
```

The execution of `snoc` takes time proportional to the length of the argument list, whereas `cons` takes constant time. So, it is preferable to use `cons`.

5.9.1.2 `concatenate`

The following function concatenates two lists in order. Remember that the two lists need to have the same type in order to be concatenated.

```
conc [] ys = ys
conc (x:xs) ys = x : (conc xs ys)
```

There is a built-in operator that does the same job; `conc xs ys` is written as `xs ++ ys`. The execution of `conc` takes time proportional to the length of the first argument list.

Exercise 63

1. Implement a double-ended queue in which items may be added at either end and removed from either end.
2. Define a function to left-rotate a list. Left-rotation of `[1,2,3]` yields `[2,3,1]` and of the empty list yields the empty list. □

5.9.1.3 `flatten`

Function `flatten` takes a list of lists, like

```
[ [1,2,3], [10,20], [], [30] ]
```

and flattens it out by putting all the elements into a single list, like

```
[1,2,3,10,20,30]
```

This definition should be studied carefully. Here `xs` is a list and `xss` is a list of lists.

```
flatten [] = []
flatten (xs : xss) = xs ++ (flatten xss)
```

Exercise 64

1. What is the type of `flatten`?
2. Evaluate

   ```
   ["I"," love"," functional"," programming"]
   ```

   and

   ```
   flatten ["I"," love"," functional"," programming"]
   ```

   and note the difference.
3. What happens if you apply `flatten` to a list of list of lists? □

5.9.1.4 `reverse`

The following function reverses the order of the items in a list.

```
rev []      = []
rev (x: xs) = (rev xs) ++ [x]  -- put x at the end of (rev xs)
```

The running time of this algorithm is $O(n^2)$, where n is the length of the argument list. (Prove this result using a recurrence relation. Use the fact that `append` takes $O(k)$ time when applied to a list of length k.) In the imperative style, reversing of an array can be done in linear time. Something is terribly wrong with functional programming! Actually, we *can* attain a linear time bound using functional programming.

The more efficient algorithm uses *function generalization* which was introduced for the `quickMlt` function for multiplication example in Section 5.5.3 (page 131). We define a function `reverse` that has two arguments `xs` and `ys`, each a list. Here `xs` denotes the part that remains to be reversed (a suffix of the original list) and `ys` is the reversal of the prefix. So, during the computation of `reverse [1,2,3,4,5]`, there will be a call to `reverse [4,5] [3,2,1]`. We have the identity

```
reverse xs ys = (rev xs) ++ ys
```

Given this identity,

```
reverse xs [] = rev xs
```

The definition of `reverse` is as follows.

```
reverse [] ys     = ys
reverse (x:xs) ys = reverse xs (x:ys)
```

Exercise 65

1. Show that the execution time of `reverse xs ys` is $O(n)$ where the length of `xs` is n.
2. Prove from the definition of `rev` that `rev (rev xs) = xs`.
3. Prove from the definition of `rev` and `reverse` that

   ```
   reverse xs ys = (rev xs) ++ ys
   ```

4. Show how to right-rotate a list efficiently (i.e., in linear time in the size of the argument list). Right-rotation of `[1,2,3,4]` yields `[4,1,2,3]`.
   Hint: use `rev`.

   ```
   right_rotate []    = []
   right_rotate xs    = y: (rev ys)
                        where
                        y:ys = (rev xs)
   ```

5. Try proving `rev (xs ++ ys) = (rev ys) ++ (rev xs)`. □

5.9.2 Towers of Hanoi

This is a well-known puzzle. Given is a board on which there are three pegs marked `'a', 'b' and 'c'`, on each of which can rest a stack of disks. There are n disks, $n > 0$, of varying sizes, numbered 1 through n in order of size. The disks are *correctly stacked* if they are increasing in size from top to bottom in each peg. Initially, all disks are correctly stacked at `'a'`. It is required to move all the disks to `'b'` in a sequence of steps under the constraints that (1) in each step, the top disk of one peg is moved to the top of another peg, and (2) the disks are correctly stacked at all times.

For $n = 3$, the sequence of steps given below is sufficient. In this sequence, a triple `(i,x,y)` denotes a step in which disk `i` is moved from peg `x` to `y`. Clearly, `i` is at the top of peg `x` before the step and at the top of peg `y` after the step.

```
[(1,'a','b'),(2,'a','c'),(1,'b','c'),(3,'a','b'),
 (1,'c','a'),(2,'c','b'),(1,'a','b')]
```

There is an iterative solution for this problem, which goes like this. Disk 1 moves in every alternate step starting with the first step. If n is odd, disk 1 moves cyclically from `'a'` to `'b'` to `'c'` to `'a'` ..., and if n is even, disk 1 moves cyclically from `'a'` to `'c'` to `'b'` to `'a'` .... In each remaining step, there is exactly one possible move: ignore the peg of which disk 1 is the top; compare the tops of the two remaining pegs and move the smaller one to the top of the other peg (if one peg is empty, move the top of the other peg to its top).

I don't know an easy proof of this iterative scheme; in fact, the best proof I know shows that this scheme is equivalent to an obviously correct recursive scheme.

The recursive scheme is based on the following observations. There is a step in which the largest disk is moved from `'a'`; we show that it is sufficient to move it only once, from `'a'` to `'b'`. At that moment, disk n is the top disk at `'a'` and there is no other disk at `'b'`. So, all other disks are at `'c'`, and, according to the given constraint, they are correctly stacked. Therefore, prior to the move of disk n, we have the subtask of moving the remaining $n - 1$ disks, provided $n > 1$, from `'a'` to `'c'`. Following the move of disk n, the subtask is to move the remaining $n - 1$ disks from `'a'` to `'c'`. Each of these subtasks is smaller than the original task, and may be solved recursively. Note that in solving the subtasks, disk n may be disregarded, because any disk can be placed on it; hence, its presence or absence is immaterial. Below, `tower n x y z` returns a list of steps to transfer `n` disks from peg `x` to `y` using `z` as an intermediate peg.

```
tower n x y z
 | n == 0    = []
 | otherwise =    (tower (n-1) x z y)
              ++ [(n,x,y)]
              ++ (tower (n-1) z y x)
```

A run of this program gives us

```
*Main> tower 3 'a' 'b' 'c'
[(1,'a','b'),(2,'a','c'),(1,'b','c'),(3,'a','b'),
 (1,'c','a'),(2,'c','b'),(1,'a','b')]
```

Exercise 66

1. What is the type of function `tower`?
2. What is the total number of moves, as a function of n?
3. Argue that there is no scheme that uses fewer moves, for any n.
4. Show that disk 1 is moved in every alternate move.
5. (very hard) What is a good strategy (i.e., minimizing the number of moves) when there are four pegs instead of three?
6. (Gray code; hard) Start with an n-bit string of all zeros. Number the bits 1 through n, from lower to higher bits. Solve the Towers of Hanoi problem for n, and whenever disk i is moved, flip the ith bit of your number and record it. Show that all 2^n n-bit strings are recorded exactly once in this procedure. □

5.9.3 Gray Code

If you are asked to list all 3-bit numbers, you will probably write them in increasing order of their magnitudes:

000 001 010 011 100 101 110 111

There is another way to list these numbers so that consecutive numbers (the first and the last numbers are consecutive too) differ in exactly one bit position.

000 001 011 010 110 111 101 100

The problem is to generate such a sequence for every n. Let us attack the problem by induction on n. For $n = 1$, the sequence 0 1 certainly meets the criterion. For $n+1$, $n \geq 1$, we argue as follows. Assume, inductively, that there is a sequence X_n of n-bit numbers in which the consecutive numbers differ in exactly one bit position. Now, we will take X_n and create X_{n+1}, a sequence of $n+1$-bit numbers with the same property. To gain some intuition, let us look at X_2. Here is a possible sequence:

00 01 11 10

How do we construct X_3 from X_2? Appending the same bit, a 0 or a 1, to the left end of each bit string in X_2 preserves the property among consecutive numbers, and it makes each bit string longer by one. So, from the above sequence we get: 000 001 011 010

Now, we need to list the remaining 3-bit numbers, which we get by appending 1s to the sequence X_2:

100 101 111 110

But merely concatenating this sequence to the previous sequence won't do; 010 and 100 differ in more than one bit position. But concatenating the reverse of the above sequence works, and we get

000 001 011 010 110 111 101 100

Define function `gray` to compute such a sequence given n as the argument. The output of the function is a list of 2^n items, where each item is a n-bit string. Thus, the output will be

```
*Main> gray 3
["000","001","011","010","110","111","101","100"]
```

Considering that we will have to reverse this list to compute the function value for the next higher argument, let us define a more general function, `grayGen`, whose argument is a natural number n and whose output is a pair of lists, `(xs,ys)`, where `xs` is the Gray code of n and `ys` is the reverse of `xs`. We can compute `xs` and `ys` in similar ways, without actually applying the reverse operation.

First, define a function `cons0` whose argument is a list of strings and which returns a list by prepending a '0' to each string in the argument list. Similarly, define `cons1` which prepends '1' to each string.

```
cons0 [] = []
cons0 (x:xs) = ('0':x):(cons0 xs)

cons1 [] = []
cons1 (x:xs) = ('1':x):(cons1 xs)
```

Then `grayGen` and `gray` are easy to define.

```
grayGen 0  = ([""],[""])
grayGen n  = ((cons0 a) ++ (cons1 b), (cons1 a) ++ (cons0 b))
                  where (a,b) = grayGen (n-1)

gray n = fst(grayGen n)
```

Exercise 67

1. Show another sequence of 3-bit numbers that has the Gray code property.

2. Prove that for all n,

```
rev a = b
      where (a,b) = grayGen n
```

You will have to use the following facts; for arbitrary lists `xs` and `ys`

```
rev (rev xs)   = xs
rev (cons0 ys) = cons0 (rev ys)
rev (cons1 ys) = cons1 (rev ys)
rev (xs ++ ys) = (rev ys) ++ (rev xs)
```

3. Given two strings of equal length, their *Hamming distance* is the number of positions in which they differ. Define a function to compute the Hamming distance of two given strings.

4. In a Gray code sequence consecutive numbers have hamming distance of 1. Write a function that determines if the strings in its argument list have the Gray code property. Make sure that you compare the first and last elements of the list. □

Exercise 68

This exercise is about computing winning strategies in simple 2-person games.

1. Given are two sequences of integers of equal length. Two players alternately remove the first number from either sequence; if one of the sequences becomes empty, numbers are removed from the other sequence until it becomes empty. The game ends when both sequences are empty, and then the player with the higher total sum wins the game (assume that sums of all the integers is odd, so that there is never a tie). Write a program to determine if the first player has a winning strategy. For example, given the lists `[2,12]` and `[10,7]`, the first player does not have a winning strategy; if he removes 10, then the second player removes 7, forcing the first player to remove 2 to gain a sum of 12 and the second player to gain 19; and if the first player removes 2, the second player removes 12, again forcing the same values for the both players.

   Solution

   Define `play(xs,ys)`, where `xs` and `ys` are lists of integers of equal length, that returns `True` iff the first player has a winning strategy. Define helper function `wins(b,xs,ys)`, where `xs` and `ys` are lists of integers, not necessarily of the same length, and `b` is an integer. The function returns `True` iff the first player has a winning strategy from the given configuration assuming that he has already accumulated a sum of `b`. Then `play(xs,ys) = wins(0,xs,ys)`. And, function `loses(b,xs,ys)` is `True` iff the first player has no winning strategy, i.e., the second player has a winning strategy.

   ```
   loses(b,xs,ys) = not (wins(b,xs,ys))
   wins(b,[],[]) = b > 0
   wins(b,x:xs,[]) = loses(-(b+x),xs,[])
   wins(b,[],y:ys) = loses(-(b+y),[],ys)
   wins(b,x:xs, y:ys) =
   ```

```
  loses(-(b+x),xs, y:ys) || loses(-(b+y),x:xs,ys)

play(xs,ys) = wins(0,xs,ys)
```

2. This problem is a variation on the previous exercise. The game involves a single sequence of numbers of even length, and the players alternately remove a number from either end of the sequence. The game ends when the sequence is empty (so, each player has removed exactly half the numbers), and the player with the higher sum wins. Determine if the first player has a winning strategy (it turns out that the first player always has a winning strategy; so, your program should always return `True`).

   You will have to represent a sequence where values can be removed from either end, but no value is ever added to it. This limited form of double-ended queue can be represented efficiently using two lists, as follows. Represent the initial sequence `xs` by `(xs, ys, n)`, where list `ys` is the reverse of `xs` and `n` is the length of the sequence. Remove a value from the left end by removing the first value from `xs`, and from the right end by removing the first value from `ys`; in each case, decrement `n`. You have the invariant that at any point the sequence is (1) the first `n` values from `xs`, or (2) the reverse of the first `n` values from `ys`.

   Solution

   We use an additional parameter `b` with the same meaning as before.

```
loses(b,xs,ys,n) = not (wins(b,xs,ys,n))
wins(b,_,_,0) = b > 0
wins(b,x:xs,y:ys,n) =
 loses(-(b+x),xs,y:ys,n-1) || loses(-(b+y),x:xs,ys,n-1)

play(xs) = wins(0,xs,rev(xs),length(xs))
```

5.9.4 Sorting

Consider a list of items drawn from some totally ordered domain such as the integers. We develop a number of algorithms for sorting such a list, that is, for producing a list in which the same set of numbers are arranged in ascending order.[4] We cannot do in situ exchanges in sorting, as is typically done in imperative programming, because there is no way to modify the argument list.

5.9.4.1 Insertion Sort

Using the familiar strategy of primitive recursion, let us define a function for sorting, as follows.

[4] A sequence of numbers $\ldots x\ y \ldots$ is *ascending* if for consecutive elements x and y, $x \leq y$ and *increasing* if $x < y$. It is *descending* if $x \geq y$ and *decreasing* if $x > y$.

```
isort []     = []
isort (x:xs) =  .. (isort xs) .. -- skeleton of a definition
```

The first line is easy to justify. For the second line, the question is: how can we get the sorted version of `(x:xs)` from the sorted version of `xs` —that is `isort xs`— and `x`? The answer is, insert `x` at the right place in `(isort xs)`. So, let us first define a function `insert y ys`, which produces a sorted list by appropriately inserting `y` in the sorted list `ys`.

```
insert y [] = [y]
insert y (z:zs)
 | y <= z   = y:(z:zs)
 | y > z    = z: (insert y zs)
```

Then, function `isort` is

```
isort []     = []
isort (x:xs) = insert x (isort xs)
```

Exercise 69

1. What is the worst-case running time of `insert` and `isort`?
2. What is the worst-case running time of `isort` if the input list is already sorted? What if the reverse of the input list is sorted (i.e., the input list is sorted in descending order)? □

5.9.4.2 Merge sort

This sorting strategy is based on merging two lists. First, we divide the input list into two lists of nearly equal size —function `divide0` of Section 5.8.5 (page 145) works very well for this— sort the two lists recursively and then merge them. Merging of sorted lists is easy; see function `merge` below.

```
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
 | x <= y   = x : (merge xs (y:ys))
 | x >  y   = y : (merge (x:xs) ys)
```

Based on this function, we develop `mergesort`.

```
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge left right
               where
                (xsl,xsr) = divide0 xs
                left      = mergesort xsl
                right     = mergesort xsr
```

Exercise 70

1. Why is

   ```
   merge [] [] = []
   ```

   not a part of the definition of `merge`?

2. Show that mergesort has a running time of $O(2^n \times n)$ where the argument list has length 2^n.

3. Modify `merge` so that it discards all duplicate items.

4. Develop a function similar to `merge` that has two ascending lists as arguments and creates an ascending list of common elements.

5. Develop a function that has two ascending lists as arguments and creates the difference, first list minus the second list, as an ascending list of elements.

6. Develop a function that has two ascending lists of integers as arguments and creates an increasing list of pairwise sums from the two lists (duplicates are discarded). □

5.9.4.3 Quicksort

Function `quicksort` partitions its input list `xs` into two lists, `ys` and `zs`, so that every item of `ys` is at most every item of `zs`. Then `ys` and `zs` are sorted and concatenated. Note that in `mergesort`, the initial partitioning is easy and the final combination is where the work takes place; in `quicksort` the initial partitioning is where all the work is.

We develop a version of `quicksort` that differs slightly from the description given above. First, we consider the partitioning problem. A list is partitioned with respect to some value `v` that is supplied as an argument; all items smaller than or equal to `v` are put in `ys` and all items greater than `v` are put in `zs`.

```
partition v []   = ([],[])
partition v (x:xs)
 | x <= v = ((x:ys),zs)
 | x > v  = (ys,(x:zs))
          where (ys,zs) = partition v xs
```

There are several heuristics for choosing `v`; let us choose it to be the first item of the given (nonempty) list. Here is the definition of `quicksort`.

```
quicksort [] = []
quicksort (x:xs) = (quicksort ys ) ++ [x] ++ (quicksort zs)
                 where (ys,zs) = partition x xs
```

Exercise 71

1. Show that each call in `quicksort` is made to a smaller argument.
2. What is the running time of `quicksort` if the input file is already sorted?
3. Find a permutation of 1 through 15 on which `quicksort` has the best performance; assume that the clause with guard `x <= v` executes slightly faster than the other clause.

   8 4 2 1 3 6 5 7 12 10 9 11 14 13 15

□

Exercise 72

1. Define a function that takes two lists of equal length as arguments and produces a boolean list of the same length as the result; an element of the boolean list is `True` iff the corresponding two elements of the argument lists are identical.
2. Define a function that creates a list of unique elements from a sorted list. Use this function to redefine function `uniq` of Section 5.8.4 (page 144).
3. Define function `zip` that takes a pair of lists of equal lengths as argument and returns a list of pairs of corresponding elements. So,

   ```
   zip ([1,2,3], ['a','b','c']) = [(1,'a'), (2,'b'), (3,'c')]
   ```

4. Define function `unzip` that is the inverse of `zip`:

   ```
   unzip (zip (xs,ys)) = (xs,ys)
   ```

5. Define function `take` where `take n xs` is a list containing the first `n` items of `xs` in order. If `n` exceeds the length of `xs` then the entire list `xs` is returned.
6. Define function `drop` where

   ```
   xs = (take n xs) ++ (drop n xs)
   ```

7. Define function `index` where `index i xs` returns the `i`th element of `xs`. Assume that elements in a list are indexed starting at 0. Also, assume that the argument list is of length at least `i`. □

Exercise 73

A matrix can be represented as a list of lists. Let us adopt the convention that each outer list is a column of the matrix. Develop an algorithm to compute the determinant of a matrix of numbers. □

Exercise 74

It is required to develop a number of functions for processing an employee database. Each entry in the database has four fields: *employee*, *spouse*, *salary* and *manager*. The employee field is a string that is the name of the employee, the spouse field is the name of his/her spouse –henceforth, "his/her" will be abbreviated to "its" and "he/she" will be "it"—, the salary field is the employee's annual salary and the manager field is the name of employee's manager. Assume that the database contains all the records of a hierachical (tree-structured) organization in which every employee's spouse is also an employee, each manager is also an employee except *root*, who is the manager of all highest level managers. Assume that *root* does not appear as an employee in the database.

A manager of an employee is also called its *direct* manager; an *indirect* manager is either a direct manager or an indirect manager of a direct manager; thus, *root* is every employee's indirect manager.

Write functions for each of the following tasks. You will find it useful to define a number of auxiliary functions that you can use in the other functions. One such function could be *salary*, which given a name as an argument returns the corresponding salary.

In the following type expressions, DB is the type of the database, a list of 4-tuples, as described above.

1. Call an employee *overpaid* if its salary exceeds that of its manager. It is *grossly overpaid* if its salary exceeds the salaries of all its indirect managers. List all overpaid and grossly overpaid employees. Assume that the salary of *root* is 100,000.

```
overpaid         :: DB ->  [String]
grossly_overpaid :: DB ->  [String]
```

2. List all employees who directly manage their spouses; do the same for indirect management.

```
spouse_manager :: DB ->  [String]
```

3. List all managers who indirectly manage both an employee and its spouse.

```
indirect_manager :: DB ->  [String]
```

4. Are there employees e and f such that e's spouse is f's manager and f's spouse is e's manager?

```
nepotism :: DB ->  [(String,String)]
```

5. Find the family that makes the most money.

```
rich :: DB ->  [(String,String)]
```

6. Define the *rank* of a manager as the number of employees it manages. Define the *worth* of a manager as its salary/rank. Create three lists in which you list all managers in decreasing order of their salaries, ranks and worth.

```
sorted_salaries :: DB ->  [String]
sorted_rank     :: DB ->  [String]
sorted_worth    :: DB ->  [String]
```

7. The database is in *normal form* if the manager of x appears as an employee before x in the list. Write a function to convert a database to normal form. Are any of the functions associated with the above exercises easier to write or more efficient to run on a database that is given in normal form?

```
normalize :: DB -> DB
```

5.9.4.4 Patience Sort

This section describes "Patience Sort", which is not really a sorting scheme at all. It is inspired by a technique used by magicians to divide a deck of cards into a sequence of piles. The technique is as follows: place the first card from the deck face-up in a pile; for every subsequent card from the deck scan all the piles from left to right until you find a pile whose top card is larger than the given card; place the given card on top of that pile; if there is no such pile then place the card in a new rightmost pile. Here, the cards are compared for order according to any fixed scheme, say first by suit (spade $<$ heart $<$ diamond $<$ club) and then by rank. Applied to a sequence of numbers, say, `7,2,3,12,7,6,8,4`, we will get four piles in the following order where the numbers in a pile are shown from top to bottom as a list: `[2,7],[3],[4,6,7,12],[8]`. Note that each pile is an increasing list of numbers from top to bottom, and the top numbers of the piles are non-decreasing from left to right. We will show one use of patience sort later in this section.

Let us write a function `patienceSort` that takes a list of numbers, not necessarily distinct, and outputs a list of piles according to the given scheme. Thus, we expect to see

```
*Main> patienceSort[7,2,3,12,7,6,8,4]
[[2,7],[3],[4,6,7,12],[8]]
*Main> patienceSort[2,2,2]
[[2],[2],[2]]
```

First, let us start with a helper function `psort` where `psort(ps,cs)` takes a list of piles `ps` and a list of numbers `cs` as input and outputs a list of piles by putting the numbers in `cs` into the given piles appropriately. Then,

```
patienceSort(cs) = psort([],cs)
```

For the design of `psort`, we consider putting numbers one at a time to the piles. To this end, we design yet another (simpler) helper function `psort1` where `psort1(ps, c)` puts a single number `c` to the list of piles `ps`. Then,

```
psort(ps,[]) = ps
psort(ps, c:cs) = psort(psort1(ps, c), cs)
```

The design of `psort1` consists of two clauses as shown below.

```
psort1([],c) = [[c]]
psort1((p:ps):pss,c)
 | c < p = (c:(p:ps)):pss
 | c >= p  = (p:ps):psort1(pss,c)
```

Longest Ascending Subsequence A problem of some importance is to find a longest ascending subsequence (*las*) in a given list of numbers. A las of a sequence is a subsequence $x_0, x_1, \cdots$, where $x_i \leq x_{i+1}$, for all i (where $i+1$ is defined). For example, given `[7,2,3,12,7,6,8,4]` a las is `[2,3,6,8]`.

The length of a las is simply the number of piles; we will take one number from each pile to construct this subsequence. Whenever we place a number x in a pile, we also store a link to the top number y in the previous pile. According to our construction, $y \leq x$. Construct a las by taking the top number of the last (rightmost) pile and following the links backwards. Prove this result (see exercise below).[5]

Exercise 75

1. Show that any ascending subsequence of the original sequence has at most one entry from a pile. Consequently, the length of the longest ascending subsequence is at most the number of piles.

2. Show that follwing the links backwards starting from any number gives an ascending subsequence.

3. Combining the two results above, prove that the proposed algorithm constructs a longest ascending subsequence.

4. Can there be many longest ascending subsequences? In that case, can you enumerate all of them given the piles?

5. (just for fun) Develop a totally different algorithm for computing a longest ascending subsequence.

[5] I am grateful to Jay Hennig, class of Fall 2010, for pointing out an error in the treatment of Longest Ascending Subsequence.

5.9.5 Polynomial Evaluation

A polynomial of degree n in variable x is of the form $a_n \times x^n + a_{n-1} \times x^{n-1} + \cdots + a_0 \times x^0$, where the coefficients a_is are real (or complex) numbers, and the highest coefficient a_n is non-zero. Thus, $2 \times x^3 - 3 \times x^2 + 0 \times x + 4$ is a polynomial of degree 3. It is customary to represent a polynomial by a list of its coefficients in the order from least to most significant coefficients, $[4, 0, -3, 2]$.

To evaluate a polynomial for a given value, say $2 \times x^3 - 3 \times x^2 + 0 \times x + 4$ at $x = 2$ gives us $2 \times 2^3 - 3 \times 2^2 + 0 \times 2 + 4 = 16 - 12 + 0 + 4 = 8$. This calculation can be done simply by writing the polynomial as

$$\begin{array}{ll} & a_0 + a_1 \times x + a_2 \times x^2 + \cdots + a_n \times x^n \\ = & a_0 + x \times (a_1 + a_2 \times x + \cdots + a_n \times x^{n-1}) \end{array}$$

The second term requires evaluation of a polynomial of lower degree, $a_1 + a_2 \times x + \cdots + a_n \times x^{n-1}$. We can translate this scheme directly to a recursive program. Below, the polynomial is represented as a list from its least significant to most significant coefficients, and x is a specific value at which the polynomial is being evaluated. We will use the convention that a polynomial with no terms always evaluates to 0.

```
ep1 [] x = 0
ep1 (a:as) x = a + x * (ep1 as x)
```

Observe that this algorithms evaluates $2 \times x^3 - 3 \times x^2 + 0 \times x + 4$ as follows.

$$\begin{array}{ll} & 4 + 0 \times x - 3 \times x^2 + 2 \times x^3 \\ = & 4 + x \times (0 - 3 \times x + 2 \times x^2) \\ = & 4 + x \times (0 + x \times (-3 + 2 \times x)) \\ = & 4 + x \times (0 + x \times (-3 + x \times (2))) \end{array}$$

This form of polynomial evaluation is known as *Horner's rule*.

5.10 Proving Facts about Recursive Programs

We are interested in proving properties of programs, imperative and recursive, for similar reasons: to guarantee correctness, establish equivalence among alternative definitions, and gain insight into the performance of the program. Our main tool in proving properties of functional programs is *induction*. In dealing with lists, it often amounts to a proof of the base case (empty list) followed by proofs for non-empty lists, using induction.

Let me illustrate the proof procedure with a specific example. First define a function that combines both reversal of a list and concatenation. Given below is function `rvc` that takes three arguments, which are all lists, and has the following specification.

```
rvc us vs q = us ++ rev(vs) ++ q    (*)
```

where ++ is the infix concatenation operator and rev reverses a list.

Using the specification of rvc, we define function rev' to reverse a list, and function conc to concatenate two lists.

```
rev' ys    = rvc [] ys []
conc xs ys = rvc xs [] ys
```

The code for rvc is:

```
rvc [] [] q     = q
rvc [] (v:vs) q = rvc [] vs (v:q)
rvc (u:us) vs q = u:(rvc us vs q)
```

Next, we show a proof that the code of rvc meets the specification (*). We will need the following facts about rev and ++ for the proof. We use the associativity of ++ wiithout explicitly referring to it.

(i) ++ is associative, i.e., `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`, for any lists xs, ys and zs

(ii) `[] ++ ys = ys` and `ys ++ [] = ys`, for any list ys

(iii) `x:xs = [x] ++ xs`, for any item x and list xs

(iv) `rev [] = []`

(v) `rev [x] = [x]`, where [x] is a list with a single item

(vi) `rev(xs ++ ys) = (rev ys) ++ (rev xs)`, for any lists xs and ys

Proof: Proof is by induction on the combined lengths of us and vs. In each case below the proof has to establish an equality, and we show that the left side and the right side of the equality reduce to the same value.

• `rvc [] [] q = [] ++ rev([]) ++ q` : From the definition of rvc, the left side is q. Using facts (iv) and (ii) the right side is q.

• `rvc [] (v:vs) q = [] ++ rev(v:vs) ++ q`:

```
      rvc [] (v:vs) q
=   {definition of rvc}
      rvc [] vs (v:q)
=   {induction hypothesis}
      [] ++ rev(vs) ++ (v:q)
=   {using fact (ii)}
      rev(vs) ++ (v:q)
=   {using facts (iii) and (i)}
      rev(vs) ++ [v] ++ [q]
```

And,

```
      [] ++ rev(v:vs) ++ q
=   {using fact (ii)}
      rev(v:vs) ++ q
=   {using fact(iii)}
      rev([v] ++ [vs]) ++ q
=   {using fact(vi)}
      rev(vs) ++ rev([v]) ++ q
=   {using fact(v)}
      rev(vs) ++ [v] ++ q
```

- `rvc (u:us) vs q = (u:us) ++ rev(vs) ++ q`:

```
      rvc (u:us) vs q
=   {definition of rvc}
      u:(rvc us vs q)
=   {induction hypothesis}
      u:(us ++ rev(vs) ++ q)
=   {using fact (iii)}
      [u] ++ us ++ rev(vs) ++ q
```

And,

```
      (u:us) ++ rev(vs) ++ q
=   {using fact (iii)}
      [u] ++ us ++ rev(vs) ++ q
```

5.11 Higher Order Functions

5.11.1 Function `foldr`

We developed a number of functions —`suml`, `multl`— in Section 5.8.4 (page 142) that operate similarly on the argument list: (1) for the empty list, each function produces a specific value (0 for `suml`, 1 for `multl`) and (2) for a nonempty list, say `x:xs`, the item `x` and the function value for `xs` are combined using a specific operator (`+` for `suml`, `*` for `multl`). This suggests that we can code a generic function that has three arguments: the value supplied as in (1) —written as `z` below—, the function applied as in (2) — written as `f` below—, and the the list itself on which the function is to be applied. Here is such a function.

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Then we can define

```
suml  xs = foldr (+)  0  xs
multl xs = foldr (*)  1  xs
```

Similarly, we can define for boolean lists

```
andl  xs = foldr (&&) True   xs
orl   xs = foldr (||) False  xs
eor   xs = foldr (/=) False  xs
```

The first two are easy to see. The third one implements the exclusive-or over a list; it treats `True` as 1, `False` as 0, and takes the modulo 2 sum over the list. That is, it returns the parity of the number of `True` elements in the list.

We can define `flatten` of Section 5.9.1.3 (page 148) by

```
flatten xs = foldr (++) [] xs
```

Note I have been writing the specific operators, such as `(+)`, within parentheses, instead of writing them as just `+`, for instance, in the definition of `suml`. This is because the definition of foldr requires `f` to be a prefix operator, and `+` is an infix operator; `(+)` is the prefix version of `+`. □

Note There is an even nicer way to define functions such as `suml` and `multl`; just omit `xs` from both sides of the function definition. So, we have

```
suml  = foldr (+)  0
multl = foldr (*)  1
```

In these notes, I will not describe the justification for this type of definition. □

Function `foldr` has an argument that is a function; `foldr` is called a *higher order function*. The rules of Haskell do not restrict the type of argument of a function; hence, a function, being a typed value, may be supplied as an argument. Function (and procedure) arguments are rare in imperative programming, but they are common and very convenient to define and use in functional programming. Higher order functions can be defined for any type, not just lists.

What is the type of `foldr`? It has three arguments, `f`, `z` and `xs`, so its type is

(type of `f`) `->` (type of `z`) `->` (type of `xs`) `->` (type of result)

The type of `z` is arbitrary, say `a`. Then `f` takes two arguments of type `a` and produces a result of type `a`, so its type is `(a -> a -> a)`. Next, `xs` is a list of type `a`, so its type is `[a]`. Finally, the result type is `a`. So, we have for the type of `foldr`

```
(a -> a -> a) -> a -> [a] -> a
```

Actually, the interpreter gives a more general type:

```
*Main> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

This means that the two arguments of `f` need not be of the same type. Here is an example; function `evenl` determines if all integers of a given list are even. For its definition, we use function `ev` that takes an integer and a boolean as arguments and returns a boolean.

```
ev x  b   = (even x) && b
evenl xs  = foldr ev True xs

*Main> evenl [10,20]
True
*Main> evenl [1,2]
False
```

Function `fold` Define a simpler version of `foldr`, called `fold`. It applies to nonempty lists only, and does not have the parameter `z`. Function `fold` is not in the standard Haskell library.

```
fold f [x] = x
fold f (x:xs) = f x (fold f xs)
```

Applying `fold` to the list `[a,b,c,d]` gives `f a (f b (f c d))`. To use an infix operator `!` on `[a,b,c,d]`, call `fold (!)[a,b,c,d]`. This gives `a!(b!(c!d))`, which is same as `a!b!c!d` when `!` is an associative operator. Quite often `fold` suffices in place of `foldr`. For example, we can define function `maxl` of Section 5.8.4, which computes the maximum element of a nonempty list, by

```
maxl = fold max
```

5.11.2 Function `map`

Function `map` takes as arguments (1) a function `f` and (2) a list of elements on which `f` can be applied. It returns the list obtained by applying `f` to each element of the given list.

```
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

So,

```
*Main> map not [True,False]
[False,True]
*Main> map even [2,4,5]
[True,True,False]
*Main> map chCase "jmisra"
"JMISRA"
*Main> map len ["Jayadev","Misra"]
[7,5]
```

The type of map is:

```
*Main> :t map
map :: (a -> b) -> [a] -> [b]
```

This function is so handy that it is often used to transform a list to a form that can be more easily manipulated. For example, to determine if all integers in a given list, `xs`, are even, we write

```
andl (map even xs)
```

where `andl` is defined in Section 5.11.1 (page 163). Here `(map even xs)` creates a list of booleans of the same length as the list `xs` such that the ith boolean is `True` iff the ith element of `xs` is even. The function `andl` then takes the conjunction of the booleans in this list.

Exercise 76

Redefine the functions `cons0` and `cons1` from page 152 using `map`. □

5.11.3 Function `filter`

Function `filter` has two arguments, a predicate `p` and a list `xs`; it returns the list containing the elements of `xs` for which `p` holds.

```
filter p []  = []
filter p (x:xs)
 | p x       = x: (filter p xs)
 | otherwise = (filter p xs)
```

So, we have

```
*Main> filter even [2,3,4]
[2,4]
*Main> filter digit ['a','9','b','0','c']
"90"
*Main> filter upper "Jayadev Misra"
"JM"
*Main> filter digit "Jayadev Misra"
""
```

The type of `filter` is

```
*Main> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Exercise 77

What is `filter p (filter q xs)`? In particular, what is `filter p (filter (not p) xs)`? □

5.12 Program Design: Boolean Satisfiability

We treat a longer example —boolean satisfiability— in this section. The problem is to determine if a propositional boolean formula is *satisfiable*, i.e., if there is an assignment of (boolean) values to variables in the formula that makes the formula *true*. For example, $(p \vee q) \ \wedge \ (\neg p \vee \neg q) \ \wedge \ (p \vee \neg q)$ is satisfiable with $p, q =$ *true*, *false*.

In Section 5.12.1, I introduce the problem more precisely and present the Davis-Putnam procedure, which is an effective solution method for this problem. In Section 5.12.2, I develop a Haskell implementation of this procedure by choosing a suitable data structure and then presenting an appropriate set of functions in a top-down fashion.

5.12.1 Boolean Satisfiability

The satisfiability problem is an important problem in computer science, one that may be applied in a surprisingly diverse range of problems, such as circuit design, theorem proving and robotics. It has been studied since the early days of computing science. Indeed, a landmark theoretical paper by Steve Cook demonstrated that if someone could devise a fast algorithm for the general satisfiability problem, it could be used to solve many other difficult problems quickly. As a result, for a long time, satisfiability was considered too difficult to tackle, and a lot of effort was invested in trying to design domain-specific heuristics to solve specific instances of satisfiability.

However, recent advances in the design of "satisfiability solvers" (algorithms for solving instances of the satisfiability problem) have changed this perception. Although the problem remains difficult in general, recent satisfiability solvers, such as Chaff [39], employ clever data structures and learning techniques that prove to work surprisingly well in practice (for reasons no one quite understands).

Today, satisfiability solvers are used as the core engine in a variety of industrial products. CAD companies like Synopsys and Cadence use them as the engine for their tools for property checking and microprocessor verification [36], in automatic test pattern generation [33], and even in FPGA routing [18]. Verification engineers at Intel, Motorola and AMD incorporate satisfiability solvers in their tools for verifying their chip designs. Researchers in artificial intelligence and robotics are discovering that their planning problems can be cast as boolean satisfiability, and solvers like Chaff outperform even their domain-specific planning procedures [27]. Other researchers are increasingly beginning to use satisfiability solvers as the engine inside their model checkers, theorem provers, program checkers, and even optimizing compilers.

5.12.1.1 Conjunctive Normal Form

A propositional formula is said to be in *Conjunctive Normal Form* (*CNF*) if it is the conjunction of a number of *terms*, where each term is the disjunction of

a number of *literals*, and each literal is either a variable or its negation. For example, the formula $(p \vee q) \ \wedge \ (\neg p \vee \neg q) \ \wedge \ (p \vee \neg q)$ is in CNF. Any boolean formula can be converted to a logically equivalent CNF formula. Henceforth, we assume that the input formula is in CNF.

5.12.1.2 Complexity of the Satisfiability Problem

The satisfiability problem has been studied for over five decades. It may seem that the problem, particularly the CNF version, is easy to solve: each term has to be made *true* for the entire conjunction to be *true* and a term can be made *true* by making any constituent literal *true*. However, the choice of literal for one term may conflict with another: for $(p \vee q) \ \wedge \ (\neg p \vee \neg q)$, if p is chosen to be *true* for the first term and $\neg p$ is chosen to be *true* for the second term, there is a conflict.

There is no known polynomial algorithm for the satisfiability problem. In fact, the CNF satisfiability problem in which each term has exactly 3 literals — known as 3-SAT— is NP-complete, though 2-SAT can be solved in linear time. However, there are several solvers that do extremely well in practice. The Chaff solver [50] can determine satisfiability of a formula with hundreds of thousands of variables and over a million terms in an hour, or so, on a PC circa 2002. This astounding speed can be attributed to (1) a very good algorithm, the Davis-Putnam procedure, which we study next, (2) excellent heuristics, and (3) fast computers with massive main memories.

5.12.1.3 The Davis-Putnam procedure

To explain the procedure, I will use the following formula, f, over variables p, q and r:

$$f \ :: \ (\neg p \vee q \vee r) \ \wedge \ (p \vee \neg r) \ \wedge \ (\neg q \vee r) \ \wedge \ (\neg p \vee \neg q \vee \neg r) \ \wedge \ (p \vee q \vee r)$$

Next, we ask whether f is satisfiable, *given that p is true*. Then, any term in f that contains p becomes *true*, and can be removed from consideration (since it is part of a conjunction). Any term that contains $\neg p$, say $(\neg p \vee q \vee r)$, can become *true* only by making $(q \vee r)$ *true* . Therefore, given that p is *true*, f is satisfiable iff f_p is satisfiable, where

$$f_p \ :: \ (q \vee r) \ \wedge \ (\neg q \vee r) \ \wedge \ (\neg q \vee \neg r)$$

Note that p does not appear in f_p.

Similarly, given that p is *false*, f is satisfiable provided that $f_{\neg p}$ is satisfiable, where

$$f_{\neg p} \ :: \ (\neg r) \ \wedge \ (\neg q \vee r) \ \wedge \ (q \vee r)$$

We have two mutually exclusive possibilities: p is *true* and $\neg p$ is *true* . Therefore, f is satisfiable iff either f_p is satisfiable or $f_{\neg p}$ is satisfiable. Thus, we have divided the problem into two smaller subproblems, each of which may be decomposed further in a similar manner. Ultimately, we will find that

- a formula is empty (i.e., it has no terms), in which case it is satisfiable, because it is a conjunction, or
- some term in the formula is empty (i.e., it has no literals), in which case the term (and hence also the formula) is unsatisfiable, because it is a disjunction.

As long as neither possibility holds, we have a nonempty formula none of whose terms is empty, and we can continue with the decomposition process.

The entire procedure can be depicted as a binary tree, see Figure 5.4, where each node has an associated formula (whose satisfiability is being computed) and each edge has the name of a literal. The literals on the two outgoing edges from a node are negations of each other. The leaf nodes are marked either `F` or `T`, corresponding to *false* and *true*. A path in the tree corresponds to an assignment of values to the variables. The value, `F` or `T`, at a leaf is the value of the formula (which is associated with the root) for the variable values assigned along the path to the leaf. Thus, in Figure 5.4, assigning *true* to p, $\neg q$ and r makes f *true* (therefore, f is satisfiable). For all other assignments of variable values, f is *false*.

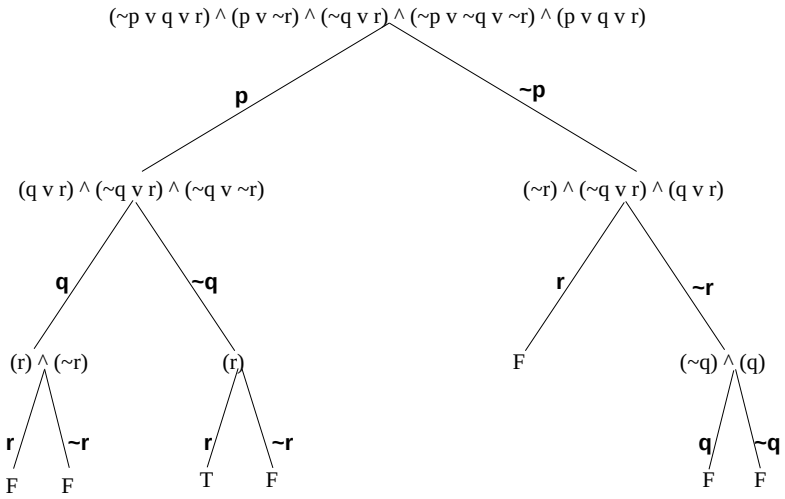

Figure 5.4: Davis-Putnam Computation

Note Assume that no term contains a variable and its negation.

5.12.2 Program Development

Data Structure Here, we consider representation of a formula in Haskell. Recall that

a *formula* is a conjunction of a set of *terms*.
a *term* is the disjunction of a set of *literals*, and
a *literal* is either a variable or the negation of a variable.

We can regard a formula as a list of terms and a term as a list of literals; conjunction and disjunction are implicit. Now, we consider representations of literals. A variable may be represented as a string, and a negation, such as $\neg p$, by a string, `"-p"`, whose first symbol is `"-"`. To preserve symmetry, let me add a `"+"` sign in front of a variable; so p is represented by `"+p"`. Then the representation of f is:

```
[
 ["-p", "+q", "+r"],
 ["+p", "-r"],
 ["-q", "+r"],
 ["-p", "-q", "-r"],
 ["+p", "+q", "+r"]
]
```

We should be clear about the types. I define the types explicitly (I have not told you how to do this in Haskell; just take it at face value).

```
type Literal = String
type Term    = [Literal]
type Formula = [Term]
```

The top-level function I define a function `dp` that accepts a formula as input and returns a boolean, `True` if the formula is satisfiable and `False` otherwise. So, we have

```
dp :: Formula -> Bool
```

If the formula is empty, then the result is `True`. If it contains an empty term, then the result is `False`. Otherwise, we choose some literal of the formula, decompose the formula into two subformulae based on this literal, solve each subformula recursively for satisfiability, and return `True` if either returns `True`.

```
dp xss
 | xss == []   = True
 | emptyin xss = False
 | otherwise   = (dp yss) || (dp zss)
                 where
                  v = literal xss
                  yss = reduce v xss
                  zss = reduce (neg v) xss
```

We have introduced the following functions which will be developed next:

`emptyin xss`: returns `True` if `xss` has an empty term,

`literal xss`: returns a literal from `xss`, where `xss` is nonempty and does not contain an empty term,

`neg v`: returns the string corresponding to the negation of `v`.

`reduce v xss`, where `v` is a literal: returns the formula obtained from `xss` by dropping any term containing the literal `v` and dropping any occurrence of the literal `neg v` in each remaining term.

Functions `emptyin`, `literal`, `reduce`, `neg`
The code for `emptyin` is straightforward:

```
-- Does the formula contain an empty list?
emptyin :: Formula -> Bool
emptyin []          = False
emptyin ([]: xss)   = True
emptyin (xs: xss)   = emptyin xss
```

Function `literal` can return any literal from the formula; it is easy to return the first literal of the first term of its argument. Since the formula is not empty and has no empty term, this procedure is valid.

```
{- Returns a literal from a formula.
   It returns the first literal of the first list.
   The list is not empty, and
   it does not contain an empty list.
-}
literal :: Formula -> Literal
literal ((x: xs):xss) = x
```

A call to `reduce v xss` scans through the terms of `xss`. If `xss` is empty, the result is the empty formula. Otherwise, for each term `xs`,

- if `v` appears in `xs`, drop the term,
- if the negation of `v` appears in `xs` then modify `xs` by removing the negation of `v`,
- if neither of the above conditions hold, retain the term.

```
-- reduce a literal through a formula
reduce :: Literal -> Formula -> Formula
reduce v []         = []
reduce v (xs:xss)
 | inl v xs       = reduce v xss
 | inl (neg v) xs = (remove (neg v) xs): (reduce v xss)
 | otherwise      = xs : (reduce v xss)
```

Finally, `neg` is easy to code:

```
-- negate a literal
neg :: Literal -> Literal
neg ('+': var) = '-': var
neg ('-': var) = '+': var
```

Function `reduce` introduces two new functions, which will be developed next.

`inl v xs`, where `v` is a literal and `xs` is a term: returns `True` iff `v` appears in `xs`,

`remove u xs`, where `u` is a literal known to be in term `xs`: return the term obtained by removing `u` from `xs`.

Functions `inl`, `remove`
Codes for both of these functions are straightforward:

```
-- check if a literal is in a term
inl :: Literal -> Term -> Bool
inl v [] = False
inl v (x:xs) = (v == x) || (inl v xs)

-- remove a literal u from term xs. u is in xs.
remove :: Literal -> Term -> Term
remove u (x:xs)
 | x == u    = xs
 | otherwise = (x : (remove u xs))
```

Exercise 78

1. Test the program.
2. Function `reduce` checks if `xss` is empty. Is this necessary given that `reduce` is called from `dp` with a nonempty argument list? Why doesn't `reduce` check whether `xss` has an empty term?
3. Rewrite `dp` so that if the formula is satisfiable, it returns the assignments to variables that make the formula *true*. □

5.12.3 Variable Ordering

It is time to take another look at our functions to see if we can improve any of them, either the program structure or the performance. Actually, we can do both.

Note that in `reduce` we look for a literal by scanning all the literals in each term. What if we impose an order on the variables and write each term in the given order of variables? Use the following ordering in f: `[ "p", "q" ,"r" ]`.

Then, a term like $(\neg p \vee q \vee r)$ is ordered whereas $(\neg p \vee r \vee q)$ is not. If each term in the formula is ordered and in `reduce v xss`, `v` is the *smallest* literal in `xss`, then we can check the first literal of each term to see whether it contains `v` or its negation.

Function `dp2`, given below, does the job of `dp`, but it needs an extra argument, a list of variables like `[ "p", "q" ,"r" ]`, which defines the variable ordering. Here, `reduce2` is the counterpart of `reduce`. Now we no longer need the functions `inl`, `remove` and `literal`.

```
dp2 vlist xss
 | xss == []   = True
 | emptyin xss = False
 | otherwise   = (dp2 wlist yss) || (dp2 wlist zss)
                 where
                  v:wlist = vlist
                  yss     = reduce2 ('+': v) xss
                  zss     = reduce2 ('-': v) xss

-- reduce a literal through a formula
reduce2 :: Literal -> Formula -> Formula
reduce2 w []         = []
reduce2 w ((x:xs):xss)
 | w == x       = reduce2 w xss
 | (neg w) == x = xs: (reduce2 w xss)
 | otherwise    = (x:xs): (reduce2 w xss)
```

A further improvement is possible. Note that `reduce2` scans its argument list twice, once for `w` and again for `neg w`. We define `reduce3` to scan the given list only once, to create two lists, one in which `w` is removed and the other in which `neg w` is removed. Such a solution is shown below, where `reduce3` is the counterpart of `reduce2`. Note that the interface to `reduce3` is slightly different. Also, we have eliminated the use of function `neg`.

```
dp3 vlist xss
 | xss == []   = True
 | emptyin xss = False
 | otherwise   = (dp3 wlist yss) || (dp3 wlist zss)
                 where
                  v:wlist   = vlist
                  (yss,zss) = reduce3 v xss

reduce3 v []         = ([],[])
reduce3 v ((x:xs):xss)
 | '+': v == x = (yss   , xs:zss)
 | '-': v == x = (xs:yss, zss   )
 | otherwise   = ((x:xs):yss, (x:xs):zss)
                  where
                   (yss,zss) = reduce3 v xss
```

Exercise 79

1. What are the types of `dp2`, `reduce2`, `dp3` and `reduce3`?
2. When `dp2 vlist xss` is called initially, `vlist` is the list of names of the variables in `xss`. Argue that the program maintains this as an invariant. In particular, `vlist` is empty iff `xss` is empty.
3. Is there any easy way to eliminate the function `emptyin`? In particular, can we assert that any empty term will be the first term in a formula?
4. Here is another strategy that simplifies the program structure and improves the performance. Convert each variable to a distinct positive integer (and its negation to the corresponding negative value). Make sure that each term is an increasing list (in magnitude). Having done this, the argument `vlist` is no longer necessary. Modify the solution to accommodate these changes. □

5.13 A Real-World Application: Google's Map-Reduce

Researchers at Google have used certain functional programming ideas to encode a variety of web-based applications which are then run on massively parallel clusters of machines. The following section is based on the work reported by Jeffrey Dean and Sanjay Ghemawat [13].

Many web based applications have two outstanding features: (1) they are large, both in the amount of computation and required storage, and (2) they can be executed effectively on many machines running in parallel, where the tasks are distributed appropriately among the machines. As an example, Google researchers performed an experiment where 10^{10} 100-byte records were searched for a particular text pattern. Using 1,800 machines (where each machine had two 2GHz Intel Xeon processors and 4GB of memory) the entire computation could be performed in 150 seconds, of which about a minute is spent on initialization. The scheme they have developed is fairly robust with respect to machine failures, and scalable in that more machines can be added easily to the cluster and that computations make effective use of a larger cluster.

The main idea behind their scheme is the observation that many web-based applications can be written as a *map* operation followed by a *reduce* operation; these operations and their terminologies are similar, though not identical, to the corresponding operations we have studied in this chapter.

First, *map* is a function that takes a (*key*, *value*) pair as argument and returns a list of (*key*, *value*) pairs. The argument keys and values may have different types from the result keys and values. As an example, the argument may be a pair (n, s), where n is a positive integer and s is a string. Here s is the contents of a text document and n is a small value, say 12, which specifies the length of words of interest. Function *map* may return a list of (w, c) pairs,

where w is a word of length n or less in the document, and c is the number of occurrences of w in the document. Typically, *map* is applied not to a single document, but millions of documents at once. This feature allows us to deploy the massive cluster of computers effectively.

Next, the outputs of all the maps, say M lists of (w, c) pairs for M documents, are aggregated into R lists where each list holds all the counts for a single word. Now, *reduce* operation is applied to each of these R lists individually, producing R outputs. In this example, a reduce operation might sum up the counts, which will tell us something about the most commonly occurring words in all the documents.

In general, the user provides the code for both *map* and *reduce*, specifies the arguments over which they have to be applied, and suggests certain parameters for parallel processing. From [13]: "The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system."

5.13.1 Some Example Problems

Here are some examples that can be expressed as Map-Reduce computations. These examples are taken from [13].

Distributed Grep "grep" is a unix utility that searches for a pattern in a given string (the string is typically a document). In this example, a large number of documents have to be searched for a specific pattern. Function *map* is applied to a document and a pattern, and it emits a line wherever there is a match. Function *reduce* is the identity function; it merely copies the supplied argument to the output.

Count of URL Access Frequency We are given a large log of web page requests. We would like to know how often certain web pages are being accessed. The map function takes a single request and outputs $\langle URL, 1\rangle$. The reduce function sums the values for the same URL and emits a total count for each URL.

Reverse Web-Link Graph We are given a large number of web documents. We would like to compute a list of URLs that appear in all these documents, and for each URL the list of documents in which it appears. So, the output is of the form $\langle target, list(source)\rangle$, where *target* is a URL mentioned in each of the *source* documents.

Function *map* applied to a single source document d outputs a list of pairs (t, d), where t is a target URL mentioned in d. Function *reduce* concatenates

the list of all source URLs associated with a given target URL and emits the pair: $\langle target, list(source)\rangle$.

Inverted Index This problem is very similar to the Reverse Web-Link Graph problem, described above. Instead of URLs, we look for common words in web documents. The required output is $\langle word, list(source)\rangle$. It is easy to augment this computation to keep track of word positions.

Distributed Sort The map function extracts a key from each document, and emits a $\langle key, document\rangle$ pair. The reduce function simply emits all pairs unchanged. But, the implementation imposes an order on the outputs which results in sorted output.

5.13.2 Parallel Implementation and Empirical Results

The *map* operation is applied individually to a large number of items; so, these computations can be done in parallel. For each key the *reduce* operation can also be applied in parallel. If *reduce* is associative, then its computation can be further distributed, because it can be applied at each machine to a group of values corresponding to one key, and the results from all the machines for the same key can be combined.

There are many possible parallel implementations which would would have different performance on different computing platforms. The following implementation is tuned for the platform at Google, which consists of large clusters of commodity PCs connected together with switched Ethernet. The machines are typically dual-processor x86 running Linux, with 2 to 4 GB of memory per machine. Commodity networking hardware is used, which typically delivers either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth. A cluster consists of hundreds or thousands of machines, and therefore machine failures are common. Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

A master processor assigns tasks to a set of worker machines. To apply *map* for a given job, the input data is first partitioned into a set of M splits by the master; typical split size is 16 to 64 megabytes. The splits are assigned by the master, and processed in parallel by different worker machines. To apply *reduce*, the outputs of the *map* invocations are partitioned into R pieces. A typical partitioning strategy is to use a hash function h, and assign key k to the partition $h(k) \bmod R$. Then all values for the same key are assigned to the same partition (as well as for all keys with the same hash value).

The authors report extremely positive results about fault tolerance. A failed worker process is detected by the master using time-out. The master then reschedules the task assigned to it (and ignores any output from the previous computation).

Acknowledgement I am grateful to Ham Richards and Kaustubh Wagle for thorough readings of these notes, and many suggestions for improvements. Rajeev Joshi has provided me with a number of pointers to the applications of boolean satisfiability.