Functional C

Pieter Hartel • Henk Muller

ADDISON-WESLEY

Functional C

Pieter Hartel Henk Muller

Functional C

Pieter Hartel
University of Southampton

Henk Muller
University of Bristol

To Marijke

Pieter

To my family and other sources of inspiration

Henk

Preface

The Computer Science Departments of many universities teach a functional language as the first programming language. Using a functional language with its high level of abstraction helps to emphasize the principles of programming. Functional programming is only one of the paradigms with which a student should be acquainted. Imperative, Concurrent, Object-Oriented, and Logic programming are also important. Depending on the problem to be solved, one of the paradigms will be chosen as the most natural paradigm for that problem.

This book is the course material to teach a second paradigm: *imperative programming*, using C as the programming language. The book has been written so that it builds on the knowledge that the students have acquired during their first course on functional programming, using SML. The prerequisite of this book is that the principles of programming are already understood; this book does not specifically aim to teach 'problem solving' or 'programming'. This book aims to:

- Familiarise the reader with *imperative programming* as another way of implementing programs. The aim is to preserve the programming style, that is, the programmer thinks functionally while implementing an imperative program.

- Provide understanding of the *differences between functional and imperative programming*. Functional programming is a high level activity. The ordering of computations and the allocation of storage are automatic. Imperative programming, particularly in C, is a low level activity where the programmer controls both the ordering of computations and the allocation of storage. This makes imperative programming more difficult, but it offers the imperative programmer opportunities for optimisations that are not available to the functional programmer.

- Familiarise the reader with the *syntax and semantics of ISO-C*, especially the power of the language (at the same time stressing that power can kill). We visit all dark alleys of C, from `void *` to pointer arithmetic and assignments in expressions. On occasions, we use other languages (like C++ and Pascal) to illustrate concepts of imperative languages that are not present in C. C has been chosen because it is a de facto standard for imperative programming, and because its low level nature nicely contrasts with SML. Those who want to learn, for example, Modula-2 or Ada-95 afterwards should not find many difficulties.

- Reinforce the *principles of programming* and *problem solving*. This is facilitated by the use of three different languages (mathematics, a functional language, and an imperative language). The fact that these widely differing languages have common aspects makes the idea that programming principles exist and that they are useful quite natural.

- Reinforce the *principle of abstraction*. Throughout the book we encourage the student to look for more abstract solutions, for example, by viewing the signature of a function as an abstraction of its purpose, by using procedural abstractions (in particular higher order functions) early on, and by using data abstraction.

- Guide the student from specification and mathematics to implementation and *software engineering*. In the first chapters the emphasis is on writing correct functions and as we make progress the emphasis gradually shifts to transforming correct functions into efficient and reusable functions. Clean interfaces are of paramount importance, and are sacrificed for better efficiency only as a last resort.

Each problem in this book is solved in three steps:

- A specification of the problem is made.

- An appropriate algorithm is found to deliver solutions that satisfy the specification.

- The algorithm is implemented as efficiently as possible. Throughout the book, the emphasis is on this third step.

The language of mathematics is used to specify the problems. This includes the basics of set theory and logic. The student should have some familiarity with the calculi of sets, predicate logic, and propositional logic. This material is taught at most universities during a first course on discrete mathematics or formal logic.

The appropriate algorithm is given in SML. SML is freely available for a range of platforms (PC's, UNIX work stations, Apple), and is therefore popular as a teaching language. As many functional languages are not too different from SML, an appendix gives a brief review of SML for those familiar with any of the other main stream functional languages, such as Miranda, Haskell, Clean, or Scheme.

As the target language to implement solutions in an imperative style we have chosen C. The choice to use C and not C++ was a difficult one. Both languages are mainstream languages, and would therefore be suitable as the target language. We have chosen C because it more clearly exposes the low level programming. To illustrate this consider the mechanisms that the languages provide for call by reference. In C, arguments must be explicitly passed as a pointer. The caller must pass the *address*, the callee must *dereference* the pointer. This in contrast with the call by reference mechanism of C++ (and Pascal and Modula-2). This explicit call by reference is a didactical asset as it clearly exposes the model behind call by reference, and its dangers (in the form of unwanted aliases).

As this book is intended to be used in a first year course, only few assumptions were made about prior knowledge of the students. Reasoning about the correctness of programs requires proof skills, which students might not have acquired at this stage. Therefore we have confined all proofs to specially marked exercises. To distinguish the programming exercises from the exercises requiring a proof, we have marked the latter with an asterisk. We are confident that the book can be used without making a single proof. However we would recommend the students to go through the proofs on a second reading. The answers to one third of the exercises are provided in Appendix A.

The student should have an understanding of the basic principles of computing. This would include base 2 arithmetic and the principles of operation of the von Neumann machine. A computer appreciation course would be most appropriate to cover this material. The book contains examples from other areas of computer science, including data bases, computer graphics, the theory of programming languages, and computer architecture. These examples can be understood without prior knowledge of these areas.

Acknowledgements

The help and comments of Hugh Glaser, Andy Gravell, Laura Lafave, Denis Nicole, Peter Sestoft, and the anonymous referees have been important to us. The material of the book has undergone its first test in Southampton in 1995/1996. The first year Computer Science students of 1995, and in particular Jason Datt and Alex Walker have given us a lot of useful feedback.

We have used a number of public domain software tools in the development of the book. The `noweb` literate programming tools of Norman Ramsey, the rail road diagramming tools from L. Rooijakkers, `gpic` by Brian Kernighan, TEX, LATEX, New Jersey SML, and the Gnu C compiler were particularly useful.

Contents

Chapter 1

Introduction

Programming is the activity of instructing a computer so that it will help to solve a problem. These instructions can be prepared on the basis of a number of paradigms. This book has been written for those who are familiar with the functional paradigm, using SML as a programming language, and who wish to learn how to program in the imperative paradigm, using C as a programming language.

1.1 The functional and the imperative paradigms

The functional and imperative paradigms operate from different viewpoints. The functional paradigm is based on the *evaluation of expressions*, and binding variables to values. The basic program phrase is the expression; the purpose of evaluating an expression is to produce a value. The order in which subexpressions are evaluated does not affect the resulting values.

The imperative paradigm is based on the *execution of statements*, and having a store where the statements can leave their results. The basic program phrase is the statement; the purpose of executing a statement is to change the store. The order in which statements are executed does affect the resulting values in the store. The current set of values in the store is referred to as the *state* of the program.

The reasons for these different approaches lie in the roots of the languages based on these paradigms. Functional languages were developed coming from mathematics. The foundation of these languages is therefore clean and well understood. Imperative languages were designed coming from the machine operational point of view: The von Neumann architecture has a memory and a processor operating on this memory. The imperative paradigm is a high level abstraction of this model.

Functional and imperative languages can be used to program in either style: it is possible to write an imperative program in SML, and it is also possible to write a functional program in C. However, these programs are often ‘unnatural’, in that their formulation is clumsy because the language does not offer the most appropriate abstractions.

As an example, consider the classical problem of generating pseudo random numbers. This is how we can formulate this problem functionally in C:

```
int functional_random( int seed ) {
  return 22 * seed % 37 ;
}
```

The functional implementation should be read as follows: the first line introduces a function by the name of `functional_random`. This function accepts an integer argument called `seed` and it also returns an integer value. The function body (enclosed in curly brackets `{` and `}`) contains a single return statement. The argument of the return statement is an expression, which multiplies the value of `seed` by `22` and then yields the remainder after division by `37`.

To use the function `functional_random`, one has to choose a suitable start value for the `seed`, for example `1`, and then repeatedly apply the function `functional_random`. Successive pseudo random values will then be obtained. Consider the following C program fragment:

```
int first  = functional_random( 1 ) ;
int second = functional_random( first ) ;
int third  = functional_random( second ) ;
```

The value of the variable `first` will be `22`; the value of the variable `second` will be `3`, (because `22*22` = `13*37+3`) and the value of `third` will be `29`.

The function `functional_random` is a *pure function*, that is the value returned by each call of `functional_random` depends exclusively on the value of its argument. This implies that `functional_random` will always give the same answer when it is called with the same argument value. A pure function is therefore a good building block.

The pseudo random number generator can also be written in an imperative style:

```
int seed = 1 ;
int imperative_random( void ) {
  seed = 22 * seed % 37 ;
  return seed ;
}
```

The imperative implementation should be read as follows: the first line defines a *global variable*, called `seed`, which holds the current seed value. The initial value is `1`. The next line introduces a function by the name of `imperative_random`. The function has no arguments, which is indicated by the word `void`. The function *changes* the value of `seed`, and returns its value after having made the change. The modification of the store is referred to as a *side effect*, because it was an effect additional to returning the pseudo random number.

That this function is imperative becomes clear when we 'mentally' execute the code. The first call to `imperative_random` will return `22`, whereupon the variable `seed` has the value `22`. This will cause the function to return `3` on the next call. So on every invocation, the function `imperative_random` will return a new value, which is exactly what we require from a pseudo random number generator. The order of calls becomes important, as the value returned by `imperative_random` now depends on the state, and not on its argument.

Both paradigms have their advantages. The imperative paradigm makes it easier to deal with state, as the state does not have to be communicated from one function to the other, it is always present. The functional paradigm allows us to create building blocks that can be used more freely. We will elaborate on these issues below.

1.1.1 The advantage of state

A useful extension of the random function would be to build a function that returns the value of a dice. The value of a dice can be computed by taking the random number modulo 6, and adding one to it, which will return a value in the range 1 … 6. The imperative function for the dice would read:

```
int imperative_dice( void ) {
  return imperative_random() % 6 + 1 ;
}
```

A random number is generated, the modulo operation is performed, and one is added. Writing the functional version is more difficult, two numbers have to be returned from this function: the value of the dice and the state of the random number generator. The caller of the functional dice would have to take one of the numbers, and remember to pass the next on to the next call.

1.1.2 The advantage of pure functions

Storing state somewhere in a hidden place has a disadvantage: it becomes more difficult to create functions that can be used as neat building blocks. As an example, assume that we would like to roll two dice every time. The theory of random number generators tells us that it is incorrect to use alternate numbers from one random number generator to generate the values of the two dice [5]. Instead, two independent random generators must be used.

The functional version offers a random generator that can be used as a building block, supposing that the initial seeds where stored in r and s, then the following fragment of code will generate a value for the two dice:

```
int x = functional_random( r ) ;
int y = functional_random( s ) ;
int dice = x%6 + 1 + y%6 + 1 ;
```

It is impossible to achieve this with the imperative version, because there is only one variable seed which stores *the* seed.

1.1.3 Idiomatic building blocks in C

Ideally, we would like to have the best of both worlds. The reader can look ahead to Chapter 8 to see a random number generator, which is a good building block, and which passes the state around in a manner that scales well beyond a single function.

This is the aim of this book: we wish to create highly idiomatic and efficient C code, but also wish to create good building blocks, preserving all the techniques that are common knowledge in the world of functional languages. Examples include pure functions, polymorphic functions, currying, algebraic data types and recursion.

1.2 Guide to the book

The next chapter discusses the basic execution model of C. The basic syntax and data types are also presented. In that chapter a declarative subset of C is used. This does not result in idiomatic or efficient C programs, but it serves to familiarise the reader with the basic syntax of C. Chapter 2 also introduces the first systematic transformation of functional code into C.

Chapter 3 discusses iteration. Functional languages iterate over data structures by means of direct recursion or by means of indirect recursion through higher order functions such as `map` and `foldl`. C offers constructions that iterate without recursion by repeatedly executing certain parts of the program code. In Chapter 3 we create efficient and idiomatic C for most example programs of Chapter 2. This is done using a number of systematic, but informal, program transformation schemas.

Chapter 4 discusses the type constructors of C that are necessary to build (non recursive) algebraic data types. These constructors are called *structures* and *unions*. The chapter discusses how to create complex types and ends with a discussion on destructive updates in these data structures (using a *pointer*).

The first 4 chapters discuss the basic data types and their C representation. More complex data types can store sequences data. There are a number of representations for storing sequences. In functional languages, lists are popular; arrays are used when efficient random access is required to the elements. Lists are less popular in C because the management of lists is more work than the management of lists in functional languages. Sequences, arrays, lists, and streams are discussed in Chapters 5 to 7.

Chapter 5 presents the basic principles of sequences and the implementation of arrays. Arrays in C are at a low level, but it is shown that high level structures, as available in functional languages, can be constructed. Lists are discussed in Chapter 6. The implementation of lists requires explicit memory management; this is one of the reasons that using lists is less convenient in C than in SML. The stream, a list of items that are consumed or written sequentially (as in I/O), are the topic of Chapter 7.

Chapter 8 finally goes into details of how the module system of C works, comparing it to the module system of SML. Modular programming is the key issue in software engineering. Defining interfaces in such a way that modules have a clear function and the proper use of state are the subjects of this chapter.

The last chapter, Chapter 9, shows three case studies in elementary graphics. The first case is completely worked out, it shows how to use an X-window system for drawing a fractal for the Mandelbrot set. The second case study is partially worked out, it designs a system for device independent graphics. There are large parts left to be implemented by the reader. The third study develops an implementation for a simple graphics language. The algorithms are outlined, and the data structures are sketched, the implementation of this case is left to the reader.

Appendix A contains the answers to a selection of the exercises. The exercises give readers the opportunity to test and improve their skills. There are two types of exercises. The normal exercises reinforce the general skills of problem solving. Many of them require an implementation of some SML code, and almost all of them require the implementation of some C functions. The exercises marked with an asterisk are targeted at readers who are interested in the fundamental issues of programming. All proofs of the theorems that we pose are left as an 'exercise⋆' to the reader.

Appendix B is a brief review of SML for people familiar with other functional languages. It suffices that people can read SML programs. We only discuss the subset of SML that we use in the book and only in terms of other functional languages. This appendix also discusses the (small set of) SML library functions that we use.

Appendix C lists the library functions of C. All programming languages come with a collection of primitive operators and data types as well as a collection of library modules providing further facilities. C is no exception and has a large set of libraries. We present a small number of functions that are present in the C library.

The last Appendix D gives the complete syntax of ISO-C using railroad diagrams. These are intuitively clearer than the alternative BNF notation for syntax. The syntax diagrams are intended as a reference.

This book is not a complete reference manual. The reader will thus find it useful to be able to refer to the ISO-C reference manual [7]. It contains all the details that an experienced C programmer eventually will have to master and that an introductory text such as this one does not provide.

Chapter 2

Functions and numbers

Functional programming in SML and imperative programming in C have much in common. The purpose of this chapter is to show that, when given an SML solution to a problem, a C solution can often be found without much difficulty. The C solutions presented here are not necessarily the best from the point of view of their efficiency, but will be a good starting point for the further refinements that are presented in later chapters. In the present chapter, we emphasise the differences between computation as perceived by the mathematician and computation as carried out on computers. We support this by introducing a model of computation, and we illustrate the concepts by solving a number of sample problems.

The first problem that will be solved is to compute the greatest common divisor of two natural numbers. This example discusses in some detail the representation of integral numbers and the C syntax to denote functions. The second example computes the values of arithmetic expressions. It serves to discuss the differences between pattern matching, as it is commonly found in functional languages, and conditionals. The third example calculates an integral power of a real number. It uses floating point numbers. This example is used to discuss the ins and outs of representing reals in C. The fourth example uses higher order functions to compute sums and products. With some extra effort, this powerful abstraction can be used in C programs, albeit in a slightly limited setting.

2.1 A model of computation

A programmer solves a problem by writing a computer program. The answer to the problem is then the result produced by the program. First and foremost, the programmer must understand the problem that is being solved. The programmer must also have an understanding of the way in which computers work, for a program is merely an instruction to the computer to carry out particular computations in a particular order. Without such an understanding, the programmer may attempt to instruct a computer to do things that it cannot do without exceeding the given resources, or worse, the programmer might attempt to make the computer do things that it cannot do at all. Compared to the human brain, computers are rather limited in what they can do. On the other hand, a computer can do certain tasks more quickly and more accurately than the human brain. A *computational model* is a basic understanding of the kind of things that a computer can do and how it does them. We will formulate two such models, one that is applicable when we write SML programs and another that applies when we write C programs.

2.1.1 A computational model for SML programs

An SML program consists of a number of function definitions and a main expression. Computation is the process of finding the value of the main expression. In all but trivial programs, the evaluation of the main expression will cause subsidiary expressions to be evaluated until no more expressions remain whose values are required. The SML programmer needs to know about this computational process for a number of reasons.

Firstly, the process is started with the main expression, so this must be given as part of the program.

Secondly, the main expression will make use of certain functions (either predefined or user defined), so these definitions must be available to the computer. These function definitions may make use of further functions, if so these must be given also. The program is complete only if all required definitions are given.

The SML programmer also needs to know how an expression is evaluated. Assume that an expression consists of a function name and a list of argument values to which that function is applied. The mechanism involved in evaluating the expression consists of four steps:

- The definition of the named function is looked up in the list of known function definitions.

- The formal arguments of the function definition are associated with the values provided by the actual arguments of the function.

- The body of the function is examined to yield a further expression that can be evaluated now. This includes evaluation of the arguments, where necessary.

- As soon as all subsidiary expressions are evaluated, the result of the function is returned. The value of this result only depends on the values of the actual arguments of the function.

The mechanism of starting with a given expression, activating the function it mentions, and looking for the next expression to evaluate comprises the computational model that underlies an implementation of SML. Knowledge of this mechanism enables the programmer to reason about the steps taken whilst the program is being executed. The programmer will have to make sure that only a finite number of such steps are required, for otherwise the program would never yield an answer. In most circumstances, the programmer will also try to make sure that the least number of steps are used to find an answer, as each step takes a certain amount of time.

Reasoning about the behaviour of a program is an important aspect of programming, and it is this computational model that gives the programmer the tool to perform such reasoning.

2.1.2 A computational model for C programs

To some extent it is possible to write C programs in a functional style, such that the computational model that we sketched above for SML is applicable. However, this has two important drawbacks. Firstly, the resulting C programs would be rather inefficient, and secondly, a programmer trained to read and write C programs written in a functional style would not be able to read C programs written by other programmers, as they would be using the language in a way that is not covered by the computational model.

In this chapter, we introduce a simplified computational model for C programs which is almost the same as that for SML. In the next chapter, we will add a model of the store, so as to expand the simple model to a full computational model of C programs.

In the simple computational model, a C program is a list of function declarations. One of the functions should have the name `main`. This function plays the same role as the main expression of an SML program. In the simple model, expressions are evaluated as for SML. The difference is that in C functions consist not of pure expressions but of statements. A statement is an operation on the state of the program, by executing statements in the right order, the program achieves its result. In this chapter we will only use three statements:

- The `return`-statement terminates the execution of a function and returns a result value.

- The expression-statement evaluates an expression, ignoring the result. To contribute to the computation, the expression will often have a *side effect*. A side effect refers to any result which is not returned as the value of the expression but which ends up somewhere else, printing output on the screen for example. Purely functional languages do not allow side effects.

- The `if`-statement conditionally executes other statements.

In the coming chapters, we will gradually introduce all statements of C. We will not use any state in this chapter, the concept of the store is introduced in Chapter 3. The following C program demonstrates a simple program:

```
/* A simple C program, it prints  Hello World  */
int main( void ) {
  printf( "Hello world\n" ) ;
  return 0 ;
}
```

The text between a “`/*`” and a “`*/`” is comment, and ignored by the C compiler. The first statement of the function `main` is:

```
printf( "Hello world\n" ) ;
```

This statement returns no useful value; its sole purpose is to print the text `Hello world` as a side effect. The second statement is:

```
return 0 ;
```

The execution of this statement results in the value `0` to be delivered as the value of the expression `main( )`. The `main` program above shows that the body of a function in C consists of a number of statements. The computational model prescribes that these statements are obeyed in the order in which they are written.

2.1.3 Compiling and executing a C program

Once the C program is written, it needs to be compiled and executed. Here we give a minimal introduction how to compile a program, which is stored in a single file. The compilation of multi module programs is dealt with in Chapter 8. On UNIX systems (or one of its equivalents such as LINUX, AUX, AIX, and so on), the C program must be stored in a file with a name that ends on `.c`. The program can then be compiled by invoking the C compiler, `cc`:

```
cc -o monkey hello.c
```

This will compile the program `hello.c`. Any errors are reported, and if the compiler is satisfied with the code, an executable file, `monkey` in this case, is created. The program can be executed by typing `monkey`:

```
Hello world
```

Newer systems, such as Macintoshes or PC's often offer integrated environments such as the Codewarrior or Borland C. From within the editor, the compiler can be called, and the program can be executed. It is impossible to give an exhaustive description of all these systems, local manuals should be consulted on how to edit, compile and execute programs on these systems.

2.2 Elementary functions

We are now ready to study an interesting algorithm. The greatest common divisor (gcd) of two positive natural numbers is the largest natural number that exactly divides both numbers. The gcd of 14 and 12 is 2, while the gcd of 14 and 11 is 1. The gcd of two numbers is given by the *specification*:

$$\begin{aligned}\text{gcd} &: (I\!N \times I\!N) \rightarrow I\!N \\ \text{gcd}(m,n) &= \max\{d \in I\!N \mid m \bmod d = 0 \wedge n \bmod d = 0\} \qquad (2.1)\end{aligned}$$

The standard *algorithm* for calculating the gcd follows Euclid's method. If, for two positive natural numbers m and n, we have that $m > n$, then the gcd of m and n is defined by:

$$\begin{aligned}\text{euclid} &: (I\!N \times I\!N) \rightarrow I\!N \\ \text{euclid}(m,n) &= \begin{cases} \text{euclid}(n, m \bmod n), & \text{if } n > 0 \\ m, & \text{otherwise} \end{cases} \qquad (2.2)\end{aligned}$$

This can be written directly in SML:

```
(* euclid : int -> int -> int *)
fun euclid m n = if n > 0
                    then euclid n (m mod n)
                    else m ;
```

The following lines show the results of applying the `euclid` function to a set of sample arguments:

```
euclid 14 12   = 2 ;
euclid 14 11   = 1 ;
euclid 558 198 = 18 ;
```

We will now give a 'transformation recipe' for creating a C function from an SML function. Such a recipe, of which we shall present several more, makes it possible to transform SML into C code *systematically*. This is important because, if we start with a tried and tested SML function, then a good C function should result from a systematic transformation. Basically, this means that one has to think only once whilst creating a functional solution, and then follow the recipe carefully to create a good C *implementation*.

To be able to give a transformation recipe, it is necessary to capture the essential aspects of the function to be transformed. Such a capture is called a *schema*. Each recipe will consist of two schemata: one to capture the SML function and one to capture the corresponding C function. The two schemata together can then be used to do the transformation. Here is the SML version of the *function schema*:

```
(*SML function schema*)
  (* $f$ : $t_1$ -> ... $t_n$ -> $t_r$ *)
  fun $f$ $x_1$ ... $x_n$
      = if $p$
           then $g$
           else $h$ ;
```

This schema looks a bit complicated because it tries to cater for as general a class of functions as possible. In particular, the notation $x_1 \ldots x_n$ represents just a single x as well as any particular number n of x-es. The symbols in italic font in the function schema are to be interpreted as follows:

- The symbolic name f stands for the name of the function to be captured. Any concrete function name may be substituted for f, for example `euclid`.

- The number n gives the number of arguments of the function f.

- The $x_1 \ldots x_n$ are the arguments of f.

- The $t_1 \ldots t_n$ are the types of the arguments $x_1 \ldots x_n$ respectively.

- The type of the function result is t_r.

- The expression p is a predicate in $x_1 \ldots x_n$.

- The expressions g and h are expressions involving the values $x_1 \ldots x_n$.

The C version of the *function schema*, which corresponds exactly to the *function schema* in SML above, is:

```
/*C function schema*/
  $t_r$ $f$( $t_1$ $x_1$, ... $t_n$ $x_n$ ) {
    if ( $p$ ) {
      return $g$ ;
    } else {
      return $h$ ;
    }
  }
```

The symbols in this schema are to be interpreted as above, with the proviso that the various syntactic differences between SML and C have to be taken care of in a somewhat ad-hoc fashion.

- Identifiers in SML may contain certain characters that are not permitted in C. These characters must be changed consistently. The requirements for C identifiers are discussed later on page 13.
- The basic types in SML are generally the same as in C, but some must be changed. For example, `real` in SML becomes `double` in C. A list of basic types is given at the end of this chapter.
- Many of the operators that appear in SML programs may also be used in C programs. The operators of C are discussed in Section 2.2.4
- Curried functions are not permitted in C, so the arguments to all function calls must be supplied.

Let us now apply the function schema to the function `euclid`. Below is a *table of correspondence* which relates the schema, the SML program and the C program. The first column of the table contains the relevant symbolic names from the schema. In the second column, we put the expressions and symbols from the SML function. The third column contains a transformation of each of the elements of the SML function into C syntax.

Schema	Functional	C
f:	`euclid`	`euclid`
t_1:	`int`	`int`
t_2:	`int`	`int`
t_r:	`int`	`int`
x_1:	`m`	`m`
x_2:	`n`	`n`
p:	`n > 0`	`n > 0`
g:	`euclid n (m mod n)`	`euclid( n, m%n )`
h:	`m`	`m`

The creation of the C version of euclid is now simply a matter of gathering the information from the third column of the table and substituting that information in the right places in the C function schema. This yields:

```
int euclid( int m, int n ) {
  if( n>0 ) {
    return euclid( n, m%n ) ;
  } else {
    return m ;
  }
}
```

The process of transforming an SML function into C is laborious but not difficult. We shall often tacitly assume that all the steps in the transformation have been made and just present the net result of the transformation.

The euclid example has been chosen because there is a C implementation that is close to the mathematical and the functional versions. The C implementation of euclid is shown below, embedded in a complete program.

```
#include <stdio.h>

int euclid( int m, int n ) {
  if( n>0 ) {
    return euclid( n, m%n ) ;
  } else {
    return m ;
  }
}

int main( void ) {
  printf( "%d\n", euclid( 14, 12 ) ) ;
  printf( "%d\n", euclid( 14, 11 ) ) ;
  printf( "%d\n", euclid( 558, 198 ) ) ;
  return 0 ;
}
```

This program shows the following lexical conventions of C:

- As in SML, the indentation in the program is chosen by the author of the program. The program layout reflects the program structure. A space can be necessary to separate two tokens, but there is no difference between a single space or a number of spaces and newlines.

- Identifiers in C consist of a sequence of letters (the case of the letters is significant), underscores ("_"), and digits. The first symbol of an identifier must be either a letter or an underscore. Valid identifiers are, for example, monkey, Monkey, ___123, or an_id_13.

The C version of the euclid program starts with an *include* directive:

```
#include <stdio.h>
```

This directive tells the compiler to include the text from the file `stdio.h` at this point. This is necessary for the program to be able to use most input and output facilities. The `include` directive and many others are discussed in Chapter 8 (on modules); for now this directive will just be used in every program that is written.

After the `include` directive, two functions are defined: `euclid` and `main`. Each of the functions consists of a function header, defining the type of the function result and the name and type of the function arguments, and a function body, enclosed in curly brackets, defining the behaviour of the function. The function headers, the function body, and the execution of this C program are discussed in turn below.

2.2.1 The header of a C function, types and identifiers

The header of the first C function in the program is:

```
int euclid( int m, int n )
```

The definition of the function `euclid` starts with the definition of the type of the value that the function will return. In this case, the type is `int` which is short for the type of integer numbers. The SML equivalent of this type is also `int`.

After the type of the function, the function name is specified (`euclid`) and then, between parentheses, the types and names of the arguments of the function are given. In this function there are two arguments. The first argument is of type `int` and is named `m`, and the second argument (after the comma) is again of type `int` and is named `n`. The argument names `m` and `n` are used to refer to the first and second argument in the body of the function.

The declaration of `main` is:

```
int main( void )
```

This header declares that the function with the name `main` will return an `int` value. The argument list is specified to be `void`, which means that the function `main` has no arguments.

In C, every function must be declared before it can be used. As the function `main` refers to `euclid`, `euclid` must be defined before `main`. Defining the functions in the other order might work (if the C compiler manages to guess the types of the function correctly), but it is bad practice to rely on this.

There are cases where both functions call each other, so-called mutually recursive functions. In this case, neither function can be defined first. The solution that C offers is the *function prototype*. A prototype is like a type signature in SML: it declares a function and its type, but does not define the internal details. After the prototype is defined, the function can be used. The function can be defined completely later on. A function prototype consists of the header of the function followed by a semicolon. Thus the prototypes of the two functions above are:

```
int euclid( int m, int n ) ;
int main( void ) ;
```

In most functional languages, types are automatically inferred by the compiler. It is good functional programming style to always specify the types of functions, so that the compiler can check the user specified types against its own inferred types. Any mismatch points out errors in the program. In C, the programmer must specify all types.

2.2.2 The body of a C function and its behaviour

After the definition of the types of the function and its arguments, the code of the function is defined. In an imperative language, this code consists of a sequence of *statements*. A statement is a command to perform some kind of operation. As is shown in detail in Section 2.2.3, a function is evaluated by evaluating each statement in turn.

In C, each statement is terminated with a semicolon. A number of statements that are logically one statement can be grouped by enclosing the statement group in curly brackets. The body of a function is a group of statements, enclosed in curly brackets. The function `euclid` contains two kinds of statements: the `return` statement and the `if` statement.

A `return` statement is used to terminate the execution of the function and to calculate the return value of the function. The return value is defined by the expression between the `return` and the semicolon terminating the statement.

An `if` statement has the general form:

```
if( p ) {
  T
} else {
  F
}
```

This can be read as follows: if the conditional expression p evaluates to true, execute the statements T, else execute the statements F. We generally use upper case italic letters to indicate where statements may occur. Upon completion of T or F, any statements following the `if` statement are executed. In the case of `euclid`, there are no statements following the `if` statement. The `if` statement above can also be used without the optional `else` part:

```
if( p ) {
  T
}
```

If the else part is omitted, the C program will just continue with the next statement if the conditional expression p evaluates to false. The curly brackets `{` and `}` are used to group statements. They can be omitted if there is only one statement between them, but we will always use them to make it easier to modify programs later on.

Expressions in C are like functional expressions except that they use a slightly different syntax. For example, the arguments of a function must be separated by commas, and they must be enclosed in brackets. Therefore the function application `f x y` in the functional language is written in C as `f(x,y)`. The operators have also different syntax: for example, $x \bmod y$ is denoted in C as `x%y`.

Most other operators have their usual meaning (a full list is given shortly in Section 2.2.4). This information is sufficient to interpret the body of the function `euclid`:

```
{
  if( n>0 ) {
    return euclid( n, m%n ) ;
  } else {
    return m ;
  }
}
```

If `n` is greater than zero, execute the `then` part:

```
return euclid( n, m%n ) ;
```

Else, if `n` is not greater than zero, execute the else part:

```
return m ;
```

The first of these two statements orders the function to return, more specifically to deliver, the value of the expression `euclid( n, m%n )` as the function's return value. The second of the statements orders to return the value of `m` (in this case, the value of `n` equals `0`).

The keyword `else` of the if statement in this particular function is redundant. Consider the two statements:

```
{
  if( n>0 ) {
    return euclid( n, m%n ) ;
  }
  return m ;
}
```

The first of these, the if statement, will execute the statement below if `n` is greater than zero:

```
return euclid( n, m%n ) ;
```

If the return is executed, the function is terminated without executing any more statements. Thus the second statement, `return m`, is only executed if `n` is equal to zero. We will come back to redundant else statements in the next section.

The body of the `main` function contains 4 statements. The first statement is:

```
printf( "%d\n", euclid( 14, 12 ) ) ;
```

It is a call to the function `printf`. This call has two arguments: the first argument is a string `"%d\n"`, and the second argument is the value that is returned after calling `euclid` with arguments `14` and `12`. As we will see shortly, this call to `printf` will print the value of `euclid( 14, 12 )`

The next two statements in the body of `main` are similar to the first: the second statement prints the greatest common divisor of the numbers `14` and `11`, and the third prints the greatest common divisor of the numbers `558` and `198`. The last statement returns the value `0` to the calling environment of the program. This issue will be discussed in more detail in the next paragraphs.

2.2.3 The execution of a C program

The execution of every C program starts by calling the function with the name `main`. Thus, every C program should have a function called `main`. Apart from the name, the type of the arguments and the type of the return value of `main` are special. `Main` should always return an `int`, and if there are arguments for `main`, they should follow the syntax discussed in Chapter 8, which deals with the environment. For now, all `main` functions will be without arguments and will always return the value `0`. This value will be interpreted by the environment as 'everything all right, the program completed satisfactorily'.

Upon execution of `main`, the first statement of the program has to be executed. In the case of the example program, this is a call to `printf`. The C language requires that all arguments of a function must be evaluated before the function is called, similar to the *strict* evaluation of SML. The values of the arguments are then passed to the function. This mechanism is known as *call by value*. Another way to pass arguments, *call by reference*, is discussed in Chapter 4.

The first argument passed to `printf`, the string `"%d\n"`, is a constant, so it does not need to be evaluated. The second argument is the expression:

```
euclid( 14, 12 )
```

This expression represents a function call that has to be evaluated first. However, before `euclid` can be executed, *its* arguments have to be evaluated. These arguments are constants (the integers `14` and `12`) so `euclid` is executed directly, with `m` having the value `14` and `n` having the value `12`. Eventually when `euclid` returns a value, the `printf` will be invoked.

The first statement of `euclid` tests if `n` is greater than `0`. Since this is true, the `then` statement is executed. This will in turn call the function `euclid` with arguments `12` and `14%12`, which is `2`. Therefore, `euclid` is called again with `m` equal to `12` and `n` equal to `2`.

The first statement of `euclid` tests again if `n` is greater than `0`. This is true, so the `then` statement is executed. This will cause the function `euclid` to be called with arguments `2` and `12%2`, which is `0`. Thus `euclid` is called with `m` equal to `2` and `n` equal to `0`.

The first statement of `euclid` tests again if `n` is greater than `0`. Since this is not true, the `else`-statement is executed, which causes the function to `return m`, which will return the value `2`. This value is returned in turn by each of the previous invocations of `euclid`, and will eventually turn up in the `main` function in which the arguments of `printf` are now fully evaluated to give:

```
printf( "%d\n", 2 ) ;
```

The function `printf` is called, it will print the first argument to the output; however, it handles percent signs in a special manner. Every time `printf` encounters a `%d`, it replaces the `%d` with a decimal representation of the integer that is taken from the argument list. In this example, the value printed is `2`, which was just calculated. The `\n` results in a newline, just like SML. The resulting output is therefore:

```
2
```

This completes the execution of the first statement of `main`. Upon completion of this first statement, the next statement of `main` is executed. Again this is a call to `printf`. The first argument of this function is a constant string, and the next argument is again a call to the function `euclid`, this time with arguments `14` and `11`. `Euclid` is invoked with `m` equal to `14` and `n` equal to `11`. Instead of giving another lengthy explanation of which function calls which function, a trace of function calls can be given as follows:

```
euclid( 14, 11 ) calls
 euclid( 11, 3 )  calls
  euclid( 3, 2 )   calls
   euclid( 2, 1 )   calls
    euclid( 1, 0 )   is
    1
```

The chain of calls will return the value 1, which is printed by the main program.

Exercise 2.1 How many times is the function `euclid` called when executing the third statement of `main`?

2.2.4 Integers

The `euclid` example works with integer numbers, which are captured by the C type `int`. There are many ways to denote integer constants in C. The most common form is to use the decimal representation, as already used before. `13` stands for the integer 13. A minus sign can be used to denote negative values, as in `-13`. A second option is to specify numbers with a different base, either base `8` or base `16`. Hexadecimal values (using base `16`) can be specified by writing them after the letters `0x`. So `0x2C` and `44` both refer to the same number. Octal numbers (with base `8`) should begin with the digit `0`. So `0377`, `0xFF` and `255` all specify the same number.

The operators that can be applied to integers are listed below, together with their SML and mathematical equivalents:

C	SML	Math	Meaning	
`+`			Unary plus	*Unary*
`-`	`~`	$-$	Unary minus	
`~`			Bitwise complement	
`*`	`*`	$\times$	Multiplication	*Binary*
`/`	`div`, `/`	div , /	Division	
`%`	`mod`	mod	Modulo	
`+`	`+`	$+$	Addition	
`-`	`-`	$-$	Subtraction	
`<<`			Bitwise shift left, x<<$s == x * 2^s$	
`>>`			Bitwise shift right, x>>$s == x/2^s$	
`&`			Bitwise and	
`|`			Bitwise or	
`^`			Bitwise exclusive or	

There is one group of operators that does not have an SML equivalent: the bit operators. Bit operations use the two's-complement binary interpretation of the integer values. The `~` operator inverts all bits, the `&`, `|` and `^` operators perform a bit-wise and, or, and exclusive-or operation on two bit patterns. The following are all true:

```
     (~0) == -1,
 (12 & 6) == 4,
 (12 | 6) == 14,
 (12 ^ 6) == 10
```

The `<<` and `>>` operations shift the bit pattern a number of positions: `x<<i` shifts `x` by `i` positions to the left, and `y>>j` shifts `y` by `j` positions to the right. Some examples:

```
 (12 << 6)  == 768,
 (400 >> 4) == 25
```

Bit operations can be supported efficiently because all modern computer systems store integers bitwise in two's-complement form.

The type `int` is only one of a multitude of C types that can be used to manipulate integer values. First of all, C supports the `unsigned int` type. The keyword `unsigned` implies that values of this type cannot be negative. The most important feature of the `unsigned int` type is that the arithmetic is guaranteed to be *modulo-arithmetic*. That is, if the answer of an operation does not fit in the domain, the answer modulo some big power of 2 is used. Repeatedly adding one to an `unsigned int` on a machine with 32-bit integers will give the series 0, 1, 2, $\ldots$ 4294967294, 4294967295, 0, 1, $\ldots$. Modulo arithmetic is useful in situations where longer arithmetic (128-bit numbers, for example) is needed. Other types of integers will be shown shortly, in Section 2.3.3.

2.2.5 Logical operators

The language C does not have a separate type for booleans, instead integers are used to represent booleans. The comparison operators, listed below, result in the integer value `1` or `0`; `1` is used to represent true and `0` is used to represent false.

C	SML	Math	Meaning
`<`	`<`	$<$	Less than
`<=`	`<=`	$\leq$	Less than or equal
`>=`	`>=`	$\geq$	Greater than or equal
`>`	`>`	$>$	Greater than
`==`	`=`	$=$	Equal
`!=`	`<>`	$\neq$	Not equal

C provides the three usual logical operators. They take integers as their operands, and produce an integer result:

C	SML	Math	Meaning
`!`	`not`	¬	Logical not, `!0==1, !x==0 (if x!=0)`
`&&`	`andalso`	∧	Logical and
`||`	`orelse`	∨	Logical or

The `!` operator performs the logical negation (not) operation, `!0` is `1` and `!1` is `0`. In addition the negation operator in C accepts any non zero value to mean true, so `!153` is `0`.

The && operator performs the logical conjunction (and) and the logical disjunction operator (or) is `||`. Some truth relations of these operators are:

```
(0 && x) == 0,
(1 && x) == x,
(0 || x) == x,
(1 || x) == 1
```

The logical && and `||` both accept any non-zero value to mean true, like the `!` operator. The operation of the && and `||` operators is identical to their SML equivalents, but different to their counterparts in some other imperative languages. Consider the following expression:

```
x != 0 && y/x < 20
```

This expression tests first if `x` equals zero. If this is the case, the rest of the expression is not evaluated (because `0&&`... is `0`). Only if `x` does not equal zero, the inequality, `y/x < 20` is evaluated. This behaviour is called *short circuit semantics*. A similar comparison in Pascal would be:

```
(x <> 0) AND (y/x < 20)
```

The semantics of Pascal are such that this leads to disaster if `x` equals zero, as `(y/x < 20)` *is* evaluated regardless of whether `x` equals zero or not.

Do not confuse the operators &&, `||` and == with their single equivalents &, `|` and =. The operators & and `|` are bit-operators, and the single equal sign, =, behaves different from the ==. It will be discussed in the next chapter. Accidentally using a = instead of a ==, results probably in a program that is syntactically correct but with different behaviour. It can be hard to discover this error.

The last logical operator that C supports is the ternary if-then-else operator, which is denoted using a question mark and colon: `c?d:e` means if `c` evaluates to a non-zero value, the value of `d` is chosen, else the value of `e` is chosen.

2.2.6 Defining a type for Booleans, `typedef` and `enum`

The official way to denote the truth values in C is `0` and `1`, but to have a clear structure for a program involving logical operations, it is often convenient to introduce a type for booleans. The following line at the beginning of a program takes care of that:

```
typedef enum { false=0, true=1 } bool ;
```

This line consists of logically two parts. The outer part is

`typedef` T `bool ;`

The keyword typedef associates a type T with a typename, bool in this case. The type can be any of the builtin types (int for example), or user constructed types such as enum { false=0, true=1 }. The keyword enum is used to define an enumerated type. The domain spans two elements in this case, false and true, where false has the numeric value 0 and true has the numeric value 1. The general form of an enumerated type is:

```
  enum { i₀, i₁, i₂ ... }
```

The i_0, i_1, i_2, ... are the identifiers of the domain. The identifiers are represented as integers. The identifiers can be bound to specific integer values, as was shown in the definition of the type bool. The combination of typedef with enum has a counterpart in SML:

```
/*C combination of typedef and enum*/
  typedef enum { i₀, i₁, i₂ ... } t ;
```

```
(*SML scalar datatype declaration*)
  datatype t = i₀ | i₁ | i₂ ... ;
```

The following C fragment defines an enumerated type language:

```
  typedef  enum { Japanese, Spanish, Chinese }  language ;
```

This is equivalent to the following SML declaration:

```
  datatype language = Japanese | Spanish | Chinese ;
```

2.3 Characters, pattern matching, partial functions

The data types integer and boolean have now been discussed at some length, so let us examine another useful data type, the character. The character is a standard scalar data type in many languages, including C. Interestingly, the character is not a standard data type in SML. Instead, SML offers strings as a basic data type. In all languages that offer characters as basic data types, strings occur as structured data types, that is, as data types built out of simpler data types. Structured data types will be discussed in detail in Chapter 4 and later chapters. To overcome the slight oddity of SML we will assume that an SML string containing precisely one element is really a character. The SML type char will be defined as follows:

```
  type char = string ;
```

As an example of a function that works on characters, consider the specification below for evaluating an expression. Suppose an expression that consists of three integers and two operators (either $+$ or $\times$) has to be evaluated. The rules for operator precedence state that multiplication takes precedence over addition, as indicated in the specification below. Here, the set Arepresents all possible characters.

$$\begin{array}{rcl} x, y, z & : & \mathbb{N} \\ \texttt{+}\ ,\ \texttt{*} & : & A \end{array}$$

$$\begin{array}{ccccccl}
x & \texttt{+} & y & \texttt{+} & z & = & (x+y)+z \\
x & \texttt{+} & y & \texttt{*} & z & = & x+(y\times z) \\
x & \texttt{*} & y & \texttt{+} & z & = & (x\times y)+z \\
x & \texttt{*} & y & \texttt{*} & z & = & (x\times y)\times z
\end{array}$$

Using pattern matching on the operators, we can write a function to evaluate an expression with addition and multiplication operators as follows:

```
(* eval : int -> char -> int -> char -> int -> int *)
fun eval x "+" y "+" z = (x + y) + z
  | eval x "+" y "*" z = x + (y * z)
  | eval x "*" y "+" z = (x * y) + z
  | eval x "*" y "*" z = (x * y) * z : int ;
```

The function `eval` takes 5 arguments: 3 integers and 2 characters. The integers are represented as the value of `x`, `y`, and `z`, and the two characters specify which operations to perform. Pattern matching is used to decide which of the four alternatives to choose. Pattern matching leads to clear and concise programs and, as a general rule, one should thus prefer a pattern matching programming style. However, C does not support pattern matching, instead, pattern matching has to be implemented using if statements.

2.3.1 Implementing pattern matching in C

As a preparation to implementing `eval` in C, consider how it could be implemented in SML using conditionals. This is not difficult once the function has been written using pattern matching, as above.

```
(* eval : int -> char -> int -> char -> int -> int *)
fun eval x o1 y o2 z
    = if o1 = "+" andalso o2 = "+"
         then (x + y) + z
         else if o1 = "+" andalso o2 = "*"
                 then x + (y * z)
                 else if o1 = "*" andalso o2 = "+"
                         then (x * y) + z
                         else if o1 = "*" andalso o2 = "*"
                                 then (x * y) * z : int
                                 else raise Match ;
```

The last statement `else raise Match ;` makes explicit what happens if `eval` is applied to a pair of operators other than `"+"` and `"*"`. The specification of `eval`, and the pattern matching of `eval` should have handled this case explicitly. The latter actually does handle the error situation by raising the exception `Match`, but it does so implicitly. We have left this problem to be solved until this late stage to show that now we are actually forced to think about this missing part of the specification: simply omitting the statement `else raise Match ;` will give a syntax error. The problem specification only partially specifies the solution. The function `eval` is a *partial function*, a function which is undefined for some values of its arguments.

The version of `eval` that uses the conditionals can be transformed into a C program using a schema in the same way as the function `euclid` was translated using a schema. However, we need a new schema, since the `eval` function contains a cascade of conditionals, whereas the function schema of Section 2.2 supports only a single conditional. The development of this schema is the subject of Exercise 2.2 below. For now, we will just give the C version of `eval`.

In C, the type `char` is the type that encompasses all character values. Character constants are written between single quotes. The code below is incomplete as we are yet unable to deal with `/*raise Match*/`. We will postpone giving the complete solution until Section 2.3.2, when we are satisfied with the structure of `eval`.

```
int eval( int x, char o1, int y, char o2, int z ) {
  if( o1 ==  +  && o2 ==  +  ) {
    return (x + y) + z ;
  } else {
    if( o1 ==  +  && o2 ==  *  ) {
      return x + (y * z) ;
    } else {
      if( o1 ==  *  && o2 ==  +  ) {
        return (x * y) + z ;
      } else {
        if( o1 ==  *  && o2 ==  *  ) {
          return (x * y) * z ;
        } else {
          /*raise Match*/
        }
      }
    }
  }
}
```

The `else`-parts of the first three if statements all contain only one statement, which is again an if statement. This is a common structure when implementing pattern matching. It becomes slightly unreadable with all the curly brackets and the ever growing indentation, which is the reason why we drop the curly brackets around these if statements and indent it flatly:

```
int eval( int x, char o1, int y, char o2, int z ) {
  if( o1 ==  +  && o2 ==  +  ) {
    return (x + y) + z ;
  } else if( o1 ==  +  && o2 ==  *  ) {
    return x + (y * z) ;
  } else if( o1 ==  *  && o2 ==  +  ) {
    return (x * y) + z ;
  } else if( o1 ==  *  && o2 ==  *  ) {
    return (x * y) * z ;
  } else {
    /*raise Match*/
  }
}
```

This indicates a chain of choices which are tried in order. The final `else`-branch is reached only if none of the previous conditions are true.

Exercise 2.2 The function schema of Section 2.2 supports only a single conditional. Generalise this schema to support a cascade of conditionals, as used by the last SML and C versions of `eval`.

Exercise 2.3 Give the table of correspondence for `eval`, using the function schema of Exercise 2.2.

As discussed before, the `else`'s are redundant in this case because each branch returns immediately. Reconciling this yields another equivalent program:

```
int eval( int x, char o1, int y, char o2, int z ) {
  if( o1 ==  +  && o2 ==  +  ) {
    return (x + y) + z ;
  }
  if( o1 ==  +  && o2 ==  *  ) {
    return x + (y * z) ;
  }
  if( o1 ==  *  && o2 ==  +  ) {
    return (x * y) + z ;
  }
  if( o1 ==  *  && o2 ==  *  ) {
    return (x * y) * z ;
  }
  /*raise Match*/
}
```

During the execution of `eval` in the form above, the four if statements will be executed one by one until one of the if statements matches. It is a matter of taste which of the two forms, the `else if` constructions or the consecutive if statements, to choose. The first form is more elegant since it does not rely on the `return` statement terminating the function. Therefore we will use that form.

The chain of if statements is not efficient. Suppose that `o1` is a `*` , and `o2` is a `+` . The first two if statements fail (because `o1` is not a `+`). The third if statement succeeds, so the value of `(x * y) + z` will be returned. There are two tests on `o1`: it is compared with the character `+` twice, whereas one test should be sufficient. To remove this second test from the code, the program has to be restructured. First, one has to test on the first operator. If it is a `+` , nested if statements will be used to determine which of the two alternatives to evaluate. Similarly, one if statement is needed to check if the first operator is a `*` . In this situation, nested if statements will determine which of the remaining two alternatives to choose. This gives the following definition of `eval`:

```
int eval( int x, char o1, int y, char o2, int z ) {
  if( o1 ==  +  ) {
    if( o2 ==  +  ) {
      return (x + y) + z ;
    } else if( o2 ==  *  ) {
      return x + (y * z) ;
    } else {
      /*raise Match*/
    }
  } else if( o1 ==  *  ) {
    if( o2 ==  +  ) {
      return (x * y) + z ;
    } else if( o2 ==  *  ) {
      return (x * y) * z ;
    } else {
      /*raise Match*/
    }
  } else {
    /*raise Match*/
  }
}
```

Exercise 2.4 Can you devise a translation schema between the pattern matching SML code and the C code that uses a nested if-then-else? If not, what is the problem?

The above program has a functional equivalent, which is not as nice to read, even though we have introduced two auxiliary functions `xymul` and `xyadd` to improve the legibility:

```
(* eval : int -> char -> int -> char -> int -> int *)
fun eval x o1 y o2 z
    = if o1 = "+"
        then xyadd x y o2 z
        else if o1 = "*"
                then xymul x y o2 z : int
                else raise Match

(* xyadd : int -> int -> char -> int -> int *)
and xyadd x y o2 z
    = if o2 = "+"
         then x + y + z
         else if o2 = "*"
                 then x + y * z : int
                 else raise Match

(* xymul : int -> int -> char -> int -> int *)
and xymul x y o2 z
    = if o2 = "+"
         then x * y + z
         else if o2 = "*"
                 then x * y * z : int
                 else raise Match ;
```

It should be the task of the compiler to replace pattern matching with a suitable if-structure. As C does not support pattern matching, it is the task of the C programmer to design an efficient statement structure in the first place.

2.3.2 Partial functions

Functions that use pattern matching or guards are often partially defined. This means that the function is not defined for the entire domain: there are arguments for which the function is undefined. We have seen that the pattern matching version of `eval` is only defined if the operators are in the set { + , * }. An attempt to call the SML function `eval 6 "/" 3 "+" 4` in its pattern matching version will result in a run time error. The program is aborted because none of the definitions of `eval` match the call.

If we were to leave the as yet unfinished statement `else /*raise Match*/` out of our C versions of `eval`, a call to `eval( 6,  / , 3,  + , 4 )` in the C implementation will result in an undefined value. Since none of the if statements match, the end of the function is reached without executing a return statement. Then the function will return with some arbitrary, unknown return value. Continuing the computation with this unknown value makes little sense, it is therefore good practice to prevent this from happening. The solution is to call the function `abort()`, which is available from `stdlib.h`:

```
  /*previous statements*/
  if( o2 ==  *  ) {
    return (x * y) * z ;
  } else {
    abort() ;
  }
} else {
  abort() ;
}
```

A call to `abort()` causes a C program to stop execution immediately, to report a run time error, and often to invoke a debugger or leave a memory dump. Then the programmer can inspect which cases were missed out. The environment will often allow the programmer to detect where the program failed, but it is good practice to call `printf` just before aborting:

```
printf("Arghh, first argument of  eval  is neither") ;
```

```
  printf("a  +  nor an  * ; it is  %c \n", o1 ) ;
  abort() ;
```

A companion of `abort` is the function `exit`. A call to the function `exit` will terminate the program. In contrast with `abort`, the function `exit` stops the program gracefully: `abort` is called to signal a programming error (for example a missing case), `exit` is called to signal a user error, or just to stop the program. The function `exit` has an integer parameter. The value `0`, indicates that the program ran successfully, any other value means that something went wrong. This number has the same role as the number passed as the return value of `main` upon normal program termination.

2.3.3 Differences and similarities between characters and integers

Character constants are denoted between single quotes: the constants `*` and `+` have already been shown before. Most single characters can be denoted in this way, but for some characters a *backslash escape* is needed, like in SML. For example, to denote a single quote, one has to write `\` . The most important backslash escapes are:

`\a`	Bell signal (alarm)
`\b`	Backspace (one character to the left)
`\f`	Form feed (new sheet of paper or clear screen)
`\n`	Newline
`\r`	Carriage return (start again on the same line)
`\t`	Tabulation (goes to next tab stop)
`\v`	Vertical Tabulation (goes to next vertical tab)
`\\`	A backslash (`\`)
`\`	A single quote “ ”
`\"`	A double quote “`"`”
`\0`	NULL-character (value 0)
`\ddd`	`ddd` should be a three digit octal number
`\xdd`	`dd` should be a two digit hexadecimal number

Although the type `char` is a separate type, it is actually part of the family of integers. In C, characters are encoded by their integer equivalent. This means that the character `q` and the integer value 113 are actually identical (assuming that an ASCII encoding of characters is used). Denoting an argument to be of type `char` means that it is a *small* integer constant. The type `char` covers all integers that are necessary to encode the character set.

The fact that the characters are a subset of the integers has a number of consequences. Firstly, functions that convert characters into integers and vice versa, such as the `chr` and `ord` of SML and other languages, are unnecessary. Secondly, all integer operators are applicable to characters. This can be useful as shown by the following equalities, which are all true:

```
a  + 1   ==  b ,
z  -  a  == 25,
9  -  0  == 9
```

However, the following (in)equalities are also true:

```
4  +  5  ==  i ,
4  +  5  !=  9
```

Care should be taken when using the equivalence between characters and integers in C.

To print a character, the function `printf` supports the `%c`-format. As an example, the following statement will print the letter `q` on the output:

```
printf( "%c\n",  q  ) ;
```

No assumptions can be made about the sign of character values. Characters are stored in small integers, but it is up to the system to decide whether signed or unsigned integers are used. This can have consequences as the comparison of two characters `c>d` may be different from test `c-d>0` (as when using unsigned character arithmetic the latter is only false if `c` equals `d`). More about characters, particularly how to read characters from the input, is discussed in Chapter 7.

Exercise 2.5 Given that a machine encodes the character `q` with `113` and the character `0` with `48`, what will the output of the following program be?

```
int main( void ) {
  char c0 =  0  ;
  int  i0 =  0  ;
  char cq = 113 ;
  int  iq = 113 ;
  printf(" %c  = %d,   %c  = %d\n", c0, i0, cq, iq ) ;
  return 0 ;
}
```

To be able to work with different character sets efficiently, C supports a number of standard predicates to classify characters. For example, the predicate `isdigit` can be used to determine whether a character is a digit `0` , `1` , ... `9` . The most important of these predicates are:

`isdigit(c)`	Yields true if `c` is a digit
`isalpha(c)`	Yields true if `c` is a letter
`isupper(c)`	Yields true if `c` is an uppercase letter
`islower(c)`	Yields true if `c` is a lowercase letter
`isspace(c)`	Yields true if `c` is whitespace (newline, tabulation, space)
`isprint(c)`	Yields true if `c` is a printable character

Additionally, two standard functions `toupper` and `tolower` are provided that map a lower case letter to the upper case equivalent, and an upper case letter to the lower case equivalent respectively. To use any of these functions, one must *include* the file `ctype.h` using the include directive:

```
#include <ctype.h>
```

The ins and outs of the include statement are reserved for Chapter 8. An example of the use of these functions is a function that converts a lowercase character to an uppercase letter, and an uppercase character to a lowercase. Here is the SML version:

```
(* makeupper : char -> char *)
fun makeupper c
    = if "a" <= c andalso c <= "z"
         then chr (ord c - ord "a" + ord "A")
         else if "A" <= c andalso c <= "Z"
                 then chr (ord c - ord "A" + ord "a")
                 else c ;
```

Note that the implementation assumes that both the upper and lower case letters are ordered contiguously. The C version replaces the tests on the characters with some of the standard predicates, but the structure of the function is the same as in SML:

```
char makeupper( char c ) {
  if( islower( c ) ) {
    return toupper( c ) ;
  } else if( isupper( c ) ) {
    return tolower( c ) ;
  } else {
    return c ;
  }
}
```

2.4 Real numbers

Characters and integers are sufficient to solve problems of a symbolic nature. For solving numerical problems, it is often desirable to use real numbers. To explain the use of real numbers in C, an algorithm that calculates r^p is discussed. In this example, r can be any real number, but p must be a non-negative integer. The specification of the power operator is as follows:

$$\begin{aligned} \cdot^{\cdot} &: (\mathbb{R} \times \mathbb{N}) \rightarrow \mathbb{R} \\ r^p &= \prod_{i=1}^{p} r \end{aligned} \tag{2.3}$$

The operator $\prod$ is like $\sum$, except that it performs multiplications instead of additions. The 'Indian' algorithm to calculate this product is based on the following observation. Expanding the product above results in the following equation:

$$r^p \quad = \quad \overbrace{\underbrace{r \times r \cdots r}_{p \text{ div } 2} \times \underbrace{r \times r \cdots r}_{p \text{ div } 2}(\times r)}^{p}$$

There are p multiplications which are grouped into two groups of p div 2 multiplications, and, if p is odd, one single multiplication. This group of p div 2 multiplications yields by definition $r^{p \text{ div } 2}$, which gives the following recursive definition of the power operator:

$$\begin{array}{rcl}
r & : & I\!R \\
p & : & I\!N \\
r^p & = & \left\{ \begin{array}{rl} 1, & \text{if } p = 0 \\ r \times r^{p-1}, & \text{if } p \text{ is odd} \\ \text{sqr}(r^{p \text{ div } 2}), & \text{otherwise} \end{array} \right. \\
 & & \text{where} \quad \text{sqr}(x) = x \times x
\end{array} \qquad (2.4)$$

This reads as follows: any number to the power 0 equals 1; any number to the power an-odd-number is equal to the number times the number to the power the-odd-number-minus-1; and any number to the power an-even-number equals the square of the number to the power the-even-number-divided-by-2.

Exercise 2.6 The original definition of r^p (2.3) would require 127 multiplications to calculate r^{128}. How many multiplications are necessary using the second definition (2.4)?

Exercise $\star$ **2.7** Prove by induction on p that the two definitions of power, (2.4) and (2.3), are equivalent.

Equation (2.4), defining `power` and its auxiliary functions `square` and `odd`, would be formulated in SML using conditionals as shown below:

```
(* square : real -> real *)
fun square x : real = x * x ;

(* odd : int -> bool *)
fun odd x = x mod 2 = 1 ;

(* power : real -> int -> real *)
fun power r p
    = if p = 0
         then 1.0 (* Note, this is a real! *)
         else if odd p
                 then r * power r (p-1)
                 else square (power r (p div 2)) ;
```

The following table gives the answers when applying the function `power` to a sample set of arguments:

```
power 3.0       0 =   1.0 ;
power 5.0       4 = 625.0 ;
power 1.037155 19 =   1.99999 ;
```

The C equivalents of these functions and three example computations are shown below as a complete C program. The general function schema of Exercise 2.2 has been used to transform the SML functions into C.

```
#include <stdio.h>

typedef enum { false=0, true=1 } bool ;

double square( double x ) {
  return x * x ;
}

bool odd( int x ) {
  return x % 2 == 1 ;
}

double power( double r, int p ) {
  if( p == 0 ) {
    return 1.0 ;
  } else if( odd( p ) ) {
    return r * power( r, p-1 ) ;
  } else {
    return square( power( r, p/2 ) ) ;
  }
}

int main( void ) {
  printf( "%f\n", power( 3, 0 ) ) ;
  printf( "%f\n", power( 5, 4 ) ) ;
  printf( "%f\n", power( 1.037155, 19 ) ) ;
  return 0 ;
}
```

The power program contains 4 functions: `square`, `odd`, `power` and `main`. The type `bool` has been declared, as in Section 2.2.6, so that the function `odd` is typed properly.

The function `square` results in a value of type `double`, which is an abbreviation for a double precision floating point number. Floating point numbers can store fractional numbers and, in general, a wider range of numbers than integers. The normal arithmetic operations, `+`, `-`, `*`, and `/`, can be performed on doubles. The ins and outs of floating point numbers are discussed later in this chapter. To print a floating point number, the function `printf` must be informed that it must print a floating point style number. To accomplish this, the format %f must occur in the format string. The function printf does not check whether the types in the format string correspond with the types of the arguments. If printf is asked to print something as a floating point number (with a %f format) but is supplied an integer argument, then the program will print in the best case an arbitrary number and in the worst case will crash.

The execution of the power program is similar to the execution of the previous programs. It starts in main, which in turn calls power, before calling printf to print the first result.

Exercise 2.8 How many times are the functions square and power called when executing the third statement of main?

2.4.1 Coercions of integers and floating point numbers

Constants of the type double are denoted in scientific notation with a decimal point and an e preceding the exponent. For example 3.14159 is the value of π to 6 digits. The speed of light in metres per second is 2.9979e8. Floating point and integer constants are distinguished by the presence of the decimal point and/or exponent. The constant 12 denotes the integer twelve, but 12.0, 12e0 and 1.2e1 denote the floating point number twelve.

In C, doubles and integers are converted silently into each other when appropriate. This is called *coercion*. The arithmetic operators are overloaded: applied to two doubles, these operators deliver a double; applied to two integers, the result is an integer; and applied to one integer and one double, the integer is coerced to a double, and the result is also a double. Although this comes in handy in many cases (the constant 2 of type int is coerced into the constant 2.0 of type double whenever that is necessary), it can lead to unexpected surprises. Here are a number of typical examples:

```
1.0 + 5.0/2.0
```

In both C and SML, the value of this expression will be 3.5. In C, the type of the expression is double; in SML it is real. The same expression with an integer 5 instead of the real 5.0 is:

```
1.0 + 5/2.0
```

When interpreted as a C expression, the result will be the double 3.5. In this case, because one of the operands of the division, 2.0, is a double, the other operand is coerced to a double. This means that a floating point division and not an integer division is required. Thus the overloading of the division operator is resolved towards a floating point division. The result of the division, 2.5, is added to 1.0 to yield the double 3.5 as the total result of the expression. SML does not accept this expression, since in SML the division operator / works exclusively on reals. The SML system will complain about a type conflict between the operands 2.0 and 5 and the operator /.

```
1.0 + 5/2
```

When interpreted as a C expression, the result will be the double `3.0`. The division operator is applied to two integers, hence the result of the division is the integer `2`. Since one of the operands of the addition is a double, `1.0`, the integer `2` will be coerced into the double `2.0` so that the result of the expression as a whole will be the double `3.0`. When interpreted as an SML expression, the same type error arises as in the previous case.

```
1 + 5/2
```

As before, the operator `/` applied to two integers will be interpreted in C as integer division. Thus, the result of the integer division is the integer `2`. The operands of the addition will both be integers, so that the value of the expression as a whole will be the integer `3`. When interpreted as an SML expression, the same type error arises as in the previous two cases, this time it can be resolved by using a `div` operator.

Internally, floating point numbers are stored in two parts: the fraction (or mantissa) and the exponent. If these are m and e, the value of a floating point number is given as $m \times 2^e$. For example, the number 0.375 is stored as mantissa 0.75 and exponent -1. Only a limited number of bits are available to store the exponent and the mantissa. Let us consider, as an example, the layout of a floating point number in a word of only 8 bits. We have opted for a three bit exponent and a five bit mantissa. This arrangement is schematical (the real representation is different, consult a book on computer architecture for full details):

exponent			mantissa				
e_2	e_1	e_0	m_4	m_3	m_2	m_1	m_0

The bits are numbered such that the least significant bit is the rightmost bit of each of the mantissa and the exponent.

We assume that the exponent stores two's-complement numbers. The bit pattern for -1 is thus 111. The mantissa is arranged such that the most significant bit m_4 counts the halves, the next bit m_3 counts the quarter, and so on. Therefore, the bit pattern for 0.75 is 11000. The representation of 0.375 as an 8-bit floating point number is thus as shown below:

exponent			mantissa				
1	1	1	1	1	0	0	0

Typically, the exponent has a range of $10^{\pm 300}$, and the mantissa has around 15 digits precision. A consequence of the limited precision of the mantissa is that the floating point numbers are a subset of the real numbers. Hence, two floating point numbers can be found in C, say `x` and `y`, such that `x` does not equal `y`, and there does not exist a floating point number between these two numbers. As an example, in one C implementation there is no floating point number between:

```
1.000000000000000222044604925031
```

and:

```
1.000000000000000444089209850062
```

or between:

```
345678912.0000000596046447753906
```

and:

```
345678912.0000001192092895507812
```

The floating point numbers actually step through the real numbers, albeit in small steps. The actual step size depends on the machine and compiler. This stepping behaviour has a number of consequences. Firstly, many numbers cannot be represented exactly. As an example, many computer systems cannot represent the number 0.1 exactly. For example, the nearest numbers are:

```
0.0999999999999999916733273153113
```

and:

```
0.1000000000000000055511151231257
```

For this reason floating point numbers should be manipulated with care: `0.1*10` is not necessarily equal to `1.0`. On some systems, it is greater than 1; on others, it is less than 1. An equality test on floating point numbers, for example `a==0.1`, will probably not give the expected result either.

2.5 Functions as arguments

The functions discussed so far operate on basic values like integers, floating point numbers, or characters. Functions can also have a function as an argument. Functions with functions as arguments are called *higher order* functions (as opposed to first order functions that operate on basic values). Higher order functions are a powerful tool to the functional programmer. They can also be used in C but require some extra effort on the part of the programmer.

2.5.1 Sums

A common operation in mathematics, statistics, and even in everyday life is the summation of a progression of numbers. We learn at an early age that the sum of an arithmetic progression is:

$$1 + 2 + 3 + \ldots n \quad = \quad \sum_{i=1}^{n} i \;=\; \frac{n \times (n+1)}{2} \tag{2.5}$$

Another useful sum, which is not so well known, adds successive odd numbers. This sum can be used to calculate the square of a number:

$$1 + 3 + 5 + \ldots (2 \times n - 1) \quad = \quad \sum_{i=1}^{n} (2 \times i - 1) \;=\; n^2 \tag{2.6}$$

It is interesting to look at the differences and similarities of these sums. The differences are that they may have other begin and end values than 1 and n (although in our two examples both summations range from 1 to n), and that they may have different expressions with i. The similarity is that the value of i ranges over a progression of numbers, thus producing a sequence of expressions, each of which uses a different value of i. The values of these expressions are added to produce the end result. The commonality in these patterns of computation can be captured in a higher order function. The differences in these patterns of computation are captured in the arguments of the higher order function. We meet here an application of the important principle of procedural abstraction: look for common aspects of behaviour and capture them in a procedure or function, whilst allowing for flexibility by using arguments. The `sum` function shown below should be regarded as the SML equivalent of the mathematical operator $\sum$:

```
(* sum : int -> int -> (int -> real) -> real *)
fun sum i n f = if i > n
                   then 0.0
                   else f i + sum (i+1) n f ;
```

The function `sum` takes as arguments a begin value (argument `i`) and an end value (argument `n`). Both of these are integers. As its third argument, we supply a function `f` which takes an integer (from the progression) and produces a real value to be summed.

The sum of an arithmetic progression is computed by the SML function `terminal` below. It uses `sum` to do the summation. The actual version of `f` is the function `int2real`. It is a function that converts an integer into a real number, so that the types of the function `sum` and its arguments are consistent.

```
(* terminal : int -> real *)
fun terminal n = let
                   fun int2real i = real i
                 in
                   sum 1 n int2real
                 end ;
```

To see that this function does indeed compute the sum of an arithmetic progression we could try it out with some sample values of `n`. A better way to gain understanding of the mechanisms involved is to expand the definition of `sum` by substituting the function `int2real` for `f`. This yields:

```
(* terminal : int -> real *)
fun terminal n
    = let
         fun int2real i = real i
         fun sum  i n = if i > n
                          then 0.0
                          else int2real i + sum  (i+1) n
      in
         sum  1 n
      end ;
```

As an example, we can now ‘manually’ evaluate `terminal 3`. This requires a series of steps to calculate the answer `6`, according to the computational model for

SML:

```
terminal 3 = sum  1 3
           = int2real 1 + sum  (1+1) 3
           = 1.0 + sum  2 3
           = 1.0 + ( int2real 2 + sum  (2+1) 3 )
           = 1.0 + ( 2.0 + sum  3 3 )
           = 1.0 + ( 2.0 + ( int2real 3 + sum  (3+1) 3 ) )
           = 1.0 + ( 2.0 + ( 3.0 + sum  4 3 ) )
           = 1.0 + ( 2.0 + ( 3.0 + 0.0 ) )
           = 6.0
```

After these preparations, we are now ready to translate our functions `sum` and `terminal` into C. Here is the C version of the `sum` function:

```
double sum( int i, int n, double (*f) ( int ) ) {
  if( i > n ) {
    return 0.0 ;
  } else {
    return f( i ) + sum( i+1, n, f ) ;
  }
}
```

The `sum` function uses the function `f` as its third argument with the following syntax:

```
double (*f)( int )
```

This means that the argument `f` is itself a function which will return a double value and which requires an integer value as an argument. (Strictly speaking `f` is a *pointer* to a function, for now the difference can be ignored; pointers are discussed in Chapter 4) The argument `f` is used by `sum` as a function to calculate the value of `f( i )`. The general syntax for declaring a functional argument of a function is:

t_r (*f)(t_1, ... t_n)

Here t_r gives the type of the return value, and $t_1, \ldots t_n$ are the types of the arguments of the function. (In turn, each of $t_1, \ldots t_n$, can be a function type.) The SML equivalent of the type of this higher order function is:

(t_1 -> ... t_n -> t_r)

A type synonym can be defined for function types using the `typedef` mechanism of C. As an example, define:

```
typedef double (*int_to_double_funcs)( int ) ;
```

This represents a type with the name `int_to_double_funcs` which encompasses all functions that have an `int` as an argument and result in a `double` value. This type synonym would have been declared in SML as:

```
type int_to_real_funcs = int -> real ;
```

Exercise 2.9 Give the table of correspondence for the transformation of the SML version of sum into the C version. Use the function schema from the beginning of this chapter.

The transformation of terminal into C poses a slight problem. C does not allow to define local functions, like the function int2real. Instead, int2real has to be defined before terminal. It would have been better if the auxiliary function int2real could have been declared locally to terminal, as it is only used within that function. Most other programming languages do permit this, Pascal or Modula-2 for example.

```
double int2real( int i ) {
  return (double) i ;   /* return i would suffice */
}

double terminal( int n ) {
  return sum( 1, n, int2real ) ;
}
```

To stay close to the SML version, the function int2real uses an explicit *type cast*: the expression (double)i coerces the expression i to the type double. In general, (type) expr coerces expr to the specified type. As type casts are highly undesirable in C (more on this in later chapters), it is better just to write return i, and leave the coercions to the compiler.

Exercise 2.10 Use Equation (2.6) to define an SML function square. Then transform this function into C.

Exercise 2.11 The value of π (≈ 3.14159) can be approximated using the following formula:

$$\begin{array}{rcl} \pi & : & I\!R \\ \frac{\pi}{4} & = & 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots \end{array} \tag{2.7}$$

Give an SML function nearly_pi which calculates an approximation to π using the sum of the first 2*n terms of the propagation given by (2.7). Pass n as the argument to nearly_pi. Transform your function into C.

2.5.2 Products

The functions used in the previous section were all defined in terms of sums. Similar functions can be given in terms of products. Here is the product function in SML:

```
(* product : int -> int -> (int -> real) -> real *)
fun product i n f = if i > n
                        then 1.0
                        else f i * product (i+1) n f ;
```

With this new function product, we can define the factorial function in the same way as terminal was defined using sum:

```
(* factorial : int -> real *)
fun factorial n = let
                    fun int2real i = real i
                  in
                    product 1 n int2real
                  end ;
```

Exercise 2.12 Give the C versions of the product and factorial functions above.

Exercise 2.13 The value of e (≈ 2.71828) can be approximated using the following formula (! is the factorial function):

$$\begin{aligned} e &: \mathbb{R} \\ e &= 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots \end{aligned} \tag{2.8}$$

Give an SML function nearly_e which calculates an approximation to e using the sum of the first n terms of the propagation given by (2.8), where n is the argument to nearly_e. Transform your function into C.

The sum and product functions are actually similar. Besides having a different name, The product function differs from the sum function in only two aspects:

1. The product function multiplies expressions where sum adds expressions.

2. The product function uses the unit element 1.0 of the multiplication where sum uses the unit element 0.0 of the addition.

The sum and product functions are so similar that it seems a good idea to try to generalise further. The common behaviour of sum and product is a repetitive application of some function. This repeated behaviour requires a base value (0.0 for sum and 1.0 for product) and a method of combining results (+ for sum and * for product). The common behaviour can be captured in a rather heavily parametrised SML function repeat:

```
(* repeat :  a -> ( b ->  a ->  a) ->
           int -> int -> (int ->  b) ->  a *)
fun repeat base combine i n f
    = if i > n
        then base
        else combine (f i) (repeat base combine (i+1) n f) ;
```

The repeat function has two functional arguments, combine and f. The best way to understand how it works is by giving an example using repeat. Let us redefine the function sum in terms of repeat:

```
(* sum : int -> int -> (int -> real) -> real *)
fun sum i n f
    = let
         fun add x y = x + y
      in
         repeat 0.0 add i n f
      end ;
```

A succinct way to characterise the way in which repeat has been specialised to sum is by observing that the following functions are equivalent:

```
sum = repeat 0.0 add
```

We have omitted the arguments i, n, and f as they are the same on both sides. With repeat, we have at our disposal a higher order function which captures the repetitive behaviour of sum, product, and many more functions. The repeat function derives its power from the wide range of functions and values that can be supplied for its five arguments.

Exercise 2.14 Redefine the SML function product in terms of repeat.

Exercise 2.15 Transform repeat and the redefined version of sum into C functions.

Exercise 2.16 Instead of using addition or multiplication, we might consider division as a method of combining results from the repeated function. This can be used to compute so-called continued fractions. The golden ratio, in mathematics prosaically known as the number ϕ ($= \frac{1+\sqrt{5}}{2} \approx 1.61803$), can be computed as follows:

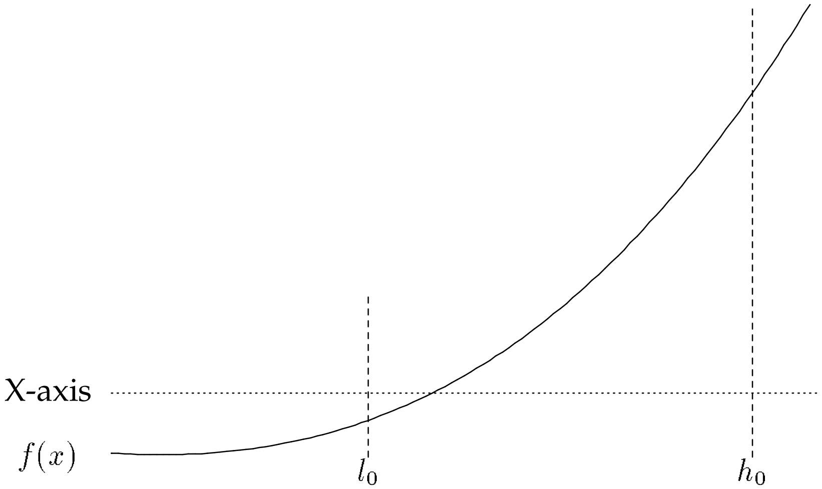
$$\begin{array}{rcl}\phi & : & I\!R \\ \phi & = & 1 + \cfrac{1}{1+\frac{1}{1+\frac{1}{1+\dots}}}\end{array} \qquad (2.9)$$

Give an SML function nearly_phi using the function repeat to calculate an approximation to ϕ. You should use the first n terms of the continued fraction given by (2.9); make n an argument of nearly_phi. Transform your function into C.

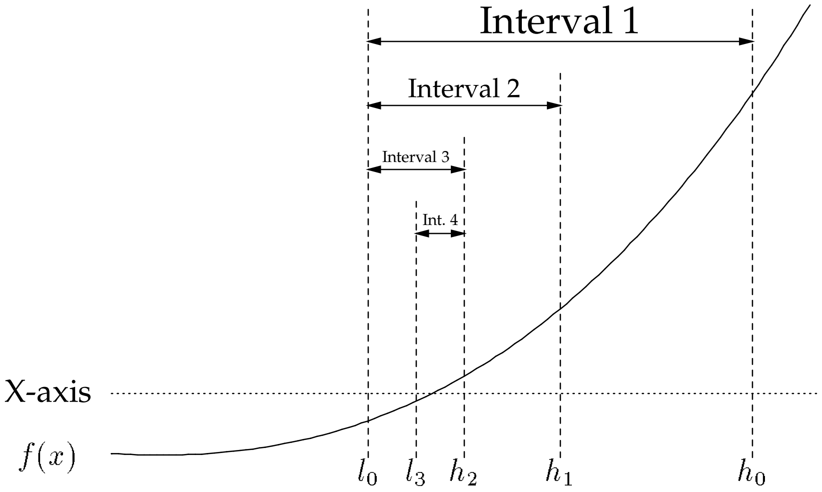
2.5.3 An extended example of higher order functions: bisection

An interesting algorithm that can be implemented with higher order functions is the *bisection method*. Bisection is the process of reducing a larger problem to a smaller problem until the problem is so small that the solution is trivial (it is an example of the important class of *divide and conquer* algorithms). The bisection method can be applied to search for some particular information in a mass of data. In this example, we use bisection to find the *root* of a function, the x-value for which the function value is 0.

Before we present the details of the bisection method, let us have a look at its graphical interpretation. The picture below shows the graph of a function $f(x)$. This function intersects the X-axis at a point between the lines marked l_0 (l for low) and h_0 (h for high). These two points form the boundaries of an interval of the X-axis which contains the root. The interval is rather large so it is not a good approximation to the root. The smaller we can make the interval, the better the approximation to the root becomes. The bisection method calculates a series of approximations, with each subsequent one being better than the previous. The process is stopped when we are sufficiently close to the root.



When given two initial boundaries l_0 and h_0, the bisection method calculates a new point m (for mid) exactly between l_0 and h_0. It then checks whether the root resides to the left or to the right of the point m. If the root resides to the left of m, the right half of the original interval between l_0 and h_0 is abandoned, and the process continues with a new interval between l_0 and $h_1 = m$. This new interval is half the size of the original interval. If the root resides to the right of m, the left half is abandoned, and the process continues with the interval between $l_1 = m$ and h_0. This interval is also half the size of the original interval. The process of halving the current interval continues until it is deemed small enough to serve as an approximation to the root. Here is a graphical rendering of the halving process:



Summarising, when given some initial estimates l_0 and h_0, the bisection method finds two points l_n and h_m, which are as near as possible to either side of the intersection point. It takes max(m, n) steps to reach this result.

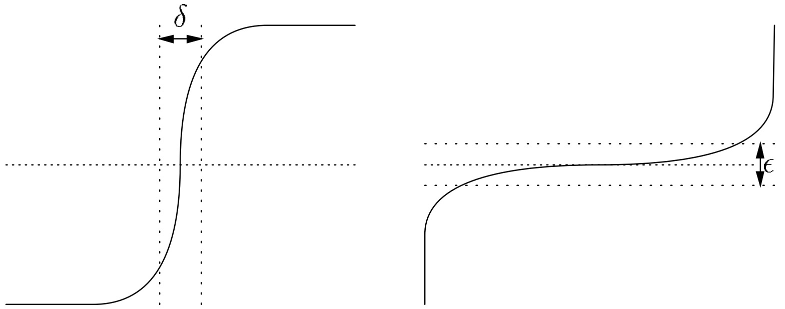
To state more accurately what 'as near as possible' actually means, we will have to use some mathematical notation. For the bisection method to work properly there must be two points given, l and h, such that $f(l) < 0 < f(h)$. If f is a continuous function, there must be a root of f between l and h. The bisection method tries to find such a root by calculating the arithmetic mean of l and h as $m = \frac{l+h}{2}$. Given a particular set of values $l < m < h$, there are three possibilities:

- If $f(m) \approx 0$, the root has been found (or at least we are close enough to the root).
- If $f(m)$ is positive, we need to move left and continue to use m as the new value of h.
- If $f(m)$ is a negative number, we continue to use m as the new value of l.

The bisection method thus closes in on the root, either by moving l up or by moving h down. The method is careful not to move either of these values past the root. In mathematical notation we write:

$$\begin{aligned}
f &: \mathbb{R} \rightarrow \mathbb{R} \\
\text{bisection} &: (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R} \\
\text{bisection}(l, h) &= \begin{cases} m, & \text{if } |f(m)| < \epsilon \\ m, & \text{if } |h - l| < \delta \\ \text{bisection}(m, h), & \text{if } f(m) \leq -\epsilon \\ \text{bisection}(l, m), & \text{otherwise} \end{cases} \\
& \quad \text{where } \; m = \frac{l+h}{2}
\end{aligned} \tag{2.10}$$

To express that we are close enough to the root, the bisection method uses two margins, ϵ and δ. We say that we are close enough to a root if either the function result is close to 0, $|f(m)| < \epsilon$, or if the root is between two bounds that are close enough, $|h - l| < \delta$. Below, this is visualised:



We can choose ϵ and δ so that the root is calculated with sufficient precision. Note that there is a problem if they are both set to zero. Regardless of how close l and h are, they will never actually meet. Each step brings them closer by halving the size of the interval. Only infinitely many steps could bring them together. Secondly, as was shown in the previous section, computers work with limited precision arithmetic only. This might cause unexpected problems with the implementation of floating point numbers. By accepting a certain error margin, both these problems are avoided.

The above version of the bisection method requires f to be an increasing function, $f(l) < 0 < f(h)$; it does not work for decreasing functions.

The mathematics of (2.10) can be transformed directly into an SML function. Here are the constants `eps` and `delta` which represent our choice for the values of ϵ and δ. Making one of them (or both) larger reduces the accuracy of the method but also causes it to find an approximation to the root faster. Conversely, making the values smaller slows down the process but makes it more accurate.

```
(* eps,delta : real *)
val eps = 0.001 ;
val delta = 0.0001 ;
```

The function `bisection` below follows the structure of the mathematics. To allow for some flexibility, we have given `bisection` three arguments. The first argument is the function `f`, whose root we are looking for. The other two arguments are the real bounds `l` and `h` that represent the current low and high bounds of the interval in which the root should be found.

```
(* bisection : (real->real) -> real -> real -> real *)
fun bisection f l h
    = let
         val m = (l + h) / 2.0
         val f_m = f m
      in
         if absolute f_m < eps
            then m
            else if absolute(h-l) < delta
                    then m
                    else if f_m < 0.0
                            then bisection f m h
                            else bisection f l m
      end ;
```

The local declarations for m and f_m ensure that no calculation is performed more than once. This makes the bisection function more efficient than a literal translation of the mathematics. The function absolute is defined as follows:

```
(* absolute : real -> real *)
fun absolute x = if x >= 0.0
                    then x
                    else ~x ;
```

As an example of using the bisection function, let us calculate the root of the parabola function below.

```
(* parabola : real -> real *)
fun parabola x = x * x - 2.0 ;
```

The following table gives the roots of the parabola function when calculated with different initial bounds. The chosen value of eps is 0.001, and the chosen value for delta is 0.0001. In each of these cases, parabola l < 0 < parabola h.

```
bisection parabola 1.0   2.0 = 1.41406 ;
bisection parabola 0.0 200.0 = 1.41449 ;
bisection parabola 1.2   1.5 = 1.41445 ;
```

The three answers are all close to the real answer $\sqrt{2} = 1.41421356\ldots$. The answers are not exactly the same because bisection computes an approximation to the answer.

A complete C program implementing and using the bisection method is given below:

```
#include <stdio.h>

double absolute( double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}

const double eps=0.001, delta=0.0001 ;

double bisection( double (*f)( double ),
                  double l, double h  ) {
  const double m = (l + h ) / 2.0 ;
  const double f_m = f( m ) ;
  if( absolute(f_m) < eps ) {
    return m ;
  } else if( absolute(h-l) < delta ) {
    return m ;
  } else if( f_m < 0.0 ) {
    return bisection( f, m, h ) ;
  } else {
    return bisection( f, l, m ) ;
  }
}

double parabola( double x ) {
  return x * x - 2.0 ;
}

int main( void ) {
  printf( "%f\n", bisection( parabola, 1.0,   2.0 ) ) ;
  printf( "%f\n", bisection( parabola, 0.0, 200.0 ) ) ;
  printf( "%f\n", bisection( parabola, 1.2,   1.5 ) ) ;
  return 0 ;
}
```

The C program for the bisection method includes the declaration of the two doubles `eps` and `delta`:

```
  const double eps=0.001, delta=0.0001 ;
```

This statement declares two *global constants*, that is, constants which are available to all functions declared in the current module. Note that two declarations, declaring them both to have type `double`, may be placed on one line. Two *local* constants are declared in the body of the function `bisection`:

```
  const double m = (l + h ) / 2.0 ;
  const double f_m = f( m ) ;
```

These local constants are only visible within the body of the function `bisection`. As any definition in C, every constant must be declared before the value can be used. Since the definition of `f_m` uses the value of `m`, the definition of `f_m` must come after the definition of `m`, so that `m` can be used to calculate `f_m`.

Exercise 2.17 Generalise the function schema of Exercise 2.2 to support an arbitrary number of local definitions as well as a cascade of conditionals.

Exercise 2.18 The output of the bisection program consists of three different approximations of the root. Trace the first two calls to `printf` to see why the approximations are different.

Exercise 2.19 Reformulate (2.10) so that it works for functions with a negative gradient. Develop a general bisection function that automatically chooses between the version that only works for a positive gradient and the version that also works for a negative gradient.

2.6 Summary

The following C constructs were introduced:

Functions The general form of a C function declaration is:

```
t_r f( t_1 v_1, ... t_n v_n ) {
  /*constant declarations*/
  /*statements*/
}
```

Each function contains declarations of local values, and a series of statements. When the function is called, the local constants will first be evaluated (in the order in which they are written), whereupon the statements are executed, in the order specified.

Constant declarations A constant c with type t and value e is declared as follows:

```
const t c = e ;
```

Statements Three forms of statements were discussed, the `return` statement, the `if` statement and the function-call statement. A return statement has the following form:

```
return e ;
```

It will terminate the execution of the current function, and return the value of the expression e as the return value of the function. The type of the expression should match the result type of the function. An `if` statement comes in two flavours, one with two branches, and one with only a then-branch:

```
if( p ) {                          if( p ) {
  T                                  T
} else {                           }
  F
}
```

Depending on the value of the conditional expression p, the first or the second (if present) series of statements, T or F, will be executed. The third statement discussed here is the expression statement, which consists simply of an expression followed by a semicolon. This statement will evaluate the expression and discard the result. This can be useful, as the expression can have a side effect. For example, calling the function `printf` will cause output to be printed.

Printing output The function printf requires a string as its first argument, printing it while substituting %-abbreviations with the subsequent arguments as follows:

`%d`	Expects an integer argument, prints it decimally
`%f`	Expects a floating point argument, prints it decimally
`%c`	Expects a character argument, prints it as a char
`%%`	Prints a single percent sign.

Expressions Expressions in C are like SML expressions, except that some of the operators are slightly different, as listed in Sections 2.2.4 and 2.2.5. Expressions are typed, but there are fewer types than usual: booleans and characters are represented by integers.

Types The following table relates C types to their SML and mathematical equivalents:

C type	SML	Mathematics
`int`	`int`	$\subset \mathbb{Z}$
`char`		A= { `a` , `b` ,... `!` , `@` ,...}
`double`	`real`	$\subset \mathbb{R}$

Extra care has to be used when using characters (which are actually integers, page 27). Booleans are also represented by integers and must be defined explicitly, as shown in Section 2.2.6.

Naming types : A name can be associated with a type using the keyword `typedef`:

`typedef` t i `;`

This will bind the identifier i to the type t. For some types, most notably function types, i must appear in the middle of t:

`typedef` t_r `(*`i`)(`t_1`, ...` t_n`) ;`

This defines a function type with name i identical to the SML type:

`type` i `=` t_1 `-> ...` t_n `->` t_r `;`

Enumerated types The general form of an enumerated type is:

`typedef enum {` i_0`,` i_1 `... }` t `;`

The enum defines a type by explicitly enumerating all possible values: i_0, i_1, ... The `typedef` causes the enumeration type to be known as t.

Main Each C program must have a `main` function. The function with the name `main` is the function that is called when the program is executed.

The most important programming principles that we have addressed in this chapter are:

- If we can convince ourselves that a functional solution to a problem satisfies its specification, the systematic transformation of the functional solution into a C implementation should then give a reasonable guarantee that the C solution also satisfies the specification. The guarantee is not watertight, as the transformations are informal.

- If possible a function should be pure. That is, the function result should depend only on the value of its arguments. The systematic transformation method introduced in this chapter guarantees that all C functions are indeed pure.

- A function should be total, that is, it should cover all its cases. The SML compiler will usually detect if a function is partially defined. In C, a runtime error might occur or the program might return a random result, if the function is incomplete. By first writing a function in SML, we obtain appropriate warnings from the compiler, and if all such warnings are taken to heart, our C programs will not have this problem. The language C does not have built-in support for pattern matching, but a chain of `if` statements can be employed to simulate pattern matching.

- A function should capture a common, useful behaviour. Variations in its behaviour should be possible by supplying different argument values. Common variations should be created as separate functions using specialised versions of more general behaviours.

- Functions should be strongly typed. Since SML is a strongly typed language, it forces us to write strongly typed functions. Our systematic approach to transforming SML programs into C programs should carry over the typing discipline from the SML code into the C code. The implementation of polymorphic functions is explained later, in Chapters 4 and 8.

- Computers do not always follow the rules of mathematics. We have shown a number of cases where standard mathematical laws do not apply to the data that is used in computers. The representation of reals in a computer is particularly troublesome. The programmer should be aware of the approximative nature of data in programs.

One vitally important issue that we have not addressed in this chapter is the efficiency of the C implementations. This is the subject of the next chapters.

2.7 Further exercises

Exercise 2.20 Write a program that converts a temperature from Centigrade to Fahrenheit: $0^{\circ}C = 32^{\circ}F$, and $10^{\circ}C = 50^{\circ}F$. Test it on 0, 28, 37 and 100 degrees Centigrade.

Exercise 2.21 Some computers offer a ‘population count’ instruction to count the number of bits in an integer that are set to 1; for example, the population count of 7 is 3 and the population count of 8 is 1.

(a) Give a specification of the `pop_count` function. Assume that the given word x is represented as a sequence of $n+1$ bits as follows $b_n b_{n-1} \ldots b_1 b_0$, where each of the $b_i \in \{0, 1\}$.

- **(b)** Give an SML function to compute the population count of a word, where an integer is used to represent a word.
- **(c)** Use the function schema to transform the SML function into an equivalent C function.
- **(d)** Show, step by step, that the C code is the result of direct transformation from SML code.
- **(e)** Write a main function to go with your C population count function. The main function should call `pop_count` with at least three interesting words and print the results.

Exercise 2.22 A *nibble* is a group of four adjacent bits. A checksum of a word can be calculated by adding the nibbles of a word together. For example, the checksum of 17 is 2 and the checksum of 18 is 3.

- **(a)** Give a specification of the `checksum` function. Assume that a given word x is represented as a sequence of $k+1$ nibbles as follows $n_k n_{k-1} \ldots n_1 n_0$, where each of the nibbles n_i has a value in the range $0 \ldots 15$.
- **(b)** Give an SML function to compute the checksum of a word, where an integer is used to represent a word. Test your function.
- **(c)** Use the function schema to transform the SML function into an equivalent C function.
- **(d)** Show, step by step, that the C code is the result of a direct transformation from the SML code.
- **(e)** Write a main function to go with your C checksum function. The main function should call `checksum` with at least three interesting words and print the results.

Exercise 2.23 The n-th Fibonacci number f_n is defined by the following recurrence relation:

$$\begin{array}{rcl} n, f_n & : & I\!N \\ f_0 & = & 0 \\ f_1 & = & 1 \\ f_n & = & f_{n-1} + f_{n-2}, \text{ if } n \geq 2 \end{array} \tag{2.11}$$

Write an SML function `fib` to calculate the n-th Fibonacci number. Then, give the corresponding C function and a main program to test the C version of `fib` for at least three interesting values of n.

Exercise 2.24 The 'nFib' number is a slight variation on the Fibonacci number. It is defined as follows:

$$\begin{array}{rcl} n, f_n & : & I\!N \\ f_0 & = & 1 \\ f_1 & = & 1 \\ f_n & = & 1 + f_{n-1} + f_{n-2}, \text{ if } n \geq 2 \end{array}$$

(a) What is the difference between the Fibonacci series and the nFib series?

(b) Write an SML function `nfib` to calculate the n-th nFib number.

(c) Give the C function that corresponds exactly to the SML version of `nfib`.

(d) Write a main function that calls `nfib` and prints its result. Test the C version of `nfib` for at least three interesting values of n.

(e) If you study the formula for f_n above closely, you will note that the value f_n is the same as the number of function calls made by your SML or your C function to calculate f_n. Measure how long it takes SML to compute f_{30} and calculate the number of function calls per second for SML.

(f) Perform the same measurement with your C program. Which of the two language implementations is faster? By how much? Document every aspect of your findings, so that someone else could repeat and corroborate your findings.

Exercise 2.25 Write an SML function `power_of_power` to compute the n-th term in the following series:

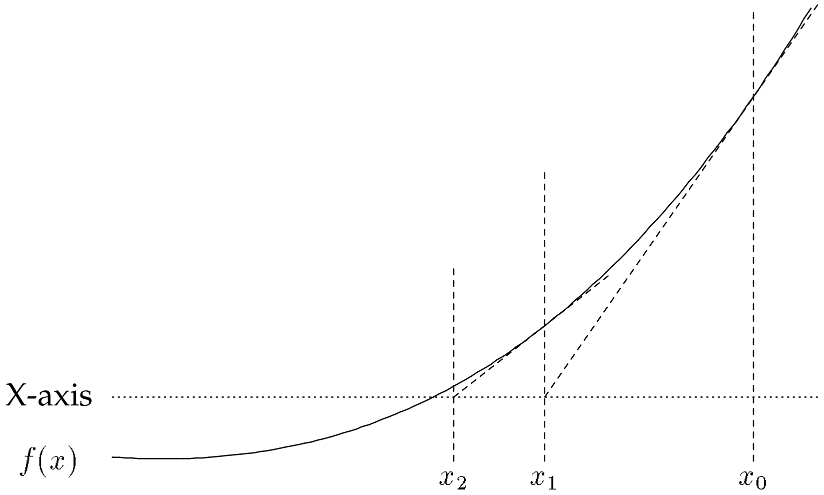
$$\overbrace{1}^{n=0}, \overbrace{m}^{n=1}, \overbrace{m^m}^{n=2}, \overbrace{m^{(m^m)}}^{n=3}, \overbrace{m^{(m^{(m^m)})}}^{n=4} \ldots$$

The n thus counts the number of occurrences of m in the term. Give the C version of `power_of_power` and give a main program to test the C version for at least three interesting values of m and n.

Exercise 2.26 The Newton-Raphson method is an alternative to the bisection method for finding the roots of a function $f(x)$. It works on the basis of the following observation: if there is a value x_i, so that $f(x_i)$ is approximately zero, then a better approximation of the root is x_{i+1} defined as:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2.12}$$

Here, the function $f'(x)$ is the derivative of $f(x)$. The working of the Newton-Raphson method is shown schematically below. Given the function $f(x)$ and a point x_0, draw the tangent of $f(x_0)$. The point where the tangent intersects the X-axis, x_1, is closer to the root of the function; x_1 is therefore a better approximation of the root. The process is repeated using x_1. Drawing the tangent of $f(x_1)$ and intersecting it with the X-axis gives the point x_2, which is closer to the root of the function than x_1. The process can be repeated until the function value becomes (almost) zero.



- **(a)** Given the function $f(x)$, its derivative $f'(x)$, and an initial approximation x_0, show the mathematics for calculating the root.
- **(b)** Give an SML function `newton_raphson` that implements the Newton-Raphson method on the basis of the mathematics from (a) above. Make sure that as much information as possible is captured by arguments.
- **(c)** Use your Newton-Raphson function to calculate the root of the `parabola` function from Section 2.5.
- **(d)** Transform the SML functions from (b) and (c) above into C and form a complete C program to calculate the roots of the `parabola` function.
- **(e)** Trace the evaluation of the following C expression:

  ```
  newton_raphson( parabola, parabola_, 0.001, 1.5 )
  ```

- **(f)** The Newton Raphson method is a fast method to determine the root, but it does not provide a root in all cases. As an example, if one tries to find the root of $f(x) = \frac{1}{x} - 1$, the root will be found if the initial value for x is 0.5, but not if the initial value is 3. What happens if the Newton-Raphson algorithm is used to determine the root of $\frac{1}{x} - 1$, starting with $x = 3$?

Chapter 3

Loops

In the preceding chapter, we have used a purely functional programming style. This style has the advantage that the correspondence between algorithms and C implementations is close. However, it is not always possible to use this style of programming in C for three reasons. Firstly, not all C data types are first class citizens; For example in C, a function cannot return an array as a result, whereas it can return a structure as a result. Secondly, more efficient implementations which require the use of constructs not directly available in a functional language, such as loops, are sometimes possible. A good C compiler would generate the same efficient code for loops and tail recursive functions. Using loops to implement non-tail recursive functions makes it possible to achieve a degree of efficiently beyond what most C compilers are able to offer using just recursion. Thirdly, the efficiency of the allocation and reuse of store can often be improved if we do not adhere strictly to the functional style.

This chapter presents a model of the store that will serve as the basis for writing idiomatic and efficient C functions. The internal workings of these functions may no longer be functional, but the interface to these functions will stay purely functional. We also provide a number of techniques to assist in the development of C functions from their functional counterparts. These techniques are based on the use of program transformation schemas, similar to the function schema of the previous chapter. The schemas of the present chapter are used to transform tail recursive functions into loops. It is generally not possible to use a schema for transforming non-tail recursion into tail recursion, because this requires intelligence that cannot be captured in a schema.

3.1 A model of the store

Computers have a store in which data is entered, maintained, and retrieved. The functional programming style hides the details of the storage management from the programmer. This makes life easy. All the programmer needs to consider is the algorithmic aspects of problem solving. The disadvantage of hiding the manipulation of the store is that the programmer sometimes may wish to exert control over exactly how data is manipulated in order to improve the efficiency of an implementation. In general, this extra control is not available to functional programmers. It is the domain of the imperative programmer. Thus in a sense, imperative programming is a relatively low level activity. This does not imply that an imperative programmer cannot build programs that achieve both a high level of abstraction and an efficient implementation. However, to achieve these two aims, the imperative programmer needs to do quite a lot of work.

The store of a computer may be implemented in many ways. The main memory, the caches, the registers, as well as secondary memory (disks), and even backup storage all implement parts of the store. It would be unreasonable for a programmer to have to think about all these different storage devices. It is easier to think in terms of a *model of the store* rather than the store itself. The model captures the essential aspects of the store and abstracts away inessential details.

A model of the store is a *mapping of locations onto values*. Locations are usually positive natural numbers, but we abstract from that and give names to the cells. The values are held in fixed size cells; most computers today use 32-bit cells. Values can be arbitrary, provided they fit into the available space. Here is a small store with 4 cells, named "A", "B", "C" and "D":

A:	. . .
B:	. . .
C:	. . .
D:	. . .

The store of the computer is accessed mainly through the arguments and the local variables of the functions (we will discuss global variables in Chapter 8). Unless essential, we will not differentiate between arguments and local variables, and we will simply refer to both as the arguments, since the values that are carried by the local variables must be stored somewhere, just like the arguments. It is the task of the compiler to allocate cells for the arguments that occur in programs.

The number of available cells is related to the amount of storage that is physically present in the computer. Therefore, a realistic model of the store will limit the number of available cells. Storage and thus cells are often in short supply, and using store impacts performance. The compiler will need to do its best to allocate arguments to cells in such a way that cells are reused as often as possible.

Consider instructing the computer directly, as is common with pocket calculators. Firstly, store four numbers, arbitrarily chosen as 17, 3, 1, and 1000, in the cells, thus allocating one cell to each number:

A:	0017
B:	0003
C:	0001
D:	1000

To add all the numbers together, start by adding 17 and 3. Then add 1 to the result, and so on. Where would the intermediate sums, 20 and 21, and the final result (1021) be stored? All cells are in use and there is no free cell. The solution is to reuse cells after they have served their useful life. One possibility would be to use cell "A" for this purpose:

A:	0017	0020	0021	1021
B:	0003	0003	0003	0003
C:	0001	0001	0001	0001
D:	1000	1000	1000	1000
	stage 1	stage 2	stage 3	stage 4

The simple model of the store shown above is adequate for our purposes. It can be refined in several ways; the interested reader should consult a book on computer architecture [14, 2]. Here, the simple model of the store will be used to investigate the behaviour of a number of C functions.

3.2 Local variable declarations and assignments

To illustrate how the model of the store might help us to understand C programs let us now write a function that implements the addition of the four cells. This program needs two new concepts, one to access the store and one concept to modify the store:

- Store access is provided by a *local variable declaration*, which associates a name with a cell in the store. The declaration of a local variable has the following general form:

  t x = v ;

  Here t should be a type, x should be an identifier and v should be the initial value of the cell associated with x. The value v should have type t. The local variables can be declared immediately at the beginning of each block of code (that is just after a {). The cell can be used in the *scope* of the variable, that is, until the end of a block of code (until the matching closing curly bracket }). A local variable declaration differs from a local constant declaration in that the variable does not represent a fixed value, but instead a cell where values can be stored and updated.

  It is good programming practice to always initialise local variables, although C does not require it. An uninitialised local variable declaration has the following general form:

  t x ;

Uninitialised variables in C have an arbitrary value. No assumptions should thus be made about the value of an uninitialised variable. Errors caused by the accidental omission of an initialisation are difficult to find, so it is a good idea to check that variable declarations contain an initialisation when something strange happens to your program.

- An assignment statement can be used to change the value of the cell associated with a local variable. Assignment statements have the general form:

  x = e ;

  Here x is an identifier and e should be an expression of the same type as x. An assignment statement may be written anywhere after the declarations of a block.

The function main below uses local variable declarations and assignments to implement a C program that adds the contents of the four cells as described in the previous section. The function main first declares four appropriately initialised local variables, A, B, C and D. It then uses three assignment statements to add the values. The cell associated with the local variable A is used to store the intermediate and final results. The program finally prints the sum as stored in A and returns to the calling environment.

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;
  A = A + D ;
  printf( "%d\n", A ) ;
  return 0 ;
}
```

We will now investigate how we can use the model of the store to study the behaviour of a C program in detail. We do this by making an *execution trace*. An execution trace is a step by step account of the execution of a program, showing the values associated with the variables involved in the execution.

Execution begins with a call to main. This is shown in Step 1 below. The cells associated with A, B, C and D are also allocated at Step 1. The rendering of the store is associated with the position in the program that has been reached by the execution. We draw a box representing the store with an arrow pointing at the position in the program that has been reached.

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;
  A = A + D ;
  printf( "%d\n", A ) ;
  return 0 ;
}
```

←
A : 17
B : 3
C : 1
D : 1000
(Step 1)

At Step 2 the store is updated by the first assignment because the value associated with A changes from 17 to 20:

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;
  A = A + D ;
  printf( "%d\n", A ) ;
  return 0 ;
}
```

←
A : 20
B : 3
C : 1
D : 1000
(Step 2)

At Step 3 the store is updated again to reflect the effect of the second assignment statement:

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;
  A = A + D ;
  printf( "%d\n", A ) ;
  return 0 ;
}
```

←
A : 21
B : 3
C : 1
D : 1000
(Step 3)

The last assignment statement updates the store again to yield the final association of the value 1021 with the variable A:

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;                 A : 1021
  A = A + D ;                 B : 3
  printf( "%d\n", A ) ;  ←    C : 1      (Step 4)
  return 0 ;                  D : 1000
}
```

The `printf` statement at Step 5 prints the value `1021` associated with `A`. The `printf` statement does not alter the contents of the store, so we will not redraw the box that displays the contents:

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;
  A = A + D ;
  printf( "%d\n", A ) ;  ← (Step 5)
  return 0 ;
}
```

The function `main` finally returns to its caller delivering the return value `0`. The entry `return 0` in the store box below indicates that the return value needs to be communicated from the *callee* (the function `main`) to the *caller*. The latter may need this value for further computations. In this particular case the caller is the operating system, which interprets a return value of 0 as successful execution of the program, and any other value as an indication of a failure.

```
int main( void ) {
  int A =   17 ;
  int B =    3 ;
  int C =    1 ;
  int D = 1000 ;
  A = A + B ;
  A = A + C ;
  A = A + D ;
  printf( "%d\n", A ) ;
  return 0 ;                ←| return 0 |(Step 6)
}
```

A cell can only be reused if the cell is not going to be accessed anymore. Therefore the order of events is important. The programmer should be able to control that order so as to guarantee the appropriate sequencing of events. In the case of the example above, it would not be possible to start adding `1000` to `1` while still using cell `A` for intermediate results. This would destroy the value `17` before it is used. The reuse of cell `A` destroys the old value associated with that cell.

One must be careful not to destroy a value that is needed later, as it will introduce an error. One of the major differences between functional programming and imperative programming is that functional programmers cannot make such errors, for they have no direct control over store reuse.

3.3 While loops

Consider the problem of finding the next leap year. The next leap year following any year y after 1582 is given by the specification:

$$\begin{array}{rcl}\text{leap} & : & I\!\!N \rightarrow I\!\!N \\ \text{leap}(y) & = & \min\{l \in I\!\!N \mid l \geq y \wedge (l \bmod 4 = 0) \\ & & \quad \wedge ((l \bmod 100 \neq 0) \vee (l \bmod 400 = 0))\}\end{array} \tag{3.1}$$

The condition for a leap year is a bit too long for this example, so we simplify it only to give the right answer between 1901 and 2099 (the general case is left to the reader, see Exercises 3.2 and 3.4):

$$\begin{array}{rcl}\text{leap} & : & I\!\!N \rightarrow I\!\!N \\ \text{leap}(y) & = & \min\{l \in I\!\!N \mid l \geq y \wedge (l \bmod 4 = 0)\}\end{array} \tag{3.2}$$

Given a particular year y, a leap year cannot be far away. The following algorithm moves onto the next year if the current year is not a leap year.

```
(* leap : int -> int *)
fun leap y = if y mod 4 <> 0
                then leap (y+1)
                else y ;
```

Exercise ⋆ 3.1 Prove that the function `leap` satisfies (3.2).

The technique that has been developed in the previous chapter to translate from an SML function to a C function is directly applicable:

```
int leap( int y ) {
  if( y % 4 != 0 ) {
    return leap( y+1 ) ;
  } else {
    return y ;
  }
}
```

Exercise 3.2 Modify the SML function `leap` to deal with any year after 1582 using (3.1) and transform the SML function into C.

When this function is executed as part of a program, the function `leap` associates a cell with the argument `y`. The cell is initialised and used, but it is not *reused*. To see this, we will look at the execution trace of `leap`. Execution begins with a call to `leap` with some suitable argument value, say 1998. This is Step 1 below. The argument `y` is also allocated at Step 1.

```
int leap( int y ) {           ← | y : 1998 | (Step 1)
  if( y % 4 != 0 ) {
    return leap( y+1 ) ;
  } else {
    return y ;
  }
}
```

At Step 2 the test on `y` is performed, which yields true. This implies that a new call must be made to `leap`.

```
int leap( int y ) {
  if( y % 4 != 0 ) {
    return leap( y+1 ) ;      ← (Step 2)
  } else {
    return y ;
  }
}
```

At Step 3 the function `leap` is entered recursively. The store is extended with the cell necessary to hold the new argument. The old argument is kept for later use as `y` . This signals that, until the new call terminates, the cell referred to as `y` is inaccessible: there is no identifier in the program with this name, and it is not even a legal identifier in C. The store is used as a stack of cells, with only the most recently stacked cell being accessible. There are now 2 cells in use; the new cell stacked on top of the old one:

```
int leap( int y ) {            ┌───────────┐
  if( y % 4 != 0 ) {         ← │ y : 1999  │ (Step 3)
    return leap( y+1 ) ;       │ y  : 1998 │
  } else {                     └───────────┘
    return y ;
  }
}
```

At Step 4 the test on the new value of y is performed, yielding true again. The store remains unaltered, and it has therefore not been redrawn:

```
int leap( int y ) {
  if( y % 4 != 0 ) {
    return leap( y+1 ) ;  ← (Step 4)
  } else {
    return y ;
  }
}
```

At Step 5 the function leap is entered for the third time. This extends the store with a third cell. The previous values of y are now shown as y and y . Only y is accessible.

```
int leap( int y ) {            ┌─────────────┐
  if( y % 4 != 0 ) {           │ y : 2000    │
    return leap( y+1 ) ;     ← │ y  : 1999   │ (Step 5)
  } else {                     │ y    : 1998 │
    return y ;                 └─────────────┘
  }
}
```

At Step 6 the test on the latest value of y is performed, yielding false, because 2000 is divisible by 4. The store remains the same:

```
int leap( int y ) {
  if( y % 4 != 0 ) {      ← (Step 6)
    return leap( y+1 ) ;
  } else {
    return y ;
  }
}
```

At Step 7 the current call to leap reaches the else-branch of the conditional. The store remains unaltered.

```
int leap( int y ) {
  if( y % 4 != 0 ) {
    return leap( y+1 ) ;
  } else {
    return y ;              ← (Step 7)
  }
}
```

At Step 8 the third call to `leap` terminates, returning the value 2000 to the previous call. In returning, the cell allocated to the argument of the third call is freed. There are now two cells in use, of which only `y` is accessible.

```
int leap( int y ) {
  if( y % 4 != 0 ) {          ┌─────────────┐
    return leap( y+1 ) ;  ←   │ return 2000 │
  } else {                    │ y : 1999    │ (Step 8)
    return y ;                │ y   : 1998  │
  }                           └─────────────┘
}
```

At Step 9 the second call also returns, removing its argument.

```
int leap( int y ) {
  if( y % 4 != 0 ) {          ┌─────────────┐
    return leap( y+1 ) ;  ←   │ return 2000 │ (Step 9)
  } else {                    │ y : 1998    │
    return y ;                └─────────────┘
  }
}
```

At Step 10 the initial call to `leap` returns the desired result, 2000. This also frees the last cell `y` in use by `leap`.

```
int leap( int y ) {
  if( y % 4 != 0 ) {
    return leap( y+1 ) ;  ←   │ return 2000 │ (Step 10)
  } else {
    return y ;
  }
}
```

The execution trace of `leap(1998)` shows how arguments are allocated and remembered during calls. The trace also shows the return values of the functions involved in the trace. The trace of `leap` does not show that arguments always *have* to be remembered. In fact, the `leap` function does not require arguments to be remembered when the calls are made. The function `leap` is said to be tail recursive: a function is *tail recursive* if the last expression to be evaluated is a call to the function itself. To contrast tail recursive with non-tail recursive functions, here is an SML example of the latter:

```
(* fac : int -> int *)
fun fac n = if n > 1
               then n * fac (n-1)
               else 1 ;
```

The last expression to be evaluated in the function body is the multiplication and not the call to `fac` itself. Before discussing non-tail recursive functions, a method will be presented that gives an efficient C implementation of a tail recursive functions.

3.3.1 Single argument tail recursion

The execution trace of the function `leap` shows that recursive functions do not necessarily produce efficient implementations. The problem is that the function remembers all previous argument values, even though it never needs them again. To remember a potentially large number of old argument values is a source of inefficiency that can, and should, be avoided.

An efficient implementation of a tail recursive function requires the use of a *loop*. There are several loop constructs in C. For this problem, the while-statement is the most appropriate. The other loop constructs will be described later in this chapter, when we will also discuss how to select a loop construct appropriate to the problem being solved. The while-loop is a statement that has the following syntactical form:

```
while( p ) {
  S
}
```

The purpose of the while-statement is to evaluate the expression p, and if this yields true, to execute the statement(s) S of the body. The evaluation of p and the execution of the body are repeated until the condition yields false. The body should be constructed such that executing it should eventually lead to the condition failing and the while-loop halting. As in the case of the if-statements, the curly brackets are not necessary if the body of the while-loop consists of just one statement. We will always use brackets though, to make the programs more maintainable.

We now show a transformation for turning a tail recursive function such as `leap` into a C function using a while-statement. Consider the following schematic rendering of a tail recursive function with a single argument:

```
(*SML single argument tail recursion schema*)
  (* f : t -> t_r *)
  fun f x
      = if p
           then f g
           else h ;
```

Here the x is an argument of type t. The p, g, and h represent expressions that may contain occurrences of x. The type of the expression p must be `bool`, the type of g must be t, and the type of h must be t_r. No calls to f must appear either directly or indirectly in any of the expressions p, g, and h. (Such cases will be discussed later in this chapter.)

A function that can be expressed in the above form of the single argument tail recursion schema can be implemented efficiently using a C function with a while-statement as shown schematically below:

```
/*C single argument while-schema*/
  t_r f( t x ) {
    while( p ) {
      x = g ;
    }
    return h ;
  }
```

The crucial statement is the *assignment*-statement x=g. This is a command that explicitly alters the value associated with a cell in the store. When executed, the assignment-statement computes the value of g, and updates the cell associated with the argument x. If x occurs in g, the value of x immediately before the assignment is used in the computation of g. After the update, the previous value is lost. The new value is used during the next iteration, both by the condition and by the assignment-statement. This amounts to the reuse of a store cell as suggested by the `leap` example.

Substitution of the schematic values from the table above into the single argument while-schema yields an iterative C version of `leap`:

```
  int leap( int y ) {
    while( y % 4 != 0 ) {
      y = y+1 ;
    }
    return y ;
  }
```

Exercise 3.3 Show the correspondence between the SML and C versions of `leap`.

Exercise 3.4 Modify the C function `leap` to deal with any year after 1582 using (3.1).

The C version of `leap` uses the constructs that the typical C programmer would use. We were able to construct this by using a systematic transformation from a relatively straightforward SML function. This is interesting as some would consider the iterative C version of `leap` as advanced. This is because the argument `y` is reused as the result during the while-loop. To write such code requires the programmer to be sure that the original value of the argument will not be needed later.

An execution trace of this new version of `leap` is now in order firstly to see what exactly the while-statement does and secondly to make sure that only a single cell is needed. At Step 1 the iterative version of `leap` is called with the same sample argument 1998 as before:

```
int leap( int y ) {                  ← | y : 1998 | (Step 1)
  while( y % 4 != 0 ) {
    y = y+1 ;
  }
  return y ;
}
```

At Step 2 the condition `y % 4 != 0` is evaluated, which yields true.

```
int leap( int y ) {
  while( y % 4 != 0 ) {    ← (Step 2)
    y = y+1 ;
  }
  return y ;
}
```

As a result, the assignment-statement `y=y+1;` is executed. This *changes* the value associated with the cell for `y` to 1999. The previous value 1998 is now lost. No call is made at this point, so no new cells are needed. The current `y` remains accessible.

```
int leap( int y ) {
  while( y % 4 != 0 ) {
    y = y+1 ;                        ← | y : 1999 | (Step 3)
  }
  return y ;
}
```

At Step 4 the condition is evaluated for the second time. The value associated with `y` has changed, but its value, 1999, is still not divisible by 4. The condition thus returns true again.

```
int leap( int y ) {
  while( y % 4 != 0 ) {    ← (Step 4)
    y = y+1 ;
  }
  return y ;
}
```

At Step 5 the statement `y=y+1;` is executed for the second time. This changes the value associated with `y` to 2000. Again, no new cell is required and `y` remains accessible.

```
int leap( int y ) {
  while( y % 4 != 0 ) {
    y = y+1 ;               ← | y : 2000 | (Step 5)
  }
  return y ;
}
```

At Step 6 the condition is evaluated for the third time, yielding false because 2000 is divisible by 4. This terminates the loop and gives control to the statement following the while-statement, which is the return statement. Thus the body of the while-statement `y=y+1;` is not executed again.

```
int leap( int y ) {
  while( y % 4 != 0 ) {
    y = y+1 ;
  }                         ← (Step 6)
  return y ;
}
```

At Step 7 the function `leap` terminates: the statement following the while-statement is executed, returning the value 2000.

```
int leap( int y ) {
  while( y % 4 != 0 ) {
    y = y+1 ;
  }
  return y ;                ← | return 2000 | (Step 7)
}
```

The execution trace shows that only one cell is ever used. It also shows that the conditional of the while-statement is evaluated three times. The body of the while-statement `y=y+1;` is executed only twice. This is a general observation: if, for some $n \geq 0$, the conditional of a while is executed $n + 1$ times, the body of the while-statement is executed n times.

We have developed an efficient and idiomatic C implementation of the `leap` function on the basis of a systematic transformation, the while-schema. The while-schema is of limited use because it cannot deal with functions that have more than one argument. In the next section, the while-schema will be appropriately generalised.

3.3.2 Multiple argument tail recursion

The function `leap` of the previous section has a single argument. This is too restrictive. For a recursive function with more than one argument, the while-schema can be generalised. The standard method in functional programming for generalizing a function with a single argument to one with many arguments is to treat the collection of all arguments as a tuple. This tupling of the arguments brings a function with more than one argument in the form below. We will call this tupling of arguments the *pre-processing* of the multiple argument while-schema.

```
(*SML multiple argument tail recursion schema*)
  (* $f$ : ($t_1$* ... $t_n$) -> $t_r$ *)
  fun $f$ ($x_1$, ... $x_n$)
      = if $p$
             then $f$ $g$
             else $h$ ;
```

Here, $x_1 \dots x_n$ are the arguments of types $t_1 \dots t_n$ respectively. The expressions p, g, and h are now dependent on $x_1 \dots x_n$. No calls to f, either directly or indirectly, must appear in any of the expressions p, g, and h.

It would have been nice if the corresponding generalisation of the while-schema could look like this.

```
/*C multiple argument while-schema*/
  $t_r$ $f$( $t_1$ $x_1$, ... $t_n$ $x_n$ ) {
    while( $p$ ) {
      ($x_1$, ... $x_n$) = $g$ ;
    }
    return $h$ ;
  }
```

Unfortunately the multiple assignment-statement is not supported by C:

```
  ($x_1$, ... $x_n$) = $g$ ;  /* incorrect C */
```

In one of the predecessors to the C language, BCPL, the multiple assignment was syntactically correct. It is an unfortunate fact of life that useful concepts from an earlier language are sometimes lost in a later language. Multiple assignment is in C achieved by a number of separate single assignment-statements. Therefore to produce a correct C function, we must do some *post-processing*. The transformation of a multiple assignment into a sequence of assignments is shown in the two examples below. The first example is an algorithm for computing the modulus of two numbers. The second example uses Euclid's algorithm to compute the greatest common divisor of two numbers.

Independent assignments

The function `modulus` as shown below uses repeated subtraction for computing the modulus of `m` with respect to `n`, where `m` and `n` both represent positive natural numbers.

```
  (* modulus : int -> int -> int *)
  fun modulus m n = if m >= n
                       then modulus (m - n) n
                       else m : int ;
```

This function has been written using a curried style, which is preferred by some because curried functions can be partially applied. C does not offer curried functions so we will need to transform the curried version of `modulus` into an uncurried version first. This requires the arguments to be grouped into a tuple as follows:

```
(* modulus : int*int -> int *)
fun modulus (m,n) = if m >= n
                       then modulus (m - n,n)
                       else m : int ;
```

If you prefer to write functions directly with tupled arguments, you can save yourself some work because the preparation above is then unnecessary.The correspondence between the elements of the multiple argument while-schema and the version of `modulus` with the tupled arguments is as follows:

schema:	Functional	C
f:	`modulus`	`modulus`
n:	`2`	`2`
(t_1*t_2):	`(int*int)`	`(int,int)`
t_r:	`int`	`int`
(x_1,x_2):	`(m,n)`	`(m,n)`
p:	`m >= n`	`m >= n`
g:	`(m - n,n)`	`(m - n,n)`
h:	`m : int`	`m`

Filling in these correspondences in the multiple argument while-schema yields:

```
int modulus( int m, int n ) {
  while( m >= n ) {
/*!*/   (m,n) = (m - n,n) ;
  }
  return m ;
}
```

The `/*!*/` marks a syntactically incorrect line. To turn this into correct C, the multiple assignment must be separated into a sequence of assignments. In this case (but not in general – see the section on Mutually dependent arguments below), we can simply separate the two components of the tuples on the left and right hand sides of the multiple assignment and assign them one by one as follows:

```
(m,n) = (m - n,n) ;                m = m - n ;
                                   n = n ;
```

The assignment-statement `n=n;` does nothing useful so we can safely remove it. This yields the following C implementation of `modulus`:

```
int modulus( int m, int n ) {
  while( m >= n ) {
    m = m - n ;
  }
  return m ;
}
```

With the aid of the multiple argument while-schema, we have implemented an efficient and idiomatic C function. The multiple argument while-schema requires pre-processing of the SML function to group all arguments into a tuple, and it requires post-processing of the C function to separate the elements of the tuple into single assignment-statements. The pre-processing (tupling) is, to some extent, the inverse of the post-processing (untupling). In this example, the post-processing was particularly easy. This is not always the case, as the next example will show.

Mutually dependent arguments

Consider the function `euclid` from Chapter 2. To prepare for the multiple argument while-schema, the arguments of the function have already been grouped into a tuple:

```
(* euclid : int*int -> int *)
fun euclid (m,n) = if n > 0
                      then euclid (n,m mod n)
                      else m ;
```

Exercise 3.5 Show the table of correspondence between the `euclid` function and the while-schema.

The multiple argument while-schema yields:

```
int euclid( int m, int n ) {
  while( n > 0 ) {
/*!*/   (m,n) = (n,m % n) ;
  }
  return m ;
}
```

The line marked `/*!*/` must again be transformed into a sequence of assignments. It would be tempting to simply separate the two components of the tuples on the left and right hand sides of the multiple assignment and assign them one by one as follows:

```
m = n ;
n = m % n ;   /* WRONG */
```

This is wrong, because first changing the value associated with `m` to that of `n` would cause `n` to become 0 always. To see this, consider the following execution trace. We are assuming that `m` and `n` are associated with some arbitrary values x and y:

← m: x n: y (Step 1)

```
m = n ;
n = m % n ;   /* WRONG */
```

```
m = n ;
```

← m: y n: y (Step 2)

```
n = m % n ;   /* WRONG */
```

```
m = n ;
n = m % n ;    /* WRONG */  ←| m: y    n: 0 | (Step 3)
```

First assigning n and then m would not improve the situation. The problem is that the two assignment statements are *mutually dependent*. The only correct solution is to introduce a temporary variable, say old_n, as shown below. The temporary variable serves to remember the current value in one of the cells involved, whilst the value in that cell is being changed.

```
int euclid( int m, int n ) {
  while( n > 0 ) {
    const int old_n = n ;
    n = m % old_n ;
    m = old_n ;
  }
  return m ;
}
```

The declaration of old_n above is a local constant declaration rather than a local variable declaration. This is sensible because old_n is not changed in the block in which it is declared. We are making use of the fact that the local variables and constants of a block exist only whilst the statements of the block are being executed. So each time round the while-loop a new version of old_n is created, initialised, used twice and then discarded. The cell is never updated, so it is a constant.

The multi-argument while-schema gives us an efficient and correct C implementation of Euclid's greatest common divisor algorithm. As we shall see, since most functions have no mutual dependencies in the multiple assignment-statement, the post-processing is thus often quite straightforward.

Exercise 3.6 Trace the execution of euclid(9,6) using the iterative version of euclid above.

3.3.3 Non-tail recursion: factorial

Not all functions are tail recursive. To investigate the ramifications of this fact on our ability to code efficient and idiomatic C functions, let us consider an example: the factorial function.

$$\begin{array}{rcl} \cdot ! & : & \mathbb{N} \rightarrow \mathbb{N} \\ n! & = & \prod_{i=1}^{n} i \end{array} \tag{3.3}$$

We have seen the recursive SML function to compute the factorial before:

```
(* fac : int -> int *)
fun fac n = if n > 1
               then n * fac (n-1)
               else 1 ;
```

Exercise ⋆ 3.7 Prove by induction over `n` that `n`! = `fac n`.

The corresponding C version of the factorial function is:

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ;
  } else {
    return 1 ;
  }
}
```

Let us now trace the execution of `fac(3)` to study the efficiency of the C code. At Step 1 the function `fac` is entered.

```
int fac( int n ) {              ← n : 3  (Step 1)
  if( n > 1 ) {
    return n * fac (n-1) ;
  } else {
    return 1 ;
  }
}
```

The test `n > 1` yields true because `n` is associated with the value `3`, so that at Step 2 the then-branch of the conditional is chosen.

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ;      ← (Step 2)
  } else {
    return 1 ;
  }
}
```

This causes a call to be made to `fac(2)` at Step 3. The number `3` will have to be remembered for later use when `fac(2)` has delivered a value that can be used for the multiplication.

```
int fac( int n ) {              ← n : 2   (Step 3)
  if( n > 1 ) {                   n  : 3
    return n * fac (n-1) ;
  } else {
    return 1 ;
  }
}
```

At Step 4 the new value of `n` is tested, yielding true again.

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ;  ← (Step 4)
  } else {
    return 1 ;
  }
}
```

This causes another call to be made, this time to `fac(1)` at Step 5. The number `2` will have to be remembered for use later, just like the number `3`.

```
                             ┌────────┐
                             │ n : 1  │
int fac( int n ) {          ←│ n  : 2 │ (Step 5)
  if( n > 1 ) {              │ n   : 3│
    return n * fac (n-1) ;   └────────┘
  } else {
    return 1 ;
  }
}
```

At Step 6 the new value of `n` is tested, yielding false.

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ;  ← (Step 6)
  } else {
    return 1 ;
  }
}
```

At Step 7 the most recent call returns `1`.

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ;
  } else {                   ┌──────────┐
    return 1 ;               │ return 1 │
  }                         ←│ n : 2    │ (Step 7)
}                            │ n  : 3   │
                             └──────────┘
```

At Step 8 the second call returns `2`, which is calculated from the return value of Step 7 and the saved value of `n`.

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ; ← return 2   (Step 8)
                             n : 3
  } else {
    return 1 ;
  }
}
```

Finally, at Step 9 the first call returns `6`.

```
int fac( int n ) {
  if( n > 1 ) {
    return n * fac (n-1) ; ← return 6   (Step 9)
  } else {
    return 1 ;
  }
}
```

Summarising, because the function `fac` is not tail recursive, we have the following efficiency problem:

- Firstly, the `fac` function generates the values $n, n-1, \ldots 3, 2, 1$ *on entry* to the recursion.
- All values $n, n-1, \ldots 3, 2, 1$ have to be remembered, which requires n cells in the store.
- Finally all values are multiplied *on exit* from the recursion.

This can be symbolised as:

$$\begin{array}{rcl} \cdot! & : & \mathbb{N} \rightarrow \mathbb{N} \\ n! & = & n \times ((n-1) \times \ldots (3 \times (2 \times 1)) \ldots) \end{array}$$

Here, the parentheses signify that, before n can be multiplied to some number, the latter must be computed first. The efficiency of the implementation can be improved if previous numbers do not have to be remembered. This is possible as multiplication is an associative operator. Therefore we have:

$$\begin{array}{rcl} \cdot! & : & \mathbb{N} \rightarrow \mathbb{N} \\ n! & = & ((\ldots (n \times (n-1)) \times \ldots 3) \times 2) \times 1 \end{array}$$

Now there is no need to remember the numbers $n, n-1 \ldots$. The number n can be multiplied immediately by $n-1$. This in turn can be multiplied directly by $n-2$, and so on. The successive argument values for the ! operator can be generated and used immediately. The only value that needs to be remembered is the *accumulated* result. This corresponds to multiplying on entry to the recursion.

Using an accumulating argument is a standard functional programming technique. Here is a function fac_accu that uses the accumulating argument technique.

```
(* fac_accu : int -> int -> int *)
fun fac_accu n b = if n > 1
                      then fac_accu (n-1) (n*b)
                      else b ;
```

Exercise ⋆ 3.8 Prove by induction over n that for all natural numbers b the following equality holds: b * n! = fac_accu n b. Then conclude that n! = fac_accu n 1.

Exercise 3.9 Show the correspondence between the SML program and the while-schema.

The function fac_accu is is tail recursive. It can therefore be implemented in C using a while-loop:

```
int fac_accu( int n, int b ) {
  while( n > 1 ) {
/*!*/   (n,b) = (n-1,n*b) ;
  }
  return b ;
}
```

The multiple assignment can be replaced by two separate assignment-statements as there is no *mutual* dependency between the two resulting assignments. They must be properly ordered, so that n is used before it is changed:

```
int fac_accu( int n, int b ) {
  while( n > 1 ) {
    b = n*b ;
    n = n-1 ;
  }
  return b ;
}
```

This shows that tail recursion, and therefore loops, can be introduced by using an accumulating argument. It is not always easy to do this and it requires creativity and ad hoc reasoning. In the case of the factorial problem, we had to make use of the fact that multiplication is an associative operation. In other cases shown later, similar reasoning is necessary.

The resulting version fac_accu of the factorial is slightly odd. It has two arguments: the first argument is the number from which the factorial is to be computed, the second number should always be one. A more useful version of factorial would have only one argument:

```
int fac( int n ) {
  return fac_accu( n, 1 ) ;
}
```

This is neither the most efficient nor the most readable version of `fac`, the function `fac_accu` is a specialised function and is probably never called from any other function. In that case we can *inline* functions: we can amalgamate the code of two functions to form one function that performs the whole task. This is more efficient and it improves the readability of the code. Inlining is a seemingly trivial operation: we replace a function call in the *caller* with the body of the called function, that is, the *callee*. However, there are some potential pitfalls:

1. The C language does not provide good support for literal substitution. Special care has to be taken with a `return` statement in the callee. After inlining in the caller, the return statement will now exit the caller instead of the callee.
2. Names of variables might clash. Variables have to be renamed in order to preserve the meaning of the program.
3. Function arguments are passed *by value*. This means that the callee can change its argument without the caller noticing. After inlining, the changes become visible to the caller. This might require the introduction of *local variables*

In the case of `factorial`, the inlining operation results in the following code:

```
int fac( int n ) {
  int b = 1 ;
  while( n > 1 ) {
    b = n*b ;
    n = n-1 ;
  }
  return b ;
}
```

Because the argument `b` of `fac_accu` was changed, we needed to introduce a local variable. The local variable is initialised to `1`.

Other places where local variables are introduced are as a replacement for the `let` clauses in SML. Any expression that has been given a name with a `let` is in general programmed with a local variable in C. This will be shown in the next section where local variables are used extensively. Over the course of this chapter, more functions will be inlined to show how to compose larger C functions.

3.3.4 More on assignments

Many of the while loops will have assignments with the following pattttern:

x = x $\oplus$ y ;

Here, $\oplus$ stands for one of the possible C operators. Examples include both assignments in the while loop of `fac` above:

```
b = n*b ;
n = n-1 ;
```

Because these patterns are so common, C has special assignment operators to deal with these patterns. Any assignment of the pattern above can be written using one of the assignment operators `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, and `^=`. The assignments below have equivalent semantics. Here x is a variable and y an expression of the appropriate type.

$$x \texttt{ = } x \oplus y \texttt{ ;} \qquad\qquad\qquad x \oplus\texttt{=} y \texttt{ ;}$$

This is the case for any of the operators `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `|`, and `^`. Thus `i=i+2` and `i+=2` (read: add `2` to `i`) are equivalent, and so are `j=j/2` and `j/=2` (read: divide `j` by 2).

Using these assignment operators has two advantages. Firstly, they clarify the meaning of the code, as `j/=2` means divide `j` by two as opposed to find the current value of `j`, divide it by two, and store it in `j` (which is *by coincidence* the same variable). The second advantage is a software engineering advantage; when more complex variables are introduced, these operators take care that the variable is specified only once. An example of this is shown in Chapter 5.

Each assignment operation is actually an *expressions* in itself. For example, `j/=2` divides `j` by two and results in this value. Therefore the expression `i += j /= 2` means divide `j` by two and add the result of that operation to `i`. Assignments placed in the middle of expressions can cause errors as the order of assignments is undefined. It is recommended to use the assignment only as a statement.

As a final abbreviation, there are special operators for the patterns v `+= 1` and v `-= 1`, denoted v`++` and v`--`, also known as the increment and decrement operators. They exist in two different forms: the pre-increment `++`i and the post-increment i`++`. The first version is identical to i`+=1` or i`=`i`+1`: add one to i and use the resulting value as the value of the expression `++`i. The post-increment means add one to i, *but use the old, non-incremented value* as the value of the expression. Similarly, `--`n has a value one lower than n`--`.

A common use of the `--` operator is inside a while condition:

```
int absurd_fac( int n ) {
  int b = n ;
  while( --n ) {
    b = n*b ;
  }
  return b ;
}
```

The statement `while( --n )` violates all rules of decency. There is a side effect in the expression (the `--` operator), and the resulting value is interpreted as a boolean (which means an implicit test against zero). This works because any integer which is not zero indicates `true`, while only zero indicates `false`. This particular version of `fac` is not robust, the loop will iterate indefinitely if `n` happens to have the value `0`.

3.3.5 Breaking out of while-loops

We are now ready to tackle the most general case of turning a tail recursive function into an efficient loop. Consider the function `bisection` from Chapter 2. This poses the following problems to the while-schema: the recursion has two termination points (see the lines marked 1 and 2) and it can continue in two different ways (lines marked 3 and 4).

```
(* bisection : (real->real) -> real -> real -> real *)
fun bisection f l h
    = let
         val m = (l + h) / 2.0
         val f_m = f m
      in
         if absolute f_m < eps
(* 1 *)     then m
            else if absolute (h-l) < delta
(* 2 *)             then m
                    else if f_m < 0.0
(* 3 *)                  then bisection f m h
(* 4 *)                  else bisection f l m
      end ;
```

The third and last version of the while-schema that we are about to see will be fully general. It takes into account:

- Multiple termination points based on arbitrary predicates.
- Multiple continuation points based on arbitrary predicates.
- An arbitrary ordering of termination- and continuation-points.
- Local definitions.

```
(*SML general tail recursion schema*)
 (* $f$ : ($t_1$* ... $t_n$) -> $t_r$ *)
 fun $f$ ($x_1$, ... $x_n$)
     = let
          val $y_1$ = $z_1$ (* $y_1$ : $t_{y_1}$ *)
          ...
          val $y_j$ = $z_j$ (* $y_j$ : $t_{y_j}$ *)
       in
          if $p_1$
             then $q_1$
             else
                  ...
                  if $p_k$
                     then $q_k$
                     else $q_{k+1}$
       end ;
```

The $x_1 \ldots x_n$ are the arguments of f, their types are $t_1 \ldots t_n$ respectively. The local variables of f are $y_1 \ldots y_j$; their values are the expressions $z_1 \ldots z_j$, and their types are t_{y_1} ... t_{y_j} respectively. The expressions $p_1 \ldots p_k$ are predicates over the arguments and the local variables. The $q_1 \ldots q_{k+1}$ are expressions that may take one of two forms. This form decides whether the present branch in the conditional is a termination or a continuation point:

continuation The form of q_i is $f\ g_i$, which means that the recursion on f may be continued from here with new values for the arguments as computed by the expression g_i. In this case, g_i must be of type (t_1* ... t_n).

termination The form of q_i is h_i, which means that the recursion on f may terminate here. The function result is delivered by the expression h_i. In this case, h_i must be of type t_r.

The g_i and h_i are all expressions ranging over the arguments and local variables. The only calls permitted to f are those explicitly listed in the continuation case above.

The general while-schema that corresponds to the general tail recursion schema is:

```
/*C general while-schema*/
 t_r f( t_1 x_1, ... t_n x_n ) {
   t_y1 y_1 ;
   ...
   t_yj y_j ;
   while( true ) {
     y_1 = z_1 ;
     ...
     y_j = z_j ;
     if( p_1 ) q_1 ;
     else if( p_2 ) q_2 ;
     ...
     else if( p_k ) q_k ;
     else q_k+1 ;
   }
 }
```

The loop is now in principle an endless loop, which can be terminated only by one of the conditional statements `if(` p_i `)` q_i`;`. Depending on the form of the q_i in the general tail recursion schema, the corresponding q_i in the general while-schema will have one of two forms:

continuation For a continuation point, the form of q_i will be a multiple assignment `{ (`x_1`,` ... x_n`)=`g_i `;}`. The loop will be reexecuted with new values for $x_1 \ldots x_n$.

termination For a termination point, the form of q_i is `{ return` h_i `;}`. This causes the loop (and the function f) to terminate.

It is important not to mix up termination and continuation points. They should be carefully identified in the general tail recursion schema, and the correspondence with the general while-schema should be maintained.

To apply the general while-schema to the `bisection` function, the arguments must be grouped in a tuple as usual. Furthermore, the types of the local variables `m` and `f_m` must be indicated. This yields:

```
(* bisection : (real->real)*real*real -> real *)
fun bisection(f,l,h)
    = let
          val m = (l + h) / 2.0  (* m   : real *)
          val f_m = f m          (* f_m : real *)
      in
          if absolute f_m < eps
(* 1 *)      then m
             else if absolute (h-l) < delta
(* 2 *)              then m
                     else if f_m < 0.0
(* 3 *)                      then bisection(f,m,h)
(* 4 *)                      else bisection(f,l,m)
      end ;
```

Cases 1 and 2 are the termination cases, and 3 and 4 are the continuation cases. The correspondence between the schema and the elements of the functions is:

schema:	Functional	C
f:	`bisection`	`bisection`
n:	`3`	`3`
$(t_1$`*` t_2`*`$t_3)$:	`(real->real*` `real*real)`	`(double(*)(double),` `double,double)`
t_r	`real`	`double`
(x_1, x_2, x_3):	`f,l,h`	`f,l,h`
p_1:	`absolute f_m   < eps`	`absolute(f_m) < eps`
p_2:	`absolute (h-l) < delta`	`absolute(h-l) < delta`
p_3:	`f_m < 0.0`	`f_m < 0.0`
h_1:	`m`	`{ return m ;}`
h_2:	`m`	`{ return m ;}`
g_3:	`(f,m,h)`	`{ (f,l,h)=(f,m,h) ;}`
g_4:	`(f,l,m)`	`{ (f,l,h)=(f,l,m) ;}`
t_{y_1}:	`real`	`double`
t_{y_2}:	`real`	`double`
y_1:	`m`	`m`
y_2:	`f_m`	`f_m`
z_1:	`(l + h) / 2`	`(l + h) / 2`
z_2:	`f m`	`f(m)`

In the multiple assignment statement `(f,l,h)=(f,m,h);` only the `l` is changed and in the statement `(f,l,h)=(f,l,m);` it is only `h` that is changed. Thus we may write `l=m;` and `h=m;` respectively. This gives the following C version:

```
double bisection( double(*f)( double ),
                  double l, double h ) {
  double m ;
  double f_m ;
  while( true ) {
    m = (l+h)/2.0 ;
    f_m = f(m) ;
    if( absolute( f_m ) < eps ) {
      return m ;
    } else if( absolute( h-l ) < delta ) {
      return m ;
    } else if( f_m < 0.0 ) {
      l = m ;
    } else {
      h = m ;
    }
  }
}
```

What we have done so far is to systematically transform an SML function with multiple exits, multiple continuation points, local declarations, and multiple arguments into a C function. Let us now reflect on the C function that we have derived. Worth noting is the use of the condition `true` in the while-statement. This creates in principle an 'endless' loop. It signals that there is no simple condition for deciding when the loop should be terminated.

Exercise 3.10 Complete the skeleton below by giving the contents of the *body of the while-loop*. Comment on the efficiency and the elegance of the implementation.

```
double bisection( double(*f)( double ),
                  double l, double h ) {
  double m = (l+h)/2.0 ;
  double f_m = f(m) ;
  while( ! ( absolute( f_m ) < eps ||
             absolute( h-l ) < delta) ) {
    /*C body of the while-loop*/
  }
  return m ;
}
```

Some books on programming in C regard loops with multiple exits an advanced feature. We prefer the multiple exit over a loop with one single complex termination condition. The SML program has guided us immediately to an efficient and readable C version.

Two methods have been shown for breaking out of a while-loop: the condition of the while-loop can become false, or a `return` statement can be executed (which terminates the function and therefore the while-loop). C offers a third method, the `break`-statement. Whenever a `break` is executed, a while-loop is terminated, and execution resumes with the statement immediately following the while-loop.

We do not need a `break` statement in any of these examples because the functions contain only a single while-loop. However, when functions are inlined to make larger entities, it might be necessary to break out of a loop in a callee to avoid terminating the caller. Inserting a `break` statement will do the trick in that case. A break will only jump out of the closest while loop, breaking about multiple while loops is not possible, normally a `return` is used in that case.

Exercise 3.11 Inline the function `bisection` in the following main program:

```
int main( void ) {
  double x = bisection( parabola, 0.0, 2.0 ) ;
  printf("The answer is %f\n", x ) ;
  return 0 ;
}
```

What is the problem? What is the solution?

We have now completed the treatment of the while-statement as a means to implement recursive functions as efficient and idiomatic C functions. The different forms of the while-schema provide useful techniques for constructing these C functions. C provides another useful loop construct, the for-loop, which is the subject of the next section.

3.4 For loops

C offers various different kinds of loop structure. The while-statement is the most general loop structure. A more specific but useful loop-structure is the for-statement. To see why the for-statement is useful, consider again the specification of the factorial function !:

$$\begin{aligned} \cdot ! &: I\!N \to I\!N \\ n! &= \prod_{i=1}^{n} i \end{aligned}$$

The computation of $n!$ starts by generating the value 1 and steps through the values 2, 3, and so on, until it reaches n. The $\prod$ operator has built-in knowledge about the following:

- evaluate and accumulate the values of the expression (here just i), with the index i bound to a sequence of values.
- the sequence of values is generated, starting with $i = 1$, incrementing the value of i by 1 and ending at the value $i = n$.

The $\prod$ operator has a direct functional equivalent with a list, where the list is purely an arithmetic sequence of the numbers 1 through n. Let us digress briefly on the use of lists and arithmetic sequences.

In many functional languages, arithmetic sequences are part of the language. It is possible to define an SML operator `--` that has the functionality of an arithmetic sequence. We wish to be able to generate both increasing and decreasing sequences:

```
(* -- : int*int -> int list *)
infixr 5 -- ;
fun (m -- n)
    = let
         fun up   i = if i > n
                      then []
                      else i :: up (i+1)
         fun down i = if i < n
                      then []
                      else i :: down (i-1)
      in
         if m <= n
            then up   m
            else down m
      end ;
```

With this definition we can create the following lists:

```
(1--3) = [1,2,3] ;
(0--0) = [0] ;
(1--0) = [1,0] ;
```

The `--` operator is quite a useful operator, but unfortunately, it is not part of the standard libraries of SML. It is used by other authors such as Wikström [16] in his text book on SML, and we shall make extensive use of it here.

Using `--`, the arithmetic sequence operator `fac` can be written as follows:

```
(* fac : int -> int *)
fun fac n = prod (1--n) ;
```

The difference between the mathematical and the functional version of `fac` is that the function `prod` is not concerned with the generation of the index values; the processes of generating the indices and accumulating the product are completely separated in the functional program. This is the principle of the separation of concerns: issues that can be separated should be separated. The separation makes it straightforward, for example, to replace `prod` by `sum` to compute the sum of the numbers rather than their product. The same mechanism of generating the numbers would be used in either case.

To make use of the separation of generating indices and accumulating the product, a further investigation of the nature of the `prod` operator is useful. The list operation `prod` is usually defined in terms of the higher order function `foldl`:

```
(* foldl : ( a-> b-> a) ->  a ->  b list ->  a *)
fun foldl f r []      = r
  | foldl f r (x::xs) = foldl f (f r x) xs ;
```

The function `foldl` is similar to the standard SML function `revfold`, but it is not the same. The arguments are in a different order, and our first argument `f` is a curried function. We prefer to use the name `foldl` because it is the standard name for this function in virtually all other functional programming languages. Furthermore, adopting the name `foldl` makes it possible to give a consistent name to its dual function `foldr`, which is equivalent to `fold` in SML. We will be needing `foldr` in the next section, where we will also give its definition.

Returning to our example and the `prod` operator, we can now give the definition of the latter using `foldl`:

```
(* prod : int list -> int *)
fun prod xs = foldl mul 1 xs ;
```

The auxiliary function `mul` is just the curried version of the multiplication operator `*`:

```
(* mul : int -> int -> int *)
fun mul x y = x * y : int ;
```

Let us now investigate how `prod` works. When given a list, `foldl` returns an expression with the elements of the list folded into a new value. The folding is done using an *accumulation* function, `mul` in this case. Therefore, the factorial function causes the following transformations to be made (using a three element list for convenience):

```
foldl mul 1    (1  ::  (2  ::  (3  ::  [])))
                ↓   ↓   ↓   ↓   ↓
= (((   1   *  1)   *   2)  *   3)
= ((    1           *   2)  *   3)
= (     2                   *   3)
=       6
```

The folding is shown schematically as the multiplications `1*1=1`, `1*2=2`, and `2*3=6`. The diagram shows that, once the numbers have been generated, they can be accumulated into a product. The numbers are never needed again. The pattern of computation above is so common that it is worthwhile to introduce a new schema. This foldl-schema operates on a definition of the following form:

```
(*SML foldl schema*)
  (* $f$ : $t_1$ -> ... -> $t_j$ -> $t_r$ *)
  fun $f$ $x_1$ ... $x_j$
      = foldl $g$ $b$ ($m$ -- $n$) ;
```

Here the $x_1 \ldots x_j$ are variables of types $t_1 \ldots t_j$. The g is an expression that represents a function of type t_r `-> int ->` t_r, the b is of type t_r, and m and n are expressions of type `int`. The variables $x_1 \ldots x_j$ may occur in the expressions g, b, m, and n. The function name f should not occur on the right hand side of the equals sign.

If $m < n$, the C implementation of the foldl-schema contains an increasing for-statement, that is, one that counts upwards:

```
/*C increasing for schema*/
  t_r f( t_1 x_1, ... t_j x_j ) {
    int i ;
    t_r a = b ;
    for( i = m; i <= n; i++ ) {
      a = g( a, i ) ;
    }
    return a ;
  }
```

The for-statement has the following elements:

- The index i which acts as the loop counter. It must be declared explicitly.
- An initialisation expression, i=m; in this example, which gives the index its initial value.
- A boolean expression to test for termination, i <= n in this example. The loop body (see below) is entered for as long as the condition is true.
- An iteration expression, i++ in this example, which is executed after the loop body to increment the index. The operator ++ means increment the variable.
- A loop body statement a=g(a, i);, which, using the index i, does the actual work. This consists of accumulating the results of the loop in the *accumulator* a.

The for-statement in the for-schema is preceded by two declarations. The first declares the index i. It is not necessary to give the index an initial value as the for-statement will do that. The second declaration t_r a=b; declares the accumulator for the loop. The accumulator must be initialised properly. The accumulator is updated in the loop, using the values generated by the index and the current value of the accumulator.

When instantiating the increasing for-schema, it is best to choose names for a and i that do not already occur in the function that is being created. In particular one should avoid the names of the arguments of the function.

If $m > n$, then the C implementation of the for-schema contains a decreasing for-statement, that is, one that counts downwards:

```
/*C decreasing for schema*/
  t_r f( t_1 x_1, ... t_j x_j ) {
    int i ;
    t_r a = b ;
    for( i = m; i >= n; i-- ) {
      a = g( a, i ) ;
    }
    return a ;
  }
```

The differences between the increasing and the decreasing for loops are in their test for termination and the iteration expression:

- In the decreasing case, the test for termination tests on greater or equal, i `>=` n.
- The iteration expression is now a decrement: i`--`.

3.4.1 Factorial using a for-loop

The increasing for-schema can be applied after substitution of the definition of `prod` in that of `fac`. This yields the following explicitly folding version of `fac`:

```
(* fac : int -> int *)
fun fac n = foldl mul 1 (1--n) ;
```

The correspondence between the foldl-schema, the increasing for-schema, and this explicitly folding version of `fac` is:

schema:	Functional	C
f:	`fac`	`fac`
j:	`1`	`1`
x_1:	`n`	`n`
t_1:	`int`	`int`
t_r:	`int`	`int`
g:	`mul`	`*`
b:	`1`	`1`
m:	`1`	`1`
n:	`n`	`n`

This yields the following C implementation below, where `mul( accu, i )` has been simplified to `accu * i`:

```
int fac( int n ) {
  int i ;
  int accu = 1 ;
  for( i = 1; i <= n; i++ ) {
    accu = accu * i ;
  }
  return accu ;
}
```

To investigate the behaviour of the for-statement, the execution of `fac(3)` will be traced. At Step 1 the argument `n` is allocated, and the variable is associated with the value 3.

```
int fac( int n ) {                  ← n : 3  (Step 1)
  int i ;
  int accu = 1 ;
  for( i = 1; i <= n; i++ ) {
    accu = accu * i ;
  }
  return accu ;
}
```

At Steps 2 and 3 the local variables `i` and `accu` are allocated. The value associated with `i` variable is undefined. This is indicated by the symbol ⊥. The value associated with `accu` is 1.

```
int fac( int n ) {
  int i ;                              n : 3
  int accu = 1 ;                    ←  i : ⊥      (Step 3)
  for( i = 1; i <= n; i++ ) {          accu : 1
    accu = accu * i ;
  }
  return accu ;
}
```

At Step 4 two actions take place. Firstly, the value 1 will be associated with the local variable `i`. This is the initialisation step of the for-statement. Secondly, a test will be made to see of the current values of `i` and `n` satisfy the condition `i <= n`. In the present case, the condition yields true.

```
int fac( int n ) {
  int i ;
  int accu = 1 ;                       n : 3
  for( i = 1; i <= n; i++ ) {       ←  i : 1      (Step 4)
    accu = accu * i ;                  accu : 1
  }
  return accu ;
}
```

At Step 5 control is transferred to the body of the for-statement. The assignment statement in the body updates the value associated with the variable `accu`. As $1 \times 1 = 1$ this cannot be seen in the rendering of the store. At step 6 the control variable is incremented to 2.

```
int fac( int n ) {
  int i ;
  int accu = 1 ;
  for( i = 1; i <= n; i++ ) {
    accu = accu * i ;
  }
  return accu ;
}
```

← n : 3
i : 2
accu : 1 (Step 6)

At Step 7 control is transferred back to the beginning of the for-statement. The initialisation is not re-executed, but the test `i <= n` is re-evaluated. It yields true again.

```
int fac( int n ) {
  int i ;
  int accu = 1 ;
  for( i = 1; i <= n; i++ ) {    ← (Step 7)
    accu = accu * i ;
  }
  return accu ;
}
```

At Step 8 the body of the for-statement is re-executed, so that the value associated with `accu` is updated. At Step 9 the value associated with the control variable is incremented to 3 and at Step 10 control is transferred back to the beginning of the for-statement. The condition yields true again.

```
int fac( int n ) {
  int i ;
  int accu = 1 ;
  for( i = 1; i <= n; i++ ) {
    accu = accu * i ;
  }
  return accu ;
}
```

← n : 3
i : 3
accu : 2 (Step 10)

Steps 11, 12 and 13 make another round though the for-loop, producing the state below. Control has been transferred back to the beginning of the for-statement for the last time. This time the condition yields false. The for-statement has terminated.

```
int fac( int n ) {
  int i ;
  int accu = 1 ;                        ┌──────────┐
  for( i = 1; i <= n; i++ ) {  ←        │ n : 3    │
    accu = accu * i ;                   │ i : 4    │ (Step 13)
  }                                     │ accu : 6 │
  return accu ;                         └──────────┘
}
```

At Step 14 the final value associated with `accu` is returned.

```
int fac( int n ) {
  int i ;
  int accu = 1 ;
  for( i = 1; i <= n; i++ ) {
    accu = accu * i ;
  }
  return accu ;                ←  │ return 6 │ (Step 14)
}
```

This concludes the study of the behaviour of the for-statement using the optimised factorial function. As expected, the amount of space that is used during the execution is constant, which is the reason why this version is more efficient than the recursive version.

The increasing and decreasing for-schemas form another useful technique that will help to build efficient and idiomatic C implementations for commonly occurring patterns of computation.

3.4.2 Folding from the right

The for-schemas of Section 3.4 are based on the use of `foldl`. This is one of two common folding functions: `foldl` folds from the left. Its dual `foldr` folds from the right.

```
(* foldr : ( a-> b-> b) ->  b ->  a list ->  b *)
fun foldr f r []      = r
  | foldr f r (x::xs) = f x (foldr f r xs) ;
```

The difference between `foldl` and `foldr` can be visualised as shown below. Here we are using a three element list for convenience. The binary folding operator is $\oplus$

and the starting value for the folding process is b.

$$\begin{array}{rcccccccc}
\texttt{foldl} \oplus b & (x_1 & :: & (x_2 & :: & (x_3 & :: & [\,] &))) \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & & \\
= (((b \oplus & x_1) & \oplus & x_2) & \oplus & x_3) & & &
\end{array}$$

by contrast

$$\begin{array}{rcccccccc}
\texttt{foldr} \oplus b & (y_1 & :: & (y_2 & :: & (y_3 & :: & [\,] &))) \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
= & (y_1 & \oplus & (y_2 & \oplus & (y_3 & \oplus & b &)))
\end{array}$$

Folding from the right by `foldr` gives rise to a second set of two for-schemas:

```
(*SML foldr schema*)
  (* f : t_1 -> ... -> t_j -> t_r *)
  fun f x_1 ... x_j
      = foldr g b (m -- n) ;
```

As before, the $x_1 \ldots x_j$ are variables of types $t_1 \ldots t_j$. The g is an expression that represents a function of type `int ->` t_r `->` t_r, the b is of type t_r, and the m and n are expressions of type `int`. The variables $x_1 \ldots x_j$ may occur in the expressions g, b, m, and n. The function name f should not occur on the right hand side of the equals sign.

Again, there are two cases depending on the ordering of m and n. Consider the case that $m > n$. The C implementation of the foldr-schema now starts from the right and increases the index i:

```
/*C increasing right folding for schema*/
  t_r f( t_1 x_1, ... t_j x_j ) {
    int i ;
    t_r a = b ;
    for( i = n; i <= m; i ++ ) {
      a = g( i, a ) ;
    }
    return a ;
  }
```

The differences between the increasing left and right folding versions of the for-statement are:

- The roles of m and n are swapped
- The arguments of the accumulating function g are swapped.

In the case that $n > m$, changes have to be made similar to those discussed in the previous section for the left folding for-statement. In total, there are four different cases of folding, each with its own C for-statement. The key elements are the following:

	$m < n$	$m > n$

from n
to m
	`foldr` g b `(`m `--` n`)`
	`for(` i `=` n`;` i `>=` m`;` i`--)`
		`foldr` g b `(`m `--` n`)`
		`for(` i `=` n`;` i `<=` m`;` i`++)`

from m
to n
	`foldl` g b `(`m `--` n`)`
	`for(` i `=` m`;` i `<=` n`;` i`++)`
		`foldl` g b `(`m `--` n`)`
		`for(` i `=` m`;` i `>=` n`;` i`--)`

The differences between the four versions of the for-schema may seem small, but they are significant. Programming errors caused by the use of an incorrect for-statement are usually difficult to find. Errors, such as the inadvertent swapping of the upper and lower bounds, cannot be detected by the compiler, since both bounds will have the same type. Painstaking debugging is the only way to track down such errors. The four schemas that we have presented provide guidance in the selection of the appropriate for-statement and should help to avoid many programming errors.

Exercise 3.12 In the previous chapter we introduced the function `repeat` to capture the common behaviour of the operators $\sum$ and $\prod$. In this chapter we are using a different capture of the common behaviour, using `foldr`, `foldl` and the `--` operator. Can you think of a good reason for this?

3.5 Generalizing loops and control structures

The schemas presented so far show how we can transform SML programs systematically into equivalent C programs. In this chapter, the schemas transform recursion into loops, thus eliminating a source of inefficiency. The use of the schemas that we have presented so far has two disadvantages that we will address in this section.

The first disadvantage is that the for-schemas of Sections 3.4 and 3.4.2 are restricted because they deal with a single arithmetic sequence only. We will alleviate this restriction by allowing more general expressions in the place of the arithmetic sequence.

The second disadvantage is that, when larger programs are implemented using the for-schemas of Sections 3.4 and 3.4.2, the resulting C code is not particularly idiomatic because all the functions that result from the schemas are smaller than the functions that are typically found in C programs. The reason for this is that where in the functional paradigm one uses a number of functions to describe a certain behaviour, in C one often writes a larger function using using a number of loops and if-statements. The iterative functions that we have created so far contain at most one loop. In this section, we will show how to create larger C functions.

As a first example program, we will show the development of a function that computes the sum of the squares of the numbers $1, 2, \ldots$. As a second, more involved example, we develop a function to decide whether a number is perfect. As a final example, we compute a table of powers and add all the elements of the table.

3.5.1 Combining foldl with map: sum of squares

Given a positive natural number n, consider the problem of summing all the squares from $1 \ldots n$:

$$\begin{aligned} \text{sum\_of\_squares} &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{sum\_of\_squares}(n) &= \sum_{i=1}^{n} i^2 \end{aligned}$$

The SML solution can be written on the basis of the following observations:

1. Generate all numbers $1 \ldots n$. This requires an arithmetic sequence (1 -- n).

2. Square each number. This is conveniently done by mapping a function square over all elements of the arithmetic sequence using the standard function map.

3. Sum the squares. For this, we use the standard function sum.

Here is the auxiliary function square:

```
(* square : int -> int *)
fun square x = x * x : int ;
```

For ease of reference, we give the definitions of the standard functions map and sum and an auxiliary function add, which is the curried version of the addition operator +:

```
(* sum : int list -> int *)
fun sum xs = foldl add 0 xs ;


(* add : int -> int -> int *)
fun add x y = x + y : int ;


(* map : ( a-> b) ->  a list ->  b list *)
fun map h []      = []
  | map h (x::xs) = h x :: map h xs ;
```

With these preparations, the SML version of sum_of_squares can now be written as follows:

```
(* sum_of_squares : int -> int *)
fun sum_of_squares n
    = sum (map square (1 -- n)) ;
```

To derive a C implementation of sum_of_squares, we need to be able to match its SML definition to the foldl-schema. In its present form, sum_of_squares does not match the foldl-schema because of the absence of a call to foldl and the presence of the call to map. The first problem is easy to solve by substituting the definition of sum. This exposes the foldl:

```
(* sum_of_squares : int -> int *)
fun sum_of_squares n
    = foldl add 0 (map square (1 -- n)) ;
```

The second problem is solved by using one of the standard equations that relate `foldl` and `map`. It has a more general form than we need right now, but it is often a good idea to try to state and prove as general a result as possible. When given a function h such that:

$$\texttt{fun}\ h\ b\ x\ =\ g\ b\ (f\ x)$$

Then the following equation is true for all finite lists xs and all f, g and b:

$$\texttt{foldl}\ g\ b\ (\texttt{map}\ f\ xs)\ =\ \texttt{foldl}\ h\ b\ xs \tag{3.4}$$

Exercise ⋆ 3.13 Prove (3.4) by induction on the length of the list xs.

To make use of the equation (3.4), we need to identify the functions f, g and h. Let us put the essential ingredients in a table of correspondence so that it is possible to see the relationships:

equation (3.4)	sum_of_squares
f	square
g	add
h	add_square
fun h b x = g b (f x)	fun add_square b x = add b (square x)

The net result of using the correspondences of the table above is:

```
(* add_square : int -> int -> int *)
fun add_square b x = add b (square x) ;

(* sum_of_squares : int -> int *)
fun sum_of_squares n
    = foldl add_square 0 (1 -- n) ;
```

This last version of `sum_of_squares` is implemented in C using the left folding increasing for schema:

```
int add( int x, int y ) {
  return x + y ;
}

int square( int x ) {
  return x * x ;
}

int add_square( int b, int x ) {
  return add( b, square( x ) ) ;
}

int sum_of_squares( int n ) {
  int i ;
```

```
    int accu = 0 ;
    for( i = 1; i <= n; i++ ) {
      accu = add_square( accu , i ) ;
    }
    return accu ;
  }
```

As a finishing touch, we can inline the body of add and square in that of add_square and then inline the body of add_square in sum_of_squares. In this case, the inlining is straightforward as the C functions add, square, and add_square all contain just a return-statement. Here is the result of the inlining:

```
  int sum_of_squares( int n ) {
    int i ;
    int accu = 0 ;
    for( i = 1; i <= n-1; i++ ) {
      accu = accu + (i * i) ;
    }
    return accu ;
  }
```

This shows that we have achieved the two goals. Firstly, we have a way of dealing with an expression that is not just a left folding over an arithmetic sequence, so we have generalised the for-schema. Secondly, we have created a slightly larger C function than we have been able to do so far.

3.5.2 Combining foldl with filter: perfect numbers

A positive natural number n is a perfect number if it is equal to the sum of those factors that are strictly less than n itself. Examples of perfect numbers are 6 ($= 1+2+3$) and 28 ($= 1+2+4+7+14$). The problem that we are about to solve is to write a function that determines whether its argument is a perfect number. Here is the specification that a perfect number n has to satisfy:

$$\begin{aligned} \text{perfect} &: \mathbb{N} \rightarrow \mathbb{B} \\ \text{perfect}(n) &= n = \sum\{1 \leq i < n \mid n \bmod i = 0\} \end{aligned} \tag{3.5}$$

Let us develop the solution by top-down program design: Assume that we already have a function sum_of_factors to compute the sum of those factors of a number that are strictly less than the number itself. Then, the first part of the solution is:

```
  (* perfect : int -> bool *)
  fun perfect n = n = sum_of_factors n ;
```

The second problem is to compute the sum of the factors. This problem has three parts, each corresponding to the three important elements of the problem specification (3.5):

1. Generate all potential factors of n which are strictly less than n. This requires an arithmetic sequence (1 -- n-1).

2. Remove all numbers `i` from the list of potential factors that do not satisfy the test `n mod i = 0`, as such `i` are relatively prime with respect to `n`. The standard `filter` function removes each element for which `rel_prime` returns true.

3. All remaining elements of the list are now proper factors, which can be summed using the standard function `sum`.

The results of the problem analysis give rise to the following program:

```
(* rel_prime : int -> int -> bool *)
fun rel_prime n i = n mod i = 0 ;

(* sum_of_factors : int -> int *)
fun sum_of_factors n
    = sum (filter (rel_prime n) (1 -- n-1)) ;
```

For ease of reference, here is the definition of the standard function `filter`:

```
(* filter : ( a->bool) ->  a list ->  a list *)
fun filter p []      = []
  | filter p (x::xs) = if p x
                          then x :: filter p xs
                          else filter p xs ;
```

As in the previous section, to derive a C implementation of `sum_of_factors` we need to be able to match its SML definition to the foldl-schema. Substituting the definition of `sum` exposes the `foldl`:

```
(* sum_of_factors : int -> int *)
fun sum_of_factors n
    = foldl add 0 (filter (rel_prime n) (1 -- n-1)) ;
```

To remove the `filter` we use an equation relating `foldl` and `filter`. This equation is similar tho that used to remove the `map` earlier. When given a function h such that:

$$\texttt{fun}\ h\ b\ x\ =\ \texttt{if}\ p\ x\ \texttt{then}\ g\ b\ x\ \texttt{else}\ b$$

Then the following equation holds for all finite lists xs and all p, g, and b:

$$\texttt{foldl}\ g\ b\ (\texttt{filter}\ p\ xs)\ =\ \texttt{foldl}\ h\ b\ xs \tag{3.6}$$

Exercise ⋆ 3.14 Prove (3.6) by induction on the length of the list xs.

To use (3.6) requires us to identify the predicate p and the functions g, and h of the equation. Here is the table of correspondences:

equation (3.6)	`sum_of_factors`
p	`(rel_prime n)`
g	`add`
h	`add_rel_prime n`

`fun` $h\ b\ x$	`fun add_rel_prime n b x`
	`    = if (rel_prime n) x`
	`         then add b x`
	`         else b`

The definition of add_rel_prime above is quite interesting. The role of the schematic variable p is played by the *partial application* (rel_prime p). Now we should ask ourselves the question: where does the n come from? If we were to try the following definition, the SML compiler would complain about the variable n being undefined:

```
(* add_rel_prime : int -> int -> int *)
fun add_rel_prime b x
    = if (rel_prime n) x
         then add b x
         else b ;
```

The solution to this problem is to make n an extra argument of add_rel_prime, as we have shown in the table of correspondence above.

We are now able to write a new version of sum_of_factors consisting of an arithmetic sequence and a call to foldl:

```
(* add_rel_prime : int -> int -> int -> int *)
fun add_rel_prime n b x
    = if (rel_prime n) x
         then add b x
         else b ;

(* sum_of_factors : int -> int *)
fun sum_of_factors n
    = foldl (add_rel_prime n) 0 (1 -- n-1) ;
```

The extra argument n to add_rel_prime is now passed explicitly by the partially applied call (add_rel_prime n). The latest version of sum_of_factors conforms to the foldl-schema and can be translated directly into C:

```
bool rel_prime( int n, int i ) {
  return n % i == 0 ;
}

int add_rel_prime( int n, int b, int x ) {
  if( rel_prime( n, x ) ) {
    return add( b, x ) ;
  } else {
    return b ;
  }
}

int sum_of_factors( int n ) {
  int i ;
  int accu = 0 ;
  for( i = 1; i <= n-1; i++ ) {
    accu = add_rel_prime( n, accu, i ) ;
  }
  return accu ;
}

bool perfect( int n ) {
  return sum_of_factors( n ) == n ;
}
```

As a finishing touch, we substitute the bodies of the functions `rel_prime`, `add`, `add_rel_prime`, and `sum_of_factors` in `perfect`. This is not completely trivial, inlining `add_rel_prime` in the function `sum_of_factors` requires us to replace both return statements with assignments to `accu`. This results in the following function:

```
int sum_of_factors( int n ) {
  int i ;
  int accu = 0 ;
  for( i = 1; i <= n-1; i++ ) {
    if( rel_prime( n, i ) ) {
      accu = add( accu, i ) ;
    } else {
      accu = accu ;
    }
  }
  return accu ;
}
```

The else branch of the `if` statement is not useful (it states that `accu` should not be modified), so this branch can be removed safely. The functions `add` and `rel_prime` can also be inlined, and the whole function can then be inlined in `perfect`, resulting in the compact C function below. The function contains a for loop and a conditional, and is thus slightly larger than the functions delivered by a straight application of a for schema.

```
bool perfect( int n ) {
  int i ;
  int accu = 0 ;
  for( i = 1; i <= n-1; i++ ) {
    if( n % i == 0 ) {
      accu = accu + i ;
    }
  }
  return accu == n ;
}
```

We have come a long way in this process, for we have derived this function in a systematic way. We could have made a short-cut by using a separate schema that recognises the combination of `foldl` and `filter` explicitly. The if-statement then ensures that the body of the for-loop is only executed for those values of `i` where `rel_prime` succeeds. This is what one would guess intuitively. The interesting point of the above reasoning is that it can be applied to the general case. Without defining any more schema's, for-loops, while-loops, if-statements, and functions can be combined.

The substitution of C functions as shown above is not always valid. This is only safe where pure functions are considered. Functions that rely on side effects, which will be shown in Chapters 4 and 5, do not always allow for safe substitution.

3.5.3 Nested for statements

Nested recursion, that is, recursion within recursion, is common. It is worthwhile to investigate whether the for-schema can be applied to deal with this. Consider the following problem. Given two natural numbers n and m, generate a table of the powers n^m of all numbers from $0 \ldots n$ and $0 \ldots m$. For $n = 4$ and $m = 5$, this yields the following table:

		$k \rightarrow m$					
	i^k	0	1	2	3	4	5
	0	1	0	0	0	0	0
i	1	1	1	1	1	1	1
$\downarrow$	2	1	2	4	8	16	32
n	3	1	3	9	27	81	243
	4	1	4	16	64	256	1024

We want to compute the sum of all these powers, 1799 in the case of the above table. In general, the sum is:

$$\begin{aligned} \text{sum\_of\_sum\_of\_powers} &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{sum\_of\_sum\_of\_powers}(n) &= \sum_{i=0}^{n} (\sum_{k=0}^{m} i^k) \end{aligned} \tag{3.7}$$

This shows a summation within a summation. It is not difficult to find a formula that computes the same answer using a single summation, but to find one that uses no summation at all is more difficult. To show how nested for-statements may be created, we will perform the double summation here.

The innermost summation of (3.7) is easily written as a function `sum_of_powers`. It will have two arguments, `m` and `i`:

```
(* sum_of_powers : int -> int -> int *)
fun sum_of_powers m i
    = sum (map (power i) (0--m)) ;
```

An integer version of the `power` function from Section 2.4 can be used here to do the actual exponentiation. The technique for dealing with a combination of `map` and `foldl` of Section 3.5.2 gives us the following C implementation of `sum_of_powers`:

```
int sum_of_powers( int m, int i ) {
  int k ;
  int accu = 0 ;
  for( k = 0; k <= m; k++ ) {
    accu = accu + power( i, k ) ;
  }
  return accu ;
}
```

The outermost summation of (3.7) also translates directly into the function `sum_of_sum_of_powers` below. This function also has two arguments, `m` and `n`.

```
(* sum_of_sum_of_powers : int -> int -> int *)
fun sum_of_sum_of_powers m n
    = sum (map (sum_of_powers m) (0--n)) ;
```

Here we have again a combination of `map` and `foldl`, so that we can create the following C implementation:

```
int sum_of_sum_of_powers( int m, int n ) {
  int i ;
  int accu = 0 ;
  for( i = 0; i <= n; i++ ) {
    accu = accu + sum_of_powers( m, i ) ;
  }
  return accu ;
}
```

The combined result of our efforts is to create two functions that together solve the problem. However, the conciseness of the mathematical specification of (3.7) is lost, even in the SML solution. C permits us to recover some of the compactness of the specification by substituting the body of `sum_of_powers` into `sum_of_sum_of_powers`. The result is an idiomatic C function with a nested for-statement:

```
int sum_of_sum_of_powers( int m, int n ) {
  int i,k ;
  int accu = 0 ;
  for( i = 0; i <= n; i++ ) {
    for( k = 0; k <= m; k++ ) {
      accu = accu + power( i, k ) ;
    }
  }
  return accu ;
}
```

3.6 Summary

The following C constructs were introduced:

Assignments In C values associated with variables can be changed. The assignment has the general form:

```
x = e
```

Here x is a variable and e is an expression; x and e must be of the same type. Usually, the assignment is used as a statement, for example `i=i+1;`. However, an assignment is in itself an expression, the value on the right hand side being the value of the expression. It is allowed to write: `b=c=0` which assigns 0 to both `b` and `c`. Using an assignment in an expression does not often improve the clarity of the program.

In addition to the ordinary assignment above, there is a family of assignment operators, `+=`, `-=`, `*=`, and so on. These operators perform a computation and then write the result back into the variable on the left hand side. For example `i+=1` is identical to `i=i+1`. Finally, there are some shortcuts, `++i` and `--i` are abbreviations for `i=i+1` and `i=i-1`.

While-loops The while loop executes statements repeatedly, until the while condition becomes false. The general form of the while loop is:

```
while( p ) {
  S
}
```

The statements S are executed for as long as the condition p does not evaluate to 0.

For-loops The for loop executes statements repeatedly, until the for condition becomes false. The general form of the for loop is:

```
for( I ; p ; N ) {
  S
}
```

Before the loop starts, I is executed, to initialise. While the condition p evaluates to true, the statements S followed by the iteration N are executed. The most frequently used forms of these loops are the incrementing and decrementing for loop:

```
for( i=0 ; i<n ; i++ ) {        for( i=n-1 ; i>=0 ; i-- ) {
  S                               S
}                               }
```

Here i stands for any integer variable, and n is the number of times that the loop is to be executed.

Breaking out of a loop The statement `break;` will cause the execution of a loop to be terminated immediately. Execution resumes after the closing curly bracket of the loop body.

From a programming principles point of view, the most important issues that we have addressed are the following:

- A good understanding of the behaviour of the store is essential before efficient C functions can be implemented. Local variables can be reused (destroying the old values). Local variables can only be used within the function in which they are declared. They exist as long as the function invocation exists.

- Side effects should be visible, and they should be localised. A function may internally use side effects to increase efficiency; however, externally it should not be distinguishable from functions that do not use side effects.

- Functions should have clear and concise interfaces. That is, if the same functionality can be obtained using fewer arguments, then this is to be preferred.

- The C programmer has to make sure that values are computed and assigned to variables before they are used. It is good programming practice to always initialise local variables.

- Tail recursive functions are preferred over non-tail recursive functions since the former can efficiently be implemented as loops. Non-tail recursive functions can often be transformed into tail recursive functions by expedient use of ad-hoc transformations, such as the accumulating argument technique.

3.7 Further exercises

Exercise 3.15 The function `sum_of_sum_of_powers` is rather inefficient if it uses a naive `power` function, that is one that performs k multiplications to compute i^k. Rewrite `sum_of_sum_of_powers` such that it completely avoids the use of a `power` function, and thus makes the least possible number of multiplications.

Exercise 3.16 Exercise 2.21 required the implementation of a `pop_count` function. Implement this function in C with a loop.

Exercise 3.17 Exercise 2.25 required the implementation of a `power_of_power` function. Reimplement the C function using loops. Try to embed the functions into each other so that you don't have an auxiliary function `power`.

Exercise 3.18 Write a C function `chess_board` with two integer arguments `width` and `height` to print out a `width`×`height` chess board using ASCII characters. If `width` $= 4$ and `height` $= 2$ the output should look as follows:

```
---------
| |X| |X|
---------
|X| |X| |
---------
```

Exercise 3.19 Using a loop, write a function that determines the root of a continuous function using the Newton-Raphson method. Use the answer of Exercise 2.26 as the starting point.

Chapter 4

Structs and Unions

The functions discussed so far operate on either integers, booleans, characters, floating point numbers, or functions. These are all primitive types in C (with restrictions) and in SML (without restrictions). Like SML, C has support for *structured* data. In its simplest form, structured data makes it possible to gather into one object a collection of previously defined values of particular types. This builds a structured data type from a collection of previously defined types. This is, for example, supported with the tuple of SML and with the `struct` of C. These so-called 'flatly structured aggregate data types' are the subject of this chapter. In subsequent chapters we will look at other forms of structured data, namely sequences of data items (arrays, Chapter 5) and recursively structured data types (lists, Chapter 6).

4.1 Structs

The primitive data structures discussed in the previous chapter make it possible to pass single numbers around. Often, one is not interested in single numbers but in a collection of numbers. An example of a collection of numbers is a point in a plane, say p. The point is represented by a tuple (p_x, p_y), where p_x is the X-coordinate and p_y is the Y-coordinate. An operation that one may wish to perform on a point is to rotate it around the origin. To rotate a point $p = (p_x, p_y)$ around the origin by 90 degrees in the clockwise direction, we compute a new point, say r, with coordinates $r_x = p_y$ and $r_y = -p_x$:

$$\begin{array}{rcl} p, r & : & (\mathbb{R} \times \mathbb{R}) \\ (r_x, r_y) & = & (p_y, -p_x) \end{array}$$

Here is the type `point` in SML:

```
type point = real * real ;
```

The SML rotation function takes a `point` and computes the rotated point as shown below. For the name of this function, we would have liked to use 'new-point-which-is-the-old-one-rotated'. This would express our intentions clearly.

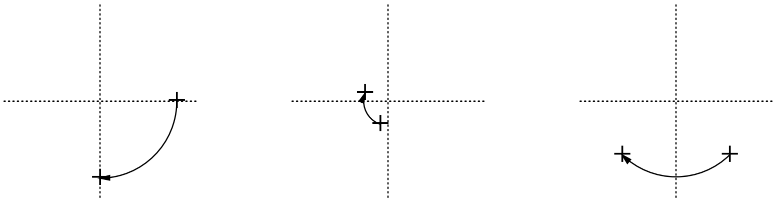
However, a long name like that is not practical, so we will use the rather conventional `rotate`.

```
(* rotate : point -> point *)
fun rotate (p_x,p_y) = (p_y,0.0-p_x) ;
```

The following equalities are all true:

```
rotate ( 1.0, 0.0) = ( 0.0,~1.0) ;
rotate (~0.1,~0.3) = (~0.3, 0.1) ;
rotate ( 0.7,~0.7) = (~0.7,~0.7) ;
```

These three rotations are shown graphically in the figure below:



C does not offer plain tuples, but it provides a more general mechanism: the `struct`. The general form of a struct declaration is:

```
struct {
  t_0 m_0 ;
  t_1 m_1 ;
  ...
}
```

Here the t_i are the types and the m_i are the names of the structure components, also known as *members*. A `typedef` can be used to associate a type name with the structured type. The declaration of a struct that contains the two coordinates of a point reads:

```
typedef struct {
  double x ;
  double y ;
} point ;
```

This declares a `struct` with two members `x` and `y`, and gives it the name `point`. The function for rotating a point should be written as follows:

```
point rotate( point p ) {
  point r ;
  r.x = p.y ;
  r.y = -p.x ;
  return r ;
}
```

The C function `rotate` has a single argument of type `point`. The return value is also of type `point`. The body of `rotate` allocates a local variable `r` of type `point` and makes two separate assignments to the members of `r`. Members are referred to using a dot-notation, so `p.y` refers to the value of the member with name `y` in the `point`-structure `p`. Therefore, the assignment `r.x = p.y` means: take the value of the y member of p and assign this value to the x member of r. The y member of r is not altered. The second assignment fills the y member of r. Finally, the value of r is returned. This is different from the SML version of rotate, which creates its result tuple in a single operation. Unfortunately, C does not have a general syntax to create a struct, in contrast with most functional languages that do have syntax to create tuples.

To print a point, a new function has to be defined, because printf has no knowledge how to print structured data. This new function print_point is shown below:

```
void print_point( point p ) {
  printf( "(%f,%f)\n", p.x, p.y ) ;
}
```

The return type of this function is void. This means it will return nothing. This function would not have a meaning in a pure functional language, as it cannot do anything. In an imperative language, such functions do have a meaning, as they can print output as a side effect. Another use of functions with a void type is shown in Section 4.4, where the different argument passing methods of C are discussed. Some imperative languages have a special notation for a function that does not return a value; such a function is called a *procedure* in Pascal or a *subroutine* in Fortran.

The body of print_point consists of a call to printf with three arguments: the format string and the values of the x and y coordinates of the point. The format string and the arguments are handled exactly as discussed in Chapter 2. The parentheses, comma, and the newline character are printed literally, while the %f parts are replaced with a numerical representation of the second and third arguments.

The function main shown below declares a local variable r of type point. Firstly, main assigns r a value corresponding to the point $(1, 0)$. It then rotates this point and prints the result, producing the output $(0, -1)$. This is followed by two similar calls to assign new values to the point r, rotate the point, and print the result.

```
int main( void ) {
  point r ;
  r.x = 1 ; r.y = 0 ;
  print_point( rotate( r ) ) ;
  r.x = -0.1 ; r.y = -0.3 ;
  print_point( rotate( r ) ) ;
  r.x = 0.7 ; r.y = -0.7 ;
  print_point( rotate( r ) ) ;
  return 0 ;
}
```

The function main shows again that C does not provide syntax to create a struct directly. However, there is one exception to this rule. In the declaration of a constant or in the initialisation of a variable, one can create a struct by listing the values of the components of the struct if all these values are constants. Using this notation, we may reformulate the main program:

```
int main( void ) {
  const point r0 = {   1, 0 } ;
  const point r1 = {-0.1, -0.3 } ;
  const point r2 = { 0.7, -0.7 } ;
  print_point( rotate( r0 ) ) ;
  print_point( rotate( r1 ) ) ;
  print_point( rotate( r2 ) ) ;
  return 0 ;
}
```

The curly bracket notation for creating a `struct` is *only* legal during the initialisation of a constant or a variable. The curly bracket notation is illegal anywhere else. In particular, it is not permitted to write `rotate( {1, 0} )` in the call to `print_point`, which would obviate the need for the local constant `r0`.

Notice the ordering of the statements of `main`. The function `main` starts with the declaration of three constants `r0`, `r1`, and `r2`. Then these three values are used in the three successive calls to `rotate`. C requires *all* constants and variables to be declared before the statements. This results in the seemingly out of order creation of the structs `r0`, `r1`, and `r2` and calls to `rotate`. Interestingly, C++ does not have this constraint. In this language variables and constants can be declared anywhere in a program, not only at the beginning of a block.

4.2 Structs in structs

A `struct` can be used not only to gather primitive data types into a flatly structured data type, but also to build more complex data structures out of those previously built. Given that we have available the type `point`, it is possible to create a new type `rectangle` by listing the coordinates of the lower left and upper right corners of the rectangle. A function that calculates the area of the rectangle would be:

$$\begin{aligned} \text{area} &: ((\mathbb{R} \times \mathbb{R}) \times (\mathbb{R} \times \mathbb{R})) \to \mathbb{R} \\ \text{area}((ll_x, ll_y), (ur_x, ur_y)) &= |(ur_x - ll_x) \times (ur_y - ll_y)| \end{aligned} \tag{4.1}$$

The SML type `rectangle` uses `point` as follows:

```
type rectangle = point * point ;
```

Using this new data type, the SML function `area` is: (the function `absolute` is defined in Section 2.5.3):

```
(* area : rectangle -> real *)
fun area ((llx,lly),(urx,ury))
    = absolute ((urx-llx) * (ury-lly)) ;
```

In C, the type of the rectangle is defined as follows, where we have repeated the definition of point for ease of reference:

```
typedef struct {
  double x ;        /* X-coordinate of point */
  double y ;        /* Y-coordinate of point */
} point ;

typedef struct {
  point ll ;        /* Lower Left hand corner */
  point ur ;        /* Upper Right hand corner */
} rectangle ;
```

The function to calculate the area then becomes:

```
double area( rectangle r ) {
  return absolute( ( r.ur.x - r.ll.x ) *
                   ( r.ur.y - r.ll.y ) ) ;
}
```

The argument r refers to the whole rectangle, r.ll refers to the lower left hand corner, and r.ll.x refers to the X-coordinate of the lower left hand corner. Similarly, r.ur.x refers to the X-coordinate of the upper right hand corner.

In this example both point and rectangle were defined as a new type, using typedef. It is equally valid only to define the type rectangle, and to embed the structure storing a point in it:

```
typedef struct {
  struct {
    double x ;        /* X-coordinate of point */
    double y ;        /* Y-coordinate of point */
  } ll, ur ;          /* Lower left & Upper right */
} rectangle ;
```

This does not define a type for point, hence points cannot be passed on their own (because they have no type with which to identify them). In general, one will embed a structure (as opposed to defining a separate type), when the structure on its own does not have a sensible meaning.

4.3 Unions in structs: algebraic data types

A tuple can be used to store a collection of values, but a tuple does not suggest how the type is being used in the program. If one has a tuple of two integers, for example (7,4), then this tuple can encode a date (the 7th of April, or the 4th of July, depending on the country you are in), but it could also encode a time (7:04 pm), or a fractional number ($\frac{7}{4}$). Algebraic data types are used instead of tuples when a particular data type may have different interpretations and/or alternative forms. For example, the following type can be defined in SML to encode a distance in either Ångströms, miles, kilometres, or light years.

```
datatype distance = Angstrom  of real |
                    Mile      of real |
                    Kilometre of real |
                    Lightyear of real ;
```

Using the data type `distance` we can write a conversion function that takes any one of the four different possibilities and calculates the distance in millimetres. Pattern matching is used to select the appropriate definition of `mm`:

```
(* mm : distance -> real *)
fun mm (Angstrom x)    = x * 1E~7
  | mm (Kilometre x)   = x * 1E6
  | mm (Mile x)        = x * 1.609344E6
  | mm (Lightyear x)   = x * 9.4607E21 ;
```

Adding distances that are specified in different units is now possible, for example:

```
mm (Mile 1.0) + mm (Kilometre 4.0) = 5609344.0 ;
```

C does not have a direct equivalent of the algebraic data type. Such a type must be composed using the three data structuring tools `enum`, `struct`, and `union`. The `enum` (Section 2.2.6) will enumerate the variants, the `struct` (Section 4.1) will hold various parts of the data, the `union`, defined below, allows to store variants.

4.3.1 Union types

The general form of the `union` definition is similar to that of a `struct` definition:

```
union {
  $t_0$ $m_0$ ;
  $t_1$ $m_1$ ;
  ...
}
```

Here the t_i are the types and the m_i are the names of the members of the `union`. The difference between a `struct` and a `union` is that the latter stores just one of its members at any one time, whereas the former stores all its members at the same time. A `struct` is sometimes called a product of types, while a `union` is a sum of types. In SML, we have the same distinction: the alternatives in an algebraic data type declaration (as separated by vertical bars) represent the sum type, and the types within each alternative represent a product type.

The storage model can be used to visualise the differences between a `struct` and a `union`. Assume that we have a structure `pair` which is defined as follows:

```
typedef struct {
  int x ;
  int y ;
} pair ;
```

Below are a structure (`structx`) and a union (`unionx`), with their respective storage requirements:

```
typedef struct {
  int i ;
  double d ;
  pair p ;
} structtype ;

structtype structx ;
```

```
typedef union {
  int i ;
  double d ;
  pair p ;
} uniontype ;

uniontype unionx ;
```

`structx:`

`i`
`d`
`p.x`
`p.y`

`unionx:`

`i`, `d`, or `p.x`
⊥ or `p.y`

In the data structure `structx`, an integer value `i`, a double precision floating point number `d`, and a pair `p` are stored *at the same time*. The data structure `unionx` either stores an integer `i`, or a floating point value `d`, or a pair `p` at any one time. The struct is thus larger than the union, as it must reserve space for all its members. The union is as large as its largest member, which is the type `pair` in this example.

Exercise 4.1 The example shown above assumes that the member `d` of the type `double` occupies one storage cell. Often, a double occupies two storage cells. Draw the storage requirements of `structx` and `unionx` under the assumption that the floating point number `d` is stored in two storage cells.

The union of C does not provide type security. Suppose that the type `uniontype` as defined earlier is used as follows:

```
int abuse_union( double x ) {
  uniontype y ;
  y.d = x ;
  return y.i ;
}
```

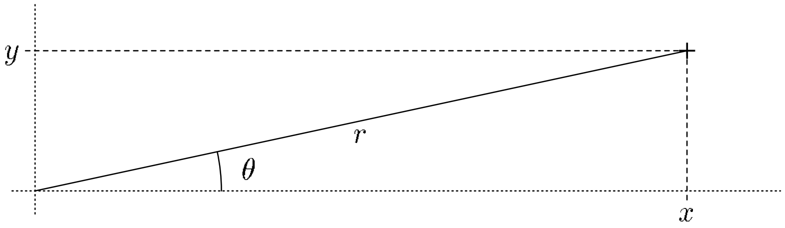
When `abuse_union( 3.14 )` is executed, it will first create a variable `y` of type `uniontype` and write the value 3.14 in the `d` member of the union. Then the value of the `i` member of the union is returned. This is legal C, but logically it is an improper operation. Probably, the function `abuse_union` will return part of the bit pattern of the binary representation of the number `3.14`, but the precise value will depend on the peculiarities of the implementation.

To use a `union` properly, the programmer must keep track of the types that are stored in the unions. Only the type that has been stored in a `union` should be retrieved. In general, the programmer has to maintain a *tag* to specify what is stored in a `union`. Because this tag has to be stored together with the `union`, both are conveniently embedded in a `struct`. This recipe results in the following C data type for maintaining a distance:

```
typedef enum { Angstrom,  Mile,
               Kilometre, Lightyear } distancetype ;

typedef struct {
  distancetype tag ;
  union {
    double angstroms ;
    double miles ;
    double kilometres ;
    double lightyears ;
  } contents ;
} distance ;
```

The possible distance types are enumerated in the type `distancetype`. The structure that holds a `distance`, has two members. The first member is the tag that specifies the type of distance being stored. The second member is the `union` that holds the values for each of the possible distances.

Exercise 4.2 There are two common notations for a point in a 2-dimensional space: Cartesian coordinates and polar coordinates. Cartesian coordinates give a pair (x, y), the polar coordinates a pair (r, θ). The meaning of (x, y) and (r, θ) are:



Define a data type `coordinate` (both in SML and C) that can hold a point in space in either of these coordinate systems.

Exercise 4.3 Give a C function to convert a polar coordinate into a Cartesian coordinate and give another C function to do the opposite. Are your functions each other's inverse? If not, can you find a coordinate that when converted sufficiently many times between polar and Cartesian 'drifts' away from the initial coordinate?

4.3.2 Pattern matching: the switch statement

SML functions using algebraic data types need pattern matching to select which data is stored. In C, this pattern matching is conveniently implemented using a *switch statement*. The switch selects one out of many options in one operation:

```
switch( e ) {
  case c₁ : S₁ ;
  case $c_2$ : $S_2$ ;
  ...
  default : $S_d$ ;
}
```

The first step of the execution of a `switch` statement is the evaluation of the expression e. After that, the value of the expression is matched against all the (constant) `case` labels c_i. If one of the constants c_i matches, the statements following it, that is S_i, are executed. Otherwise, if none of the cases match, the statements following the `default` label are executed. (The default part may be omitted, but it is good programming practice to catch unforeseen cases using the default.)

The switch can be used to define the `mm` function, that converts any distance to millimetres:

```
double mm( distance x ) {
  switch( x.tag ) {
    case Angstrom:  return x.contents.angstroms * 1e-7 ;
    case Kilometre: return x.contents.kilometres * 1e6 ;
    case Mile:      return x.contents.miles * 1.609344e6 ;
    case Lightyear: return x.contents.lightyears*9.4607e21;
    default:        abort() ;
  }
}

int main( void ) {
  distance x, y ;
  x.tag = Mile ;      x.contents.miles      = 1.0 ;
  y.tag = Kilometre ; y.contents.kilometres = 4.0 ;
  printf( "%f\n", mm( x ) + mm( y ) ) ;
  return 0 ;
}
```

The main program will add 1 mile and 4 kilometres, and print the result in millimetres (5609344).

The `switch`-statement has remarkable semantics, which a C programmer must be aware of. When the execution of the selected statements has completed, *the program will normally continue to execute the statements following the next* `case` *or* `default`. This is called *falling through*. To prevent this from happening, either a `return` or a `break` statement must be inserted in the statements that follow each case label and the default label. A `return` will terminate the enclosing function, as is the case in the function `mm` shown before. A `break` will terminate the execution of the switch statement, and continue with the first statement after the `switch`.

All case labels in a switch statement must be unique. The order of the cases is irrelevant since any expression will match at most one `case`. The `default` can also be at any place, since it only matches if none of the constants match. However, the order of the case labels is important if we wish to exploit the 'falling through' property. Consider the following obscure, but legal, C function:

```
double double_power( double r, int p ) {
  switch( p ) {
    case 4 : r = r * r ;
    case 3 : r = r * r ;
    case 2 : r = r * r ;
    case 1 : r = r * r ;
  }
  return r ;
}
```

The function double_power calculates r^{2^p} for $0 \leq p \leq 4$ as each successive multiplication is executed in turn. It is considered bad programming style to write this kind of program. However, falling through is acceptable when several cases need to be handled in precisely the same way. Here is a rather contrived example of good use of falling through. The function legs counts the number of legs of the species given:

```
typedef enum { Ant, Ecoli, Lizard,
               Cat, Stork, Fuchsia } species ;

int legs( species s ) {
  switch( s ) {
    case Ant:    return 6;
    case Cat:
    case Lizard: return 4;
    case Stork:  return 2;
    default:     return 0;
  }
}
```

The expression legs(Cat) evaluates to 4, just like legs(Lizard). The expression legs(Fuchsia) returns 0.

4.3.3 Algebraic types in other imperative languages

From a theoretical point of view the struc and union of C are well designed. The language offers product types (the struct) and sum types (the union). These are essentially separate concepts, so the language should have separate notations for these concepts. However, from a programming point of view, it is not so clear that these two concepts should be kept separate at all times. The problem is, that as we have seen, there is nothing in a sum type (the union) to help us distinguish between the various cases. The theorist has two notions of sum types: the disjoint sum and the coalesced sum. The disjoint sum makes it possible to distinguish between the different variants of the sum type; the coalesced sum makes this impossible. Such a distinction could have been made in C but it has not been.

There is only one way to safely use a union in C, that is when the union is embedded in a struct. In this context we can store, together with the union itself, the tag that remembers which of the variants of the union is stored. The designers of

ALGOL-68 and Pascal found this such an important concept that they introduced a construction that specifies both the variants and the tag, all at once. As an example we show the C and Pascal definitions of the `point` data structure:

```
  typedef struct {           type point = record
    double x ;                 x: real;
    double y ;                 y: real;
  } point ;                  end;
```

In Pascal, a floating point number is of type `real`. The words in a member declaration in Pascal are written a different order. The enumerated types of Pascal and C are similar:

```
  typedef enum               type coordinatetype =
   { Polar, Cartesian }        ( Polar, Cartesian ) ;
     coordinatetype;
```

The main difference between the two languages with respect to building structured data is that the Pascal `record` has a second use: it can have variants, which correspond to the C struct/union combination. Here is the correspondence between the two languages:

```
  typedef struct {           type coordinate = record
    coordinatetype tag ;       case tag : coordinatetype
    union {                    of
      struct {                   Cartesian:
        double x ;               (   x : real;
        double y ;                   y : real;
      } cartesian ;              );
      struct {                   Polar:
        double r ;               (   r : real;
        double theta ;               theta : real;
      } polar ;                  );
    } contents ;
  } coordinate ;             end ;
```

The Pascal `coordinate` record always has a `tag` member, which discriminates between the `Cartesian` and the `Polar` coordinate system. If the `tag` is `Cartesian`, then only the `tag`, `x`, and `y` members are logically accessible. A similar restriction holds for `Polar`. The compiler can be designed to make sure that the restriction is enforced.

In the language C++, it was also recognised that a `union` is often part of a structure. Hence, it was decided that a C++-`union` may be *anonymous*. In the above example, the identifier `contents` can be omitted so that the union may be defined as:

```
typedef struct {
  coordinatetype tag ;
  union {
    struct {
      double x ;
      double y ;
    } cartesian ;
    struct {
      double r ;
      double theta ;
    } polar ;
  } ;
} coordinate ;
```

The definition in C++ of a variable `c` of the type `coordinate` makes it possible to access the elements of `c` with `c.tag`, `c.cartesian.x`, `c.cartesian.y`, `c.polar.r`, and `c.polar.theta`. This is opposed to the way the components are accessed in C, using `c.contents.cartesian.x`. The use of tags is not mandatory in C++, so unions like the one shown above are still unsafe.

4.4 Pointers: references to values

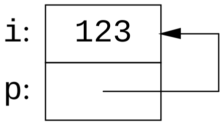
Until this moment, every data structure has been handled *by value*, in exactly the same way as SML and other functional languages treat data structures. Handling large data structures by value can be inefficient, as they must often be copied. In SML, the compiler can optimise the code so that large data structures are passed *by reference* instead of by value. In C, the programmer must explicitly use a call by reference mechanism if data structures are not to be copied.

In C, the call-by-reference mechanism is based on manipulating, not the data itself, but *pointers* to the data. A pointer is a reference to a place where data is stored, it is pointing to some data location. In terms of a von Neumann machine, a pointer is the address of a memory location. A pointer is itself a value, so that one can perform operations on a pointer as well as on the data.

The pointer concept provides the means to implement any conceivable operation on data, but the mechanism itself is of a low level of abstraction. Its proper use is more difficult to master than the use of data structures in functional languages, but the rewards in terms of increased efficiency may be considerable. After explaining the basic principles of pointers, an example program will show how pointers are used to improve efficiency.

4.4.1 Defining pointers

Consider an example, where we have an integer `i` and a pointer `p` which refers to `i`. In terms of a storage model, this can be visualised as follows:



The variable i is an integer with a current value 123; the variable p is a pointer which refers to i. We say that p is a pointer referring to an integer, and the type of p is 'pointer to integer'. In C, this type is denoted as int *. The asterisk * after the base type int indicates that this is a pointer to an int. The function pointer_example contains declarations for i and p and initialises them:

```
void pointer_example( void ) {
  int i = 123 ;
  int *p = &i ;
  printf("i: %d, *p: %d\n", i, *p ) ;
}
```

The variable p is initialised with the value &i. The prefix & is the *address*-operator, it returns a reference to the variable, or, in terms of a von Neumann model, the address of the memory cell where i is stored. The prefix-& can only be applied to objects that have an address, for example a variable. The expression &45 is illegal, since 45 is a constant and does not have an address.

The call to printf prints two integer values, the values of i and *p. The prefix-* operator is the *dereference*-operator. It takes a pointer and returns the value to which the pointer is pointing. It is the inverse operation of & and may only be applied to values with a pointer type. Thus *i is illegal, as i is an integer (not a pointer), and *3.14e15 is also illegal as 3.14e15 is a floating point number and not a pointer. The output of the function pointer_example will be:

```
i: 123, *p: 123
```

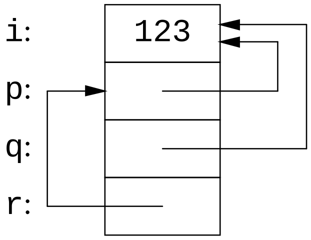
Note that it is essential for i to be a variable, and not a const; it is illegal to apply the address operator to something that is denoted const. As with any other type, a typedef can be used to define a type synonym for pointers. The type pointer_to_integer is defined below to be a synonym for a pointer to an int:

```
typedef int * pointer_to_integer ;
```

The general form of a pointer type is base_type *. The asterisk denotes 'pointer to'. The base_type may be any type, including another pointer type. This means that we can use a type int **, which denotes a pointer to a pointer to an integer. The asterisks bind from left to right, so this type is read as (int *)* or a pointer to (a pointer to an integer). An extended function uses two more pointers, q and r:

```
void extended_pointer_example( void ) {
  int   i = 123 ;
  int  *p = &i ;
  int  *q = &i ;
  int **r = &p ;
  printf("i: %d, *p: %d, *q: %d, **r: %d\n", i, *p, *q, **r ) ;
}
```

Just before the `printf` function is called, the storage of `i` … `r` is initialised as follows:



The variable `q` is a pointer to an `int` and points to the same storage cell as `p`. The variable `r` is a pointer to a pointer to an `int`; it is initialised to point to `p`. In the call to `printf`, we find the argument `**r`. The dereference operator `*` is applied twice in succession, once to `r`, resulting in `*r`, the value to which `r` was pointing. Subsequently `*r` is dereferenced resulting in `*(*r)` which results in the value of `i`. The function above will print the output:

```
i: 123, *p: 123, *q: 123, **r: 123
```

It is important to fully appreciate the meaning of the `&` and `*` operator. Note that `&i` has the same value as `p` and `q` and that `&p` has the same value as `r`, but that `&q` has a value different from `r`.

Exercise 4.4 Define a type `pointer_to_pointer_to_double`.

Exercise 4.5 What is the meaning of the types `type0` and `type1` in the following type synonyms?

```
typedef struct {
  double *x ; int y ;
} type0 ;
typedef type0 *type1 ;
```

Exercise 4.6 Consider the following:

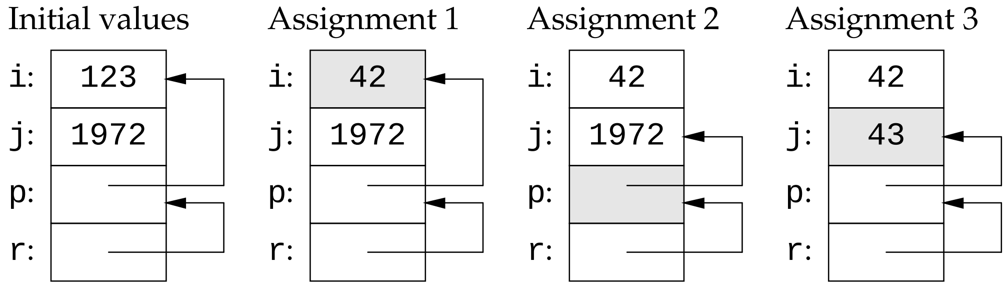
(a) Is `*&i` the same as `i` (given that `i` is a variable of type `int`)?
(b) Is `*&`x the same as x for all x?
(c) Is `&*p` the same as `p` (given that `p` is a variable of type `int *`)
(d) Is `&*`z the same as z for all z?

4.4.2 Assignments through pointers

After pointers have been defined, one can not only access values through them (as shown in the previous section), but also write values through pointers. Assigning values through pointers is a low level activity, which can cause havoc in big programs. Consider the following function:

```
void pointer_assignment( void ) {
  int   i = 123 ;
  int   j = 1972 ;
  int  *p = &i ;
  int **r = &p ;
  *p = 42 ;              /* First assignment */
  *r = &j ;              /* Second assignment */
  *p = i + 1 ;           /* Third assignment */
  printf("i: %d, j: %d, *p: %d, **r: %d\n", i, j, *p, **r );
}
```

The state of the store just after the initialisation and after each of the three subsequent assignments are shown in the following figure:



At first, p is pointing to i, and r is pointing to p. The first assignment writes 42 to *p. That is, it is not written to p but to the location where p is pointing to, which is i. Thus the value of i changes from 123 to 42. The second assignment writes a value to *r. Since r is pointing to p, the value of p will be overwritten with &j, a pointer to j. The third assignment writes again to *p, because p is now pointing to j, the value of j is changed. The output is therefore:

```
i: 42, j: 43, *p: 43, **r: 43
```

By following the pointers, one can exactly determine which values are overwritten. However, viewed from a higher level, the semantics are awkward:

- Although the variable i is not assigned to anywhere in the function, its value is changed. It was 123 after the initialisation, but it has the value 42 at the end of the function. The value is changed because of the first assignment to *p.

- The first and the third assignment are both assignments to *p. Although nothing has been assigned to p in the meantime, the first assignment writes to i, while the third one writes to j. The second assignment changes p by assigning via *r.

These 'hidden' assignments are caused by *aliases*. Immediately after the initialisation, there are three ways in this program to reach the value stored in i: via i, via *p, and via **r. Because *p and **r both refer to i, they are called *aliases* of i. Aliases cause a problem when one of them is updated. Updating one of the aliases (for example *p or **r) can cause the value of all the other aliases to change. Updating aliased variables can causes obscure errors, and it is recommended only to update non-aliased variables.

In a functional language aliases do exist, but the programmer does not have to worry about them. As values cannot be updated, an alias cannot be distinguished from a copy. Other languages do allow aliases, but some do not allow them to be overwritten.

4.4.3 Passing arguments by reference

One of the main uses of pointers in C is to pass function arguments by reference, instead of by value. Rather than passing the value to the function, we pass a pointer to the value to the function. When used carefully, this has two advantages:

1. Passing a pointer to a large data structure is more efficient than passing the data structure itself. Passing a data structure by value will cause the implementation to make a copy of the data structure and to pass that to the function, while passing a data structure by reference only passes a pointer.

2. Data structures that are passed by reference can be modified. The modification of values should be performed with utmost care, as the modifications will be visible to any procedure using the data structure.

To show these two advantages, we will use the core of a personnel information system as an example. The personnel system stores the employee's number, salary, year of birth, and first year of employment in a 4-tuple:

$$\text{employee} \equiv (\mathbb{N} \times \mathbb{R} \times \mathbb{N} \times \mathbb{N})$$

The SML representation of this 4-tuple is:

```
type employee = int * real * int * int ;
```

The operation defined on the type `employee` is a function that may be used to give the employee a pay rise by a given percentage:

```
(* payrise : employee -> real -> employee *)
fun payrise (nr, salary, birth, employed) inc
    = (nr, salary * (1.0+inc/100.0), birth, employed) ;
```

The C version of the data type `employee` is the following:

```
typedef struct {
  int employee_number ;
  double salary ;
  int year_of_birth ;
  int year_of_employment ;
} employee ;
```

There are two ways to implement the function `payrise`; it can be implemented in an applicative way, or in an imperative way. Applicative means that data items, once they have been created, will not be changed. Imperative means that data items may be changed. The applicative implementation of `payrise` follows the pattern that we have seen before:

```
employee payrise_ap( employee e, double inc ) {
  e.salary = e.salary * (1 + inc/100) ;
  return e ;
}
```

The structure containing the data on employee e is modified and returned Tracing the execution of a program increasing the salary of one employee shows an inefficient usage of memory and execution time:

```
int main( void ) {
  employee e0 = { 13, 20000.0, 1972, 1993 } ;
  employee e1 ;
  e1 = payrise_ap( e0, 3.5 ) ;
  return 0 ;
}
```

←
e0 : (13,20000.0,1972,1993)
e1 : (⊥, ⊥, ⊥, ⊥)

At Step 1 the function main has initialised the variable e0 with 4 values. The struct e1 has an undefined value (literally undefined: it can be anything).

```
employee payrise_ap( employee e, double inc ) {
  e.salary = e.salary*(1+inc/100);
  return e ;
}
```

←
inc: 3.5
e : (13,20000.0,1972,1993)
e0′: (13,20000.0,1972,1993)
e1′: (⊥, ⊥, ⊥, ⊥)

At Step 2 the function payrise_ap is called with two arguments, the value of e0 and the value 3.5. On the stack, we find the variables of main, and the arguments of payrise_ap. The value of e is a complete copy of the value associated with e0 .

```
employee payrise_ap( employee e, double inc ) {
  e.salary = e.salary*(1+inc/100);
  return e ;
}
```

←
inc: 3.5
e: (13,20000.0,1972,1993)
e0′: (13, 20000.0,1972,1993)
e1′: (⊥,⊥, ⊥, ⊥)

At Step 3 the salary has been modified, and the whole structure is returned. Subsequently, the new record is copied to the variable e1:

```
int main( void ) {
  employee e0 = { 13, 20000.0, 1972, 1993 } ;
  employee e1 ;
  e1 = payrise_ap( e0, 3.5 ) ;
  return 0 ;
}
```

←
e0 : (13,20000.0,1972,1993)
e1 : (13,20700.0,1972,1993)

At Step 4 control has returned to main, and the return value of payrise_ap has been copied into e1. The inefficiency in execution time arises from the fact that the record has been copied two times (from e0 to the argument e and from e to the variable e1). In terms of memory usage, the program is also inefficient: during Steps 2 and 3, the stack contained three of the database records. The efficiency can be improved by not passing the structure, but by passing a pointer to the structure instead and by working on the data directly:

```
void payrise_im( employee *e, double inc ) {
  (*e).salary= (*e).salary * (1+inc/100) ;
}
```

The function payrise_im has two arguments. The first argument e is a pointer to a employee. The second argument inc is the percentage. The function result is void, indicating that the function performs its useful work by means of a side effect. In this case the return statement is usually omitted.

The expression on the right hand side of the assignment statement calculates the new salary of the employee. The expression (*e).salary should be read as follows: take the value of the pointer e, dereference it to *e so that we obtain the value of the whole employee record, and then select the appropriate member (*e).salary, which is the salary of the employee. This pattern of first dereferencing a pointer to a struct and then selecting a particular member of the struct is so common that C allows more concise syntax for the combined operation: e->salary.

The assignment causes the new salary value to be *written back* into the salary member of the object to which e is pointing. The structure that was passed as the function argument is *updated*, this update is the side effect of the function. A truly idiomatic C function can be created by using the -> and the *= operator (defined in Chapter 3):

```
void payrise_im( employee *e, double inc ) {
  e->salary *= 1 + inc/100 ;
}
```

Remember that a *= b is short for a = a * b. To demonstrate the greater efficiency of the use of payrise_im, we give the trace of a main program that calls the function payrise_im.

```
int main( void ) {
  employee e0 = { 13, 20000.0, 1972, 1993 } ;
  payrise_im( &e0, 3.5 ) ;
  return 0 ;
}
```

←

e0 : (13,20000.0,1972,1993)

Control starts in main, and the local variable e0 is initialised. Now payrise_im is going to be called.

```
void payrise_im( employee *e, double inc ) {
  e->salary *= 1 + inc/100 ;
}
```

←

inc : 3.5
e : Points to e0
e0′: (13,20000.0,1972,1993)

At Step 2 the function payrise_im has been called. Its first argument, e, is not a copy but a pointer to the employee structure associated with e0 . Consequently, less storage is needed, and time is also saved because the employee structure has not been copied.

```
void payrise_im( employee *e, double inc ) {
  e->salary *= 1 + inc/100 ;
}
```

←

inc: 3.5
e : Points to e0
e0′: (13,20700.0,1972,1993)

At Step 3 after the update of e->salary, the value of the variable e0 in the main program has changed. The pointer e has been used both for reading the old salary, and for updating it with the new salary.

```
int main( void ) {
  employee e0 = { 13, 20000.0, 1972, 1993 } ;
  payrise_im( &e0, 3.5 ) ;
  return 0 ;
}
```

← e0 : (13,20700.0,1972,1993)

At Step 4 control has returned to main, where the program can now inspect e0 in the knowledge that it has been updated as a side effect of payrise_im. In comparison with the applicative version shown earlier, the structure has never been copied, and at most one copy of the structure was present on the stack at any moment in time. What we have lost is clarity: given the old structure the applicative version created a new structure, and did not touch any data private to main. The imperative version modifies the local variables of main.

The SML compiler might actually generate code comparable to the imperative payrise_im. It would do so when the compiler can infer that the current employee record is no longer necessary after it has been updated. The programmer should not have to worry about the safety of this optimisation, as the compiler would not apply it when it would be unsafe to do so.

Exercise 4.7 Reimplement the program of page 102, where a point was rotated around the origin. Implement it in such a way that

1. The function print_point gets a pointer to a point as its argument.
2. The function rotate gets a pointer to a point as the argument, which is modified on return. The function should have a return type void.
3. The function main calls rotate and print_point in the appropriate way.

4.4.4 Lifetime of pointers and storage

Passing around references to values opens a large trap in C. Consider the following code fragment:

```
int *wrong( void ) {
  int x = 2 ;
  return &x ;
}

int main( void ) {
  int *y = wrong() ;
  printf("%d\n", *y ) ;
  return 0 ;
}
```

The function wrong is *legal* C; no compiler will complain about the code. The function has a local variable, initialised with the value 2, and a pointer to this value is returned as the function result. The typing of this function is correct, since x is of type int, so &x is a pointer to an integer, int *, which happens to be the return type of wrong.

To see what the problem is, the function wrong has to be traced. Upon returning from wrong, a pointer to x is returned, but when the function terminates, all local variables and arguments of the function cease to exist. That is, the storage cell that was allocated to x will probably be used for something else. This means that x no longer exists, while there is still a pointer to x. In this particular program, this pointer is stored in y in the main function. Such a pointer to data that has disappeared is known as a *dangling reference* or a *dangling pointer*. The pointer was once valid, but it is now pointing into an unknown place in memory. Using this pointer can yield any result; one might find the original value, any random value, or the program might simply crash.

To understand why a dangling pointer is created, the concept of the lifetime of a pointer must be explained. Each variable has a certain lifetime. In the case of the argument of a function, the variable lives until the function terminates. Ordinary variables that are declared inside a block of statements, for example in the function body or in the body of a while loop, are created as the block is entered and live until that block of statements is finished.

The rule to prevent a dangling pointer is: the pointer must have a lifetime that is not longer than the lifetime of the variable that is being pointed to. In the example of the function wrong, the return value of the function lives longer than the variable x, hence there is a dangling pointer. In the main program that calls payrise_im on page 118, the variable e0 lives longer than the pointer to it, which is passed as an argument to payrise_im; therefore, there is no dangling pointer.

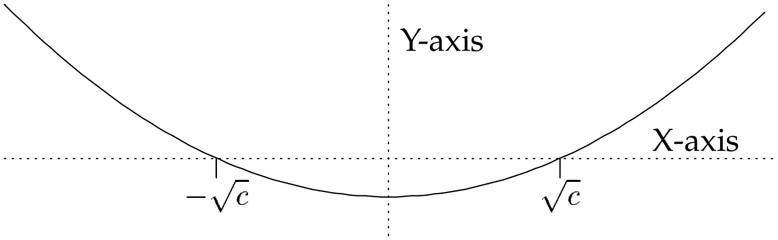
In later chapters, we will discuss other forms of storage with a lifetime that is not bound to the function invocation: dynamic memory (Chapter 5), and global variables (Chapter 8). With lifetimes that differ, dangling pointers become less obvious. In larger programs it is also difficult to find errors with dangling pointers. The symptoms of a dangling pointer may occur long after the creation of it. In the example program wrong, y was used immediately in main. It is not unusual for a dangling pointer to lay around for a long time before being used. The problems with dangling pointers are so common that there is a market for commercial software tools that are specifically designed to help discover such programming errors.

4.5 Void pointers: partial application

The functions that have been discussed so far operate on one specific type. This can be a bit restrictive; it is useful to be able to create functions that have arguments of a generic type. Pointers in C offer a mechanism that allows values of an unspecified, generic type to be passed. To see why arguments of a generic type can be useful, consider again the function bisection as discussed in Section 2.5.3. It determines the root of a function in a given interval. It is likely that the roots of a *parametrised* function may also be needed. For example, one might want to determine the roots of a family of functions, depending on some parameter c:

$$\begin{aligned} \text{parabola} &: \mathbb{R} \to \mathbb{R} \\ \text{parabola}(x) &= x^2 - c \end{aligned}$$

The function defines a parabola that has sunk through the X-axis with a depth c. The roots of this parabola are at the points $\sqrt{c}$ and $-\sqrt{c}$. (In Chapter 2, we used the function $x^2 - 2$, which had a root at $\sqrt{2}$):



As another, slightly more involved example, one might want to determine a zero of an even more general parabolic function which has the form below. This parabola is parametrised over b and c:

$$\begin{aligned} \text{quadratic} &: \mathbb{R} \to \mathbb{R} \\ \text{quadratic}(x) &= x^2 - bx - c \end{aligned}$$

In a functional language, there are two ways to solve this problem. The first solution is to use partial application of functions. The functions `parabola` or `quadratic` can be partially applied to a number of arguments before passing the result to `bisection`. To see how this is done, define the SML function `parabola` as follows:

```
(* parabola : real -> real -> real *)
fun parabola c x = x * x - c : real ;
```

The partially applied version of `parabola` with the value of `c` bound to `0.1` is written as `(parabola 0.1)`. This is a function of type `real -> real` and therefore it is fit to be supplied to `bisection`. Examples of evaluation include:

```
bisection (parabola 2.0) 1.0 2.0 = 1.4140625 ;
bisection (parabola 4.0) 1.0 4.0 = 1.999755859375 ;
```

Similar to the function `parabola`, we can write a function `quadratic` in SML as follows:

```
(* quadratic : real -> real -> real -> real *)
fun quadratic b c x = x * x - b * x - c : real ;
```

To turn `quadratic` into a function suitable for use by `bisection`, we need to partially apply it with two argument values, one for `b` and one for `c`. We can now write the following expressions:

```
bisection (quadratic  2.0 3.0) 1.0   4.0 = 3.000244140625 ;
bisection (quadratic ~2.5 8.1) 0.0 100.0 = 1.8585205078125;
```

The essential issue is that we use partial application of a function to aggregate a function and a number of its arguments into a new function. Partial application is not available in C, so we need to look for an alternative mechanism. The most straightforward alternative is to pass not only the function of interest, but also any extra arguments, as separate entities rather than as one aggregate. In both SML and C, the definition of bisection can be changed to pass the extra arguments. Let us study the solution in SML first. The revised version of bisection, called extra_bisection, is given below. It has an extra argument x, whose only purpose is to pass information to the call of the function f and to the recursive calls of extra_bisection:

```
(* extra_bisection : ( a->real->real) ->
                      a -> real -> real -> real *)
fun extra_bisection f x l h
    = let
         val m = (l + h) / 2.0
         val f_m = f x m          (* arg. x added *)
      in
         if absolute f_m < eps
            then m
            else if absolute(h-l) < delta
                    then m
                    else if f_m < 0.0
(* arg. x added *)       then extra_bisection f x m h
(* arg. x added *)       else extra_bisection f x l m
        end ;
```

The definition of parabola remains unaltered, as aggregating a single value into a single argument does not make a visible change. The following call passes the value 2.0 under the name x to the call f x m:

```
extra_bisection parabola 2.0 1.0 2.0
```

The definition of quadratic must be changed to aggregate the two arguments b and c into one new, tupled argument:

```
(* quadratic : real*real -> real -> real *)
fun quadratic (b,c) x = x * x - b * x - c : real ;
```

The call extra_bisection quadratic (2.0,3.0) 1.0 4.0 passes the tuple (2.0,3.0) under the name x to the places where the information is required.

Thus, if there is a single argument, it can be passed plain. If there are several arguments, they must be encapsulated in a tuple before they are passed. The type of the extra argument is denoted with a type variable a. A value of any type can be passed, as long as the uses of the type a are consistent. This solution is less elegant than the solution that uses partially applied functions because it requires extra_bisection to pass the extra argument around explicitly. It is this solution that we will use in C, because C has support for passing arguments of an unknown type but no support for partial application.

A C variable of type void * can hold a pointer to *anything*. For example, a pointer to an integer, a pointer to a double, a pointer to a struct, and even a pointer to a function can all be held in a `void *`. The language only insists that it is a *pointer* to something. This means that the generalised version, `extra_bisection`, can be implemented as follows:

```
double extra_bisection( double (*f)( void *, double ),
                        void * x, double l, double h  ) {
  double m ;
  double f_m ;
  while( true ) {
    m = (l + h) / 2.0 ;
    f_m = f( x, m ) ;  /* argument x added */
    if( absolute( f_m ) < eps ) {
      return m ;
    } else if( absolute( h-l ) < delta ) {
      return m ;
    } else if( f_m < 0 ) {
      l = m ;
    } else {
      h = m;
    }
  }
}
```

The argument `f` is a function that must receive two arguments: a pointer to something and a floating point number. When called, `f` returns a floating point number. The extra argument `x` of `extra_bisection` must be a pointer to something. This pointer is passed to the function `f` in the statement `m = f( arg, m ) ;`. The C versions of `parabola` and `quadratic` must also be defined in such a way that they accept the extra arguments:

```
double parabola( double *c, double x ) {
  return x * x - (*c) ;
}
```

The function `parabola` has two arguments: a pointer to a `double c` and a `double x`. To obtain the value associated with `c`, the pointer must be dereferenced using the asterisk. Note that the parentheses are not necessary; the unary `*` operator (pointer dereference) has a higher priority than the binary `*` (multiplication), but for reasons of clarity, parentheses have been introduced.

```
typedef struct {
  double b ;
  double c ;
} double_double ;

double quadratic( double_double *bc, double x ) {
  return x * x - bc->b * x - bc->c ;
}
```

The function `quadratic` also has two arguments: the first argument is a pointer `bc` to a structure that contains two doubles, the second argument is the double

x. The structure double_double is unpacked in the body of quadratic; Then bc->b is used to refer to the b argument, and bc->c refers to the c argument.

Before extra_bisection can be called, its arguments must be wrapped appropriately, as is shown in the main function below:

```
int main( void ) {
  double        c ;
  double_double dd ;
  c = 2.0 ;
  printf("%f\n", extra_bisection( parabola, /* Type error */
                                  &c, 1.0, 2.0 ) ) ;
  c = 4.0 ;
  printf("%f\n", extra_bisection( parabola, /* Type error */
                                  &c, 1.0, 4.0 ) ) ;
  dd.b = 2.0 ;
  dd.c = 3.0 ;
  printf("%f\n", extra_bisection( quadratic,/* Type error */
                                  &dd, 1.0, 4.0 ) ) ;
  dd.b = -2.5 ;
  dd.c = 8.1 ;
  printf("%f\n", extra_bisection( quadratic,/* Type error */
                                  &dd, 0.0, 100.0 ) ) ;
  return 0 ;
}
```

The arguments that are to be passed to parabola and quadratic must be prepared and stored before they can be passed. The & operator cannot be applied to constant values, hence, the values to be passed to parabola (2.0 in the first call and 4.0 in the second call) are first stored in a variable, c, whereupon the pointer to this variable (&c) is passed to extra_bisection. Similarly, the structure dd is filled with the appropriate arguments, whereupon a pointer to the structure is passed to extra_bisection.

There is one problem left in this implementation of main. The compiler will not accept the calls to extra_bisection as they stand. The reason is that the arguments parabola and quadratic are of the wrong type. The required type is this:

```
double (*)( void *, double )
```

The type offered by parabola is different; it is:

```
double (*)( double *, double )
```

The type offered by quadratic is yet something else:

```
double (*)( double_double *, double )
```

The offered types cannot be accepted because the required type is:

> A function that accepts a generic pointer.

The type offered by parabola is:

> A function that requires a pointer to a double.

The type offered by quadratic is:

> A function that requires a pointer to a double_double struct.

To solve this type mismatch problem, we have to go back to the functions parabola and quadratic to re-define them in such a way that they accept a pointer to anything indeed:

```
double parabola( void *p, double x ) {
  double *c = p ;
  return x * x - (*c) ;
}

typedef struct {
  double b ;
  double c ;
} double_double ;

double quadratic( void *p, double x ) {
  double_double *bc = p ;
  return x * x - bc->b * x - bc->c ;
}
```

The function parabola now accepts a pointer to anything. Because it requires a pointer to a double internally, it first *casts* it to a pointer to a double. Type casts from a void * to any pointer are always legal, hence the C compiler will translate the assignment of void *p to double *c without complaints. These type casts are implicit type casts, like the casts between integer and floating point numbers (see page 33) where the compiler can introduce them without harm. Similarly, the assignment from void *p to double_double *bc is without problems.

4.5.1 The danger of explicit type casts

It is enlightening to introduce explicit type casts. An explicit type cast has the following general form:

(t) x

Here t is the type to which the variable x has to be cast. Therefore (double) 2 has the same type and value as 2.0. Likewise, the assignment in parabola above could have been written:

```
double *c = (double *) p ;
```

Alternatively, the problem with the incorrect typing in main could have been solved by calling extra_bisection with:

```
extra_bisection( (double (*)(void *,double)) parabola,
                 &c, 4.0, 7.0) ;
/*initialise dd*/
extra_bisection( (double (*)(void *,double)) quadratic,
                 &dd, 4.0, 7.0 ) ;
```

This explains to the compiler that it should not bother with checking the types of `parabola` and `quadratic` because it is told what the type is intended to be. Although type-casts can be informative, they do have a serious disadvantage: the compiler is not critical about explicit type casts. Any type cast is considered to be legal, hence one could write:

```
extra_bisection( (double (*)(void *,double)) 42,
                 &dd, 1.0, 3.0 ) ;
```

The *integer number* 42 is cast to the type 'function with two arguments …' hence it is a legal call to `extra_bisection` with the integer number 42 as a function. Needless to say, the execution of this program is unpredictable, and most likely meaningless. Because explicit type casts are unsafe, we discourage their use. Instead, we recommend relying on implicit type casts of the language. Although they are not fool-proof, they are less error prone.

4.5.2 Void pointers and parametric polymorphism

There is an important difference between parametric polymorphism in a strongly typed language and the use of `void *` in C. In a strongly typed language, different occurrences of the same type variable must all be instantiated to the same type. In C, different occurrences of `void *` may represent different types, as the type `void *` stands literally for a pointer to anything. As an example, consider again the C version of `extra_bisection`, where the type `void *` appears twice in the function prototype:

```
double extra_bisection( double (*f)( double, void *),
                        void * x, double l, double h  )
```

The first and the second declaration of `void *` have no relation according to the C language definition. This should be contrasted with the type of the SML definition of the same function which is:

```
(* extra_bisection : ( a->real->real) ->
                      a -> real -> real -> real *)
```

Here the type variable `a` refers to the one and the same type. In more complex definitions, various type arguments can be used (`a`, `b`, …); in C, they are all the same, `void *`.

Type consistency is a valuable property as it prevents programming errors. In the example above, the function `extra_bisection` could have been called as follows:

```
char t ;
extra_bisection( quadratic, &t, 0, 1 ) ;
```

This is completely legal C: `&t` is a pointer to a character, so it conforms to the type `void *`. The function `quadratic` requires a pointer to a structure, so it also conforms to the type `void *`. The fact that this character pointer is going to be used (in `quadratic`) as a pointer to a structure is not noted by the compiler, since it does not enforce the consistent use of polymorphic types. Polymorphism is discussed in more detail in Chapter 6.

Exercise 4.8 Rewrite the higher order function `sum` of Section 2.5.1 so that it is non-recursive. Use it to implement a program that calculates the number of different strings with at most n letters that you can make with x different letters. The number of strings is defined by:

$$\begin{aligned} k, x, n &\in I\!N \\ k &= x^1 + x^2 + \ldots + x^n \end{aligned} \tag{4.2}$$

You will have to pass x as an extra argument through `sum` to the function that is being summed.

Exercise 4.9 Exercise 2.13 required you to approximate e given a series. Rewrite it so that it calculates e^x, for a real number x. The function is given by:

$$\begin{aligned} x &\in I\!R \\ n &\in I\!N \\ e^x &= \sum_{i=0}^{n} \frac{x^i}{i!} \\ &= \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \ldots \end{aligned} \tag{4.3}$$

Eliminate all recursion from your solution.

4.6 Summary

The following C constructs have been introduced:

Data type constructors A `struct` can hold a collection of values, which are logically one entity. A `union` can hold one value of a number of different types of values. They are defined as follows:

```
typedef struct                  typedef union
  /*optional name*/ {             /*optional name*/ {
  t1 m1 ; ... tn mn ;             t1 m1 ; ... tn mn ;
} t ;                           } t ;
```

Here t_i and m_i are the types and names of the members. The `struct` can optionally be supplied with a structure-name. (This will be discussed in Chapter 6).

Structure and union members are accessed using the `'.'` operator. The notation x`.`m accesses field m of variable x.

Switch A `switch` statement allows the programmer to choose one alternative out of many. The switch statement has the following syntax:

```
switch( e ) {
  case c1 : S1
  case c2 : S2
  ...
  default : T
}
```

When executing the switch, the expression e is evaluated, if it matches any of c_i, execution resumes at that point in the code, with S_i. If none of the case labels c_i match, the statements following the `default` label, T are executed. The case labels c_i should be unique. To jump to the end of the switch, a `break ;` statement should be used, usually each of S_i will end with a break statement:

`case` c_i`:` S_i `; break ;`

Pointers Pointers are a reference to a value (a memory address). Pointers are used, amongst others, to pass arguments by reference. For any but the basic types this is more efficient than passing arguments by value. Passing arguments by reference allows a function to modify the argument. Such modifications cause the function to have a side effect. The type of a pointer to t is denoted by t`*`. Pointers are dereferenced using the unary `*` operator. A pointer to a variable is obtained using the `&` operator. If p is a pointer to a structure, then p`->`m accesses field m of `*`p

Void type The type `void` is used to indicate that a function does not have an argument or that it does not have a resulting value. The notation `void *` indicates the type of a pointer to anything.

The programming principles of this chapter are:

- Implement algebraic data types using structs and unions with a tag to discriminate between the various alternatives of the union. The tag is essential to distinguish between the various cases of the union.

- Passing large data structures around in C is inefficient in space and time. For good efficiency, the structures should be passed by reference. The disadvantage of passing arguments by reference is that it makes it more difficult to reason about functions, as they might modify the arguments by a side effect.

- The lifetime of data should not be shorter than the lifetime of a pointer to the data. For example, a function should never return a pointers to a local variable. Otherwise the pointer will become a *dangling pointer*, which points to non-existing space.

- Use general higher order functions with partially applied function arguments instead of specialised functions. The mechanism in C supporting partial application is at a lower level of abstraction than the mechanisms available in a functional language. It requires care to use this mechanism, but it is nevertheless a useful programming technique that is indispensable for implementing large scale programs. Examples of its use, for example in the X-windows library, are shown in Chapter 9. Another use of the `void *`, to implement polymorphic types, is also discussed in Chapter 8.

- Do not rely on the C compiler to check consistent use of `void *` type pointers. Use the functional version of the solution to check the types carefully.

- The use of type casts breaks the type system and thus reduces the opportunity that the compiler has to detect errors in a program. Type casts should thus be avoided.

4.7 Further exercises

Exercise 4.10 The C languages offers operators to create and dereference pointers.

(a) Explain the relationship between the C language operators * and &.

(b) What is the output of the following program?

```
int main( void ) {
  int i = 3 ;
  int * p = & i ;
  int * * q = & p ;
  printf( "%d %d %d %d %d %d %d\n",
          3, i, p==*q, *p, q==& p, *q==&i, **q ) ;
  return 0 ;
}
```

(c) State what the output of the following program would be and explain why.

```
int twice( int (*f) (int), int a ) {
  return f( f( a ) ) ;
}

int add( int a ) {
  return a + 1 ;
}

int main( void ) {
  printf("%d\n", twice( add, 2 ) ) ;
  return 0 ;
}
```

(d) Which are the differences between the program below and that of question (c)?

```
int twice( int (*f) (int, int, int),
           int x, int y, int a ) {
  return f( x, y, f( x, y, a ) ) ;
}

int add( int a, int b, int c ) {
  return a + b + c ;
}

int main( void ) {
  printf("%d\n", twice( add, 2, 3, 4 ) ) ;
  return 0 ;
}
```

(e) Which are the differences between the program below and that of question (d):

```
typedef struct {
  int l, r ;
} pair ;

int twice( int (*f) ( void *, int ),
           void * x, int a ) {
 return f( x, f( x, a ) ) ;
}

int add( void * p, int c ) {
  pair * q = p ;
  return q->l + q->r + c ;
}

int main( void ) {
  pair p = {2, 3} ;
  printf("%d\n", twice( add, &p, 4 ) ) ;
  return 0 ;
}
```

Exercise 4.11 Design a program that displays the character set of your computer in a sensible way. Your solution should have a function that classifies a character. Given a character, the function should return a structure that identifies the class of the character and some information within the class. The classes to be distinguished are:

1. A digit. The numerical value of the digit must be returned as well.
2. A lower case letter. In addition the index of the letter (1..26) must be returned.
3. An upper case letter. In addition the index of the letter (1..26) must be returned.
4. White space. Nothing extra needs to be returned.
5. Something else.

The rest of your program should call this function for each printable character and print the character, the numeric equivalent, and the classification for all of them.

Exercise 4.12 An accounting package needs to calculate net salaries given gross salaries, and gross salaries given net salaries.

(a) Previously, floating point numbers were used to represent money. This is not proper, as the figure `1.01` cannot be represented exactly (Section 2.4). Design a data structure that stores an amount of money with a pair of integers, where one integer maintains the whole pounds (dollar, yen, ...), and the other represents the pennies (cent, sen, ...).

(b) Define functions that operate on the money data type. The functions should allow to add to amounts of money, subtract, and multiply a sum with an integer, and divide a sum by an integer. The division should round amounts to the nearest penny/cent/sen.

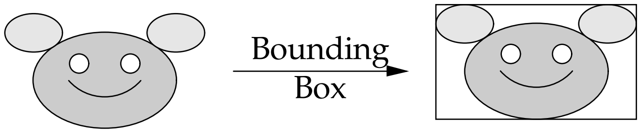
(c) Define a function that calculates the net salary given the following formula:

$$
\begin{aligned}
r &= \begin{cases} 9g/10, & \text{if } g > 15000 \\ g, & \text{otherwise} \end{cases} \\
p &= \begin{cases} r, & \text{if } g > 18500 \\ 9r/10, & \text{otherwise} \end{cases} \\
n &= \begin{cases} p, & \text{if } p < 10000 \\ 4p/5 + 2000, & \text{if } 10000 \leq p < 15000 \\ 3p/5 + 5000, & \text{if } 15000 \leq p < 20000 \\ p/2 + 7000, & \text{otherwise} \end{cases}
\end{aligned}
$$

where g is the gross salary, r is after retirement contributions are taken out, p is after insurance is taken out, and n is the net salary.

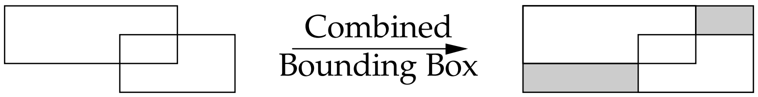
(d) Define a function that determines a gross salary that someone has to be paid in order to get a given net salary. Do not attempt to invert the equations for p and n (this is in real life often not feasible), instead use the function defined above and search for the right amount using `bisection`.

Exercise 4.13 A 'Bounding Box' is a concept used in 2-dimensional graphics. A bounding box of a picture is defined as the rectangular box that is just large enough to contain the whole picture:



Assuming that the edges of the bounding box are parallel to the X and Y axes, a bounding box is fully specified by a pair of coordinates: the coordinates of the lower left hand corner and the upper right hand corner.

Bounding boxes need to be manipulated. In particular, If two elements of a drawing have bounding boxes, the combined bounding box needs to be calculated:



In general, two bounding boxes are combined by defining a new bounding box with the minimum x and y coordinates as the lower left corner and the maximum x and y coordinates as the coordinates of the top right hand corner. Define and implement the following:

(a) A datatype to hold the bounding box.

(b) A function that normalises a bounding box (that is: it ensures that it stores the lower left and upper right corners, and not, for example, the lower right and upper left corners).

(c) A function that combines two bounding boxes into one bounding box.

(d) A datatype to define a line.

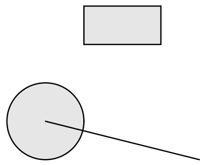
(e) A datatype to define a circle.

(f) A datatype to define a rectangle.

(g) A datatype called `element` that can hold any of the line, circle, or rectangle.

(h) A function that takes a value of the type `element` and that produces the bounding box.

(i) A main program that calculates the combined bounding box of the following components:



A circle at $(2, 2)$ with radius 1, a rectangle with the corners at $(3, 5)$ and $(5, 4)$, and a line from $(2, 2)$ to $(6, 1)$.

Assume in all of the above that the X and Y coordinates are stored in integers.

Exercise 4.14 Define a `struct` to represent some of the following 2D objects: circle, square, box, parallelogram, triangle, and ellipse. Then write a function that when given a 2D object computes the area of the object.

Exercise 4.15 Using the `struct` and function of the previous exercise, define another `struct` that makes it possible to represent 3D objects, such as a sphere, cube, cone, cylinder and pyramid. Write a function to compute the volume of the 3D objects.

Chapter 5

Arrays

The C `struct` of the previous chapter provides the means to collect a *small, fixed* number of data items in a single entity. The items may be of different types. The main topic of this chapter is the *array* data structure. An array gathers an *arbitrary* number of elements into a single entity. The elements of an array must all be of the same type. Sample applications of arrays are vectors of numbers and databases of records.

There are other data structures designed to store a sequence of data items of identical type, the *list* for example. Although the purpose of lists and arrays is the same, the data structures have different performance characteristics. For this reason, some problems must be solved by using lists (arrays would be inefficient), while for other problems the programmer should choose to use arrays (because lists would be inefficient).

Although lists and arrays can be expressed both in imperative and in functional languages, the 'natural' data structure for both paradigms is different. Handling arrays in a functional language can be inefficient, since a purely functional semantics does not allow elements of the array to be updated without creating a new version of the whole array. (We ignore the non-declarative extensions of SML and sophisticated compiler optimisations designed to detect single threaded use of arrays). Handling lists in C can be cumbersome, as memory has to be managed explicitly. For these reasons, problems that do not have a specific preference for either lists or arrays are generally solved with lists in a functional language, but implemented in C using arrays. In the next chapter we compare the efficiency of list and array representations.

This chapter is the first of three that discuss the implementation of sequences of data. Therefore the present chapter starts with a model to explain how one can reason about sequences. After that, the representation of arrays in C is explained. Arrays in C are of a low level of abstraction, that is the programmer has to be concerned with all details of managing arrays. Arrays in C can be used to build powerful abstractions, that hide some of the management details.

While discussing arrays, we present the concept of *dynamic* memory. This is needed to implement dynamic arrays and also to implement lists in C, as shown in Chapter 6.

5.1 Sequences as a model of linear data structures

From an abstract point of view, an array is a *sequence over a set of values*. A sequence is defined as a function from natural numbers to a set of values. A familiar example is a string: it is a sequence of characters. A string can be interpreted as a function by explicitly listing all possible argument/result pairs of the function. We write such pairs using the 'maplet' symbol $\mapsto$. Here are the strings "cucumber" and "sandwich" represented as functions c and s respectively.

$$\begin{aligned}
c, s &: \mathbb{N} \to A \\
c &= \{0 \mapsto \ c \ , 1 \mapsto \ u \ , 2 \mapsto \ c \ , 3 \mapsto \ u \ , \\
&\quad\ 4 \mapsto \ m \ , 5 \mapsto \ b \ , 6 \mapsto \ e \ , 7 \mapsto \ r \ \} && (5.1) \\
s &= \{0 \mapsto \ s \ , 1 \mapsto \ a \ , 2 \mapsto \ n \ , 3 \mapsto \ d \ , \\
&\quad\ 4 \mapsto \ w \ , 5 \mapsto \ i \ , 6 \mapsto \ c \ , 7 \mapsto \ h \ \} && (5.2)
\end{aligned}$$

Note that these sequences start with the index 0: they are *zero-based*. Furthermore, each sequence numbers the elements consecutively. Indeed, sequences can be generalised to start at arbitrary numbers, to leave 'gaps' in the domain, or to use any other discrete domain. Such more advanced data structures include *associative arrays* and *association lists*. They are beyond the scope of this book; the interested reader should consult a book on algorithms and data structures, such as Sedgewick [12].

The advantage of the interpretation of an array as a sequence is that many useful operations on sequences can be expressed using only elementary set theory. We will discuss the five most important sequence operations in the following sections. The first three operations, taking the length of a sequence and accessing and updating an element of a sequence, are used immediately when we define arrays. The next two operations, concatenating two sequences and extracting a subsequence from a sequence, are used when discussing lists in Chapter 6.

5.1.1 The length of a sequence

The length of a sequence is the cardinality of the set of argument/result pairs. We will denote the cardinality of a set s as $\#s$. For the set s defined by (5.2), we have $\#s = 8$.

5.1.2 Accessing an element of a sequence

To access an element of a sequence, we can apply the sequence (which is a function!) to a natural number. The i-th element of the sequence c is $c(i)$; for example, the last character of the string (5.1) is $c(7) = \ r \ $. The function to access an element will almost always be partial. This means that an undefined value is obtained if we try to access an element of a sequence that is outside its domain, for example $c(8) = \perp$ (the symbol $\perp$ means 'undefined'). The notation with parentheses to access an element of a sequence is actually used to access arrays in Fortran; in C, one has to use square brackets to access the element of an array.

5.1.3 Updating an element of a sequence

The third operation that we need is updating a sequence. When given a sequence s and an index $k \in \text{domain}(s)$, then the sequence $s(k \mapsto v)$ is the same as s except at the index position k where it has the value v:

$$\begin{array}{rcl} \cdot(\cdot \mapsto \cdot) & : & (I\!N \to \alpha \times I\!N \times \alpha) \to (I\!N \to \alpha) \\ s(k \mapsto v) & = & \{i \mapsto s(i) \mid i \in \text{domain}(s) \wedge i \neq k\} \cup \{k \mapsto v\} \end{array} \tag{5.3}$$

For example, the sequence s from (5.2) can be updated as follows:

$$\begin{array}{rcl} s(0 \mapsto \ S\ ) & = & \{0 \mapsto \ S\ , 1 \mapsto \ a\ , 2 \mapsto \ n\ , 3 \mapsto \ d\ , \\ & & 4 \mapsto \ w\ , 5 \mapsto \ i\ , 6 \mapsto \ c\ , 7 \mapsto \ h\ \} \end{array}$$

5.1.4 The concatenation of two sequences

To concatenate two sequences, the set union operator seems appropriate, but with a twist. If we try to concatenate the two strings c and s above by naively writing $c \cup s$, the result would be a relation but not a function:

$$\begin{array}{rcl} c \cup s & = & \{0 \mapsto \ c\ , 0 \mapsto \ s\ , 1 \mapsto \ u\ , 1 \mapsto \ a\ , \\ & & 2 \mapsto \ c\ , 2 \mapsto \ n\ , 3 \mapsto \ u\ , 3 \mapsto \ d\ , \\ & & 4 \mapsto \ m\ , 4 \mapsto \ w\ , 5 \mapsto \ b\ , 5 \mapsto \ i\ , \\ & & 6 \mapsto \ e\ , 6 \mapsto \ c\ , 7 \mapsto \ r\ , 7 \mapsto \ h\ \} \end{array}$$

When two sequences are concatenated, we must decide explicitly which one will be the first part of the resulting sequence and which one will the last part. The domain of the sequence that will be the last part must be changed. To do this, we define the operator $\frown$ to denote string concatenation as follows:

$$\begin{array}{rcl} \cdot \frown \cdot & : & (I\!N \to \alpha \times I\!N \to \alpha) \to (I\!N \to \alpha) \\ s \frown t & = & s \cup \{(i + \#s) \mapsto t(i) \mid i \in \text{domain}(t)\} \end{array} \tag{5.4}$$

Applying the concatenation operator to the two strings of (5.2) and (5.1) yields a proper function:

$$\begin{array}{rcl} c \frown s & = & \{0 \mapsto \ c\ , 1 \mapsto \ u\ , 2 \mapsto \ c\ , 3 \mapsto \ u\ , \\ & & 4 \mapsto \ m\ , 5 \mapsto \ b\ , 6 \mapsto \ e\ , 7 \mapsto \ r\ , \\ & & 8 \mapsto \ s\ , 9 \mapsto \ a\ , 10 \mapsto \ n\ , 11 \mapsto \ d\ , \\ & & 12 \mapsto \ w\ , 13 \mapsto \ i\ , 14 \mapsto \ c\ , 15 \mapsto \ h\ \} \end{array}$$

The definition of the concatenation operator looks a bit complicated because concatenation *is* actually a complicated operation. When one array is concatenated with another in a program, at least one of them must be copied to empty cells at the end of the other. (If there are no empty cells, then both of them need to be copied!) This notion of copying is captured by the change of domain for the second sequence.

5.1.5 The subsequence

With the concatenation operator we can build larger sequences from smaller ones. It is also useful to be able to recover a subsequence from a larger sequence. When given that $l, u \in \text{domain}(s)$, then a subsequence $s(l \ldots u)$ of a sequence s can be defined as follows:

$$\begin{aligned} \cdot(\cdot \ldots \cdot) &: (\mathbb{N} \to \alpha \times \mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \to \alpha) \\ s(l \ldots u) &= \{(i-l) \mapsto s(i) \mid i \in \text{domain}(s) \wedge l \leq i \leq u\} \end{aligned} \quad (5.5)$$

The lower bound of the subsequence is l, and the upper bound is u. The domain of the resulting subsequence is changed, so that the index of the left most element of every sequence is always 0.

As an example, we will take the subsequence corresponding to the string "and" from the string "sandwich":

$$s(1 \ldots 3) = \{0 \mapsto \quad a \quad, 1 \mapsto \quad n \quad, 2 \mapsto \quad d \quad\}$$

We now have at our disposal a notion of sequences based on elementary set theory. Sequences will be used in this chapter to specify a number of operations on arrays. In the next chapter, we shall look at alternative implementations of sequences based on lists.

5.2 Sequences as arrays in SML

Arrays are not part of the SML language, but the SML/NJ implementation offers the type `array` as part of the standard library. We will be using the SML/NJ array to represent a sequence. Several functions are defined on the type `array`; we will briefly review the most important ones here. Like our sequences, arrays in SML/NJ are zero-based. The index of the first element is always 0, and the index of the last element is always `n-1` where `n` is the length of the array.

5.2.1 Creating an SML array

An array is created by one of two functions, `array` or `tabulate`. The first is the least powerful of the two: `array(n,v)` creates an array of length `n` such that all array elements have the initial value `v`. This corresponds to the following sequence:

$$\begin{aligned} \texttt{array(n,v)} = \; &\{0 \mapsto v, \\ &1 \mapsto v, \\ &\vdots \\ &(n-1) \mapsto v\} \end{aligned}$$

The type of the SML `array` function is:

```
(* array : int *  a ->  a array *)
```

The second SML function that can be used to create an array is more general: `tabulate(n,f)` creates an array of length `n` corresponding to the sequence:

$$
\begin{aligned}
\texttt{tabulate(n,f)} \;=\;& \{0 \mapsto f(0), \\
& 1 \mapsto f(1), \\
& \vdots \\
& (n-1) \mapsto f(n-1)\}
\end{aligned}
$$

The type of `tabulate` is:

```
(* tabulate : int * (int-> a) ->  a array *)
```

The function `tabulate` is more versatile and will be used in many cases. The `array` function is *only* used for initialising an array that will be updated later.

5.2.2 The length of an SML array

The number of elements of an array `s` is given by the function `length(s)`. This corresponds directly with the operation # on a sequence. The type of `length` is:

```
(* length :  a array -> int *)
```

In SML/NJ the name `length` is used for both lists and arrays, but `length` is not overloaded. When using arrays and lists in the same SML module one should be careful to indicate whether `Array.length` or `List.length` is needed.

5.2.3 Accessing an element of an SML array

The function `sub(s,i)` accesses the `i`-th element of an array `s`. This corresponds directly to accessing an element of a sequence using the operation $s(i)$. Trying to access an element that is not within the domain of the array gives an error. The type of `sub` is:

```
(* sub :  a array * int ->  a *)
```

5.2.4 Updating an element of an SML array

The final operation that we need is an array update. This `upd` operation can be defined if we follow closely the specification of the sequence update $s(k \mapsto v)$:

```
(* upd :  a array * int *  a ->  a array *)
fun upd(s,k,v) = let
                    val n   = length(s)
                    fun f i = if i = k
                                 then v
                                 else sub(s,i)
                 in
                    tabulate(n,f)
                 end ;
```

The function upd copies all elements of the old array s to a new array, except at index position i, where the new value v is used. The old array s is not changed.

5.2.5 Destructive updates in SML

SML provides a built-in function update, which has a completely different behaviour from that of our function upd. The type of update is:

```
(* update :  a array * int *  a -> unit *)
```

The result type of the function is unit, not a array. This implies that the function can only do useful work by updating one of its arguments as a side effect. In this particular case it changes the contents of the array *destructively*. SML provides the side effecting function update purely for efficiency reasons. It is costly to create a new array just to change one element. If the programmer knows that the old array is not going to be needed again, then no harm is done by reusing the cells occupied by the old array and changing its value. However, the function update can be used only when the programmer is sure that the old contents of the array are not going to be used again.

In general, it is difficult to know when it is safe to use update, so we discourage its use. In this book, we do not make use of update, so as not to blur the division between the algorithmic aspects of programming, for which we use SML, and the problem of creating efficient programs, for which we use C.

This completes the representation of sequences in SML. The implementation of the two remaining useful functions corresponding to concatenation and subsequencing are left as an exercise.

Exercise 5.1 Give an SML function concatenate(s,t) to concatenate two arrays s and t.

Exercise 5.2 Define an SML function slice(s,l,u) to return the data of the array s from index l to index u as a new array. This corresponds to taking a subsequence.

5.3 Sequences as arrays in C

Arrays form a proper part of the C language. In this section we give a brief general overview of the arrays. We will elaborate each concept in considerable detail in subsequent sections.

5.3.1 Declaring a C array

In C an uninitialised array is declared as follows:

t a[n] ;

Here t is the type of the elements of the array, a is the name of the array and n is a compile time constant (see below) giving the number of elements of the array. An initialised array declaration has the general form:

t a[n] = { v_0, v_1, ... } ;

The initial values of the array elements $v_0 \ldots v_{n-1}$ must all be of type t. The upperbound n may be omitted from an initialised array declaration, in which case the number of initial values determines the size of the array:

t a[] = { v_0, v_1, ... } ;

It is possible for the explicit number of elements and number of initial values to disagree. We will say more about this later.

If an array is passed as an argument to a function the type of the array argument is denoted as follows:

t a[]

Here t is the type of the elements of the array a. C does not provide a facility to discover the number of elements of an array argument. We will discuss this extensively below.

5.3.2 Accessing an element of a C array

An array can be subscripted using the expression.

a[e]

Here a is an array and e is an integer valued expression. As in our sequences and in SML, the lower bound of an array in C is always 0. Thus the first element of a is the element with index 0. The upper bound of the array is the number of elements in the array minus 1. For example, if an array has n elements, the array bounds are 0 and $n-1$.

5.4 Basic array operations : Arithmetic mean

We will now discuss the arrays of C in more detail. Consider the problem of calculating the arithmetic mean of a sequence of numbers. The mean $\overline{s}$ of a sequence of numbers s is defined as:

$$\begin{aligned} s &: \mathbb{N} \rightarrow \mathbb{R} \\ \overline{s} &: \mathbb{R} \\ \overline{s} &= \frac{\sum_{i=0}^{n-1} s(i)}{n} \qquad \text{where} \qquad n = \#s \end{aligned} \tag{5.6}$$

All numbers of the sequence are to be added together, and the sum should be divided by the length of the sequence. Here is the SML algorithm to compute the mean of an array of reals:

```
(* mean : real array -> real *)
fun mean s
    = let
          val n = length s ;
          fun add_element sum i = sum + sub(s,i)
      in
          foldl add_element 0.0 (0 -- n-1) / real(n)
      end ;
```

The function mean computes the length of the array s. It then iterates over the elements of the arithmetic sequence 0 -- n-1 and calls add_element for every index from 0 to n-1. The expression sub(s,i) returns the i-th element of the array s. The value of the array element is then added to the current sum.

Exercise ⋆ 5.3 Prove that SML function mean satisfies the specification given by (5.6).

The SML function mean can be transformed into an efficient C function using the increasing, left folding for schema of Chapter 3. The resulting, nearly complete C implementation uses an increasing for loop:

```
double mean( double s[] ) {
  int n = /*length s*/ ;
  int i ;
  double sum = 0.0 ;
  for( i=0 ; i < n ; i++ ) {
    sum = sum + s[i] ;
  }
  return sum/n ;
}
```

The array argument of the function mean is declared as an array s of doubles:

```
double s[]
```

C does not provide an operation to determine the number of elements of an array, hence the qualification 'nearly complete' for the first C version of mean. The choice of not providing for a length operation was made for efficiency reasons: not having to maintain the length of an array saves space and time. However, in the present case, we need the length of the array. The conventional solution to this problem is to explicitly maintain the number of elements. In the case of the function mean, an extra argument n is passed specifying the number of elements of the array. The definitive version of mean becomes:

```
double mean( double s[], int n ) {
  int i ;
  double sum = 0.0 ;
  for( i=0 ; i < n ; i++ ) {
    sum = sum + s[i] ;
  }
  return sum/n ;
}
```

Working with the *number of elements* of an array as opposed to the *upper bound* is usual in C. It is important to see the difference, as the upper bound is one less than the length. This is the reason that the for loop in the body of main tests on `i < n`:

```
for( i=0 ; i < n ; i++ ) {
```

The for loop operates over the domain $0 \leq i < n$. Indeed almost any for loop in C that traverses an array has the following form:

```
for( i = 0 ; i < n ; i++ )
```

This is opposed to the form:

```
for( i = 1 ; i <= n  ; i++ )
```

Confusing the number of elements with the upper bound will lead to an 'off by one error', accessing one element too many or too few. This is probably the most common error amongst programmers fluent in other languages but not used to the particular way C handles arrays.

To access an element of the array, the notation `s[i]` is used: `s` is the name of the array and `i` is an expression which is used to index the array. The C expression `s[i]` is equivalent to the SML expression `sub(s,i)`. In SML and many other languages the indexing operator requires the index to be within the bounds of the array that is being indexed. The C language does not require bounds to be checked. Therefore, indexing an array with a large or negative index, for example `s[-100]`, might not result in a run time error, but will probably result in some random value being returned.

When an array is declared, the number of elements in the array must be specified, as is shown in the `main` function used to test the function `mean`:

```
int main( void ) {
  /* constant 4 used in the next line */
  double data[4] = { 55.0, 90.0, 83.0, 74.0 } ;
  /* same constant 4 used in the next line */
  printf( "%f\n", mean( data, 4 ) ) ;
  return 0 ;
}
```

In this program, `data` is an array that can store four doubles. The size of an array must be a *compile time* constant. This means that the compiler must be able to calculate precisely how long the array is. Any positive integer constant or an expression using only constants is legal. When an array is declared, it may be initialised, using the curly brackets as shown in the second line of the function `main`.

```
double data[ 4 ] = { 55.0, 90.0, 83.0, 74.0 } ;
```

The first value of the list, `55.0`, will be assigned to the element with index `0`, the next one to the element with index `1`, and so on, until the last value (`74.0`) is assigned to the element with index `3`. If the number of elements in the list of initialisers exceeds the capacity of the array, a syntax error will result. If fewer elements are specified in the initialiser list, the remaining elements of the array will be initialised to zero.

The main program for mean above uses the same constant 4 twice: once in the declaration of the array and once when calling mean. It is tempting to write:

```
int main( void ) {
  const int array_length = 4 ;
  double data[array_length] = { 55.0, 90.0, 83.0, 74.0 } ;
  printf( "%f\n", mean( data, array_length ) ) ;
  return 0 ;
}
```

Unfortunately, this is invalid C, as array_length is *not* a compile time constant, despite the fact that it is obvious that it will always be 4. In general, the keyword const means that the compiler marks the variables as read-only (that is, they cannot be changed by the assignment statement) but it does not accept them as constants. For proper constants, we have to use the C preprocessor, which offers a *macro facility*. A macro is a means of giving a name to an arbitrary piece of text, such that wherever the name appears, the piece of text is inserted. The macro definition mechanism uses the keyword #define as follows:

```
#define array_length 4

int main( void ) {
  double data[array_length] = { 55.0, 90.0, 83.0, 74.0 } ;
  printf( "%f\n", mean( data, array_length ) ) ;
  return 0 ;
}
```

The first line declares that array_length is *syntactically equivalent* to 4. The text array_length anywhere after this declaration will be replaced by the string 4. This syntactic replacement can cause some unexpected behaviour as will be explained in Chapter 8.

5.5 Strings

In the second chapter, we introduced string constants. We postponed discussing the type of this string constant. Now that arrays have been introduced, the type of strings and their usage can be discussed in greater detail.

In C, a string is stored as an array of characters. A string constant is denoted by a series of characters enclosed between double quotes ". If there are n characters between the quotes, the compiler will store the string in an array of $n + 1$ characters. The first n characters hold the characters of the string, while the element with index n (the $n+1$-th element, as the first element has index 0) holds a special character, the NULL-character. The NULL-character, denoted as \0 , signals the end of a string.

The motivation for ending a string with a NULL-character is that any function can now determine the length of a string, even though C does not provide a mechanism to determine the size of an array. By searching for the NULL-character, the end of the string can be found. Note that a string cannot contain a NULL-character and that an array storing a string must always have one extra cell to accommodate the \0 . Consider the definition of a function that determines the length of a string, `strlen`. This function is defined as follows:

$$\begin{aligned} \text{strlen} &: (\mathbb{N} \rightarrow A) \rightarrow \mathbb{N} \\ \text{strlen}(s) &= \#s \end{aligned}$$

In SML, the definition of `strlen` just uses the primitive function `size`:

```
(* strlen : string -> int *)
fun strlen a = size a ;
```

In C, we will have to search for the NULL character as follows:

```
int strlen( char string[] ) {
  int i=0 ;
  while( string[i] !=  \0  ) {
    i++ ;
  }
  return i ;
}
```

Each character of the string is compared with the NULL-character. When the NULL character is found, the index of that character equals the length of the string (as defined in the beginning of this section). This function assumes that there will be a NULL character in the string. If there is no NULL character, the function will access elements outside the array boundaries, which will lead to an undefined result.

5.5.1 Comparing strings

The function `strlen` above is one of a dozen string processing functions that are predefined in the standard C library. The most important of these functions are discussed below, as they allow programmers to handle strings without reinventing string processing functions over and over again. To use these functions the following include directive must be present in the program.

```
#include <string.h>
```

The first two functions to be discussed are functions that compare strings. In many modern languages the relational operators, ==, >=, and so on, can be used to compare arbitrary data. In C, the relational operators work on characters, integers, and floating point numbers, but they do not operate structured data, and hence, not on strings. Instead, the programmer must call a function `strcmp`. This function has two strings as arguments and returns an integer: zero if the strings are equal, a negative number if the first string is 'less than' the second string, and a positive number if the first string is 'greater than' the second. Less than and greater than use the underlying representation of the character set (as integers) to order strings. The following inequalities are all true:

```
strcmp( "monkey", "donkey" )     > 0,
strcmp( "multiple", "multiply" ) < 0,
strcmp( "multi", "multiple" )    < 0,
strcmp( "51", "3" )              > 0,
strcmp( "51", "312" )            > 0
```

A variant of `strcmp`, `strncmp` compares at most n characters. If no difference is found in the first n characters, the function `strncmp` returns `0`, indicating that the (first n characters of the) strings are equal. The number of characters to compare is passed as the third argument of `strncmp`. Here are some expressions that use `strncmp`:

```
strncmp( "multiple", "multiply", 8 ) < 0,
strncmp( "multiple", "multiply", 7 ) == 0
```

5.5.2 Returning strings; more properties of arrays

The functions `strlen`, `strcmp`, and `strncmp` all deliver an integer as the result value (returning the length of the string or the result of the equality test). More complicated operations, like string concatenation, require the returning of a string as a result. In C, it is not possible for a function to return an array as a result. Arrays in C are ‘second class citizens’. In a functional language, an array is a first class citizen in that it can be manipulated in exactly the same way as any other data type. In C, arrays cannot be assigned and an array cannot be returned as function result. Furthermore, when an array is passed as an argument to a function, a special parameter passing mechanism is used. Instead of passing an array by value, as is done for basic data types, an array is passed by reference.

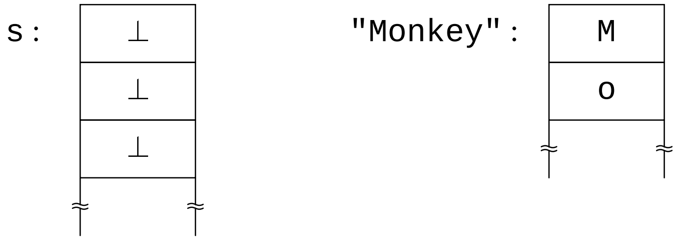
Because of these restrictions on arrays, functions that need to return a string (or any other array) must do some in a special way. Below, we will introduce a first solution. Section 5.7 shows a more elegant method that requires the use of dynamic memory. As an example program, we will discuss the function `strcpy` that copies one array to another array. This is an operation which is not necessary in SML, but in C it is a necessary operation (as strings cannot be assigned). Because the copying function cannot return a string, the array where the result appears is passed as an argument to the function. The C program below defines and uses the function `strcpy`. This function is part of the standard library, so it is given here by way of example only:

```
void strcpy( char output[], char input[] ) {
  int i = 0 ;
  while( input[i] !=  \0  ) {
    output[i] = input[i] ;
    i = i + 1 ;
  }
  output[i] =  \0  ;
}

int main( void ) {
  char s[ 10 ] ;
  strcpy( s, "Monkey" ) ;
```

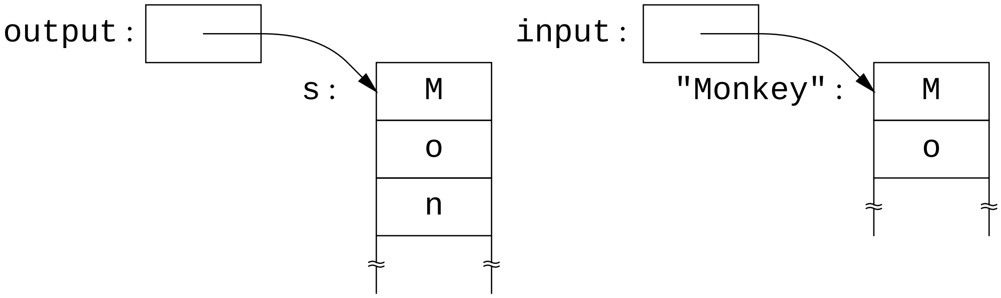
```
    printf( "The string  %s \n", s ) ;
    return 0 ;
  }
```

The reason why `strcpy` works is that the array `output`, the destination of the copy operation, is not passed by value, but *by reference*, following the model presented in Section 4.4.3. This means that the array is not copied, and only one version of the array is present. Any changes made to the array `s` will be visible to the calling function `main`. The function `printf` has a format to print strings, the `%s`-format.

An array is passed by reference because an array in C is not a collection of values, but only a *pointer to* the collection of values. It is fundamental to understand the close relationship between arrays and pointers. The execution of the main program above will allocate an array `s`, with space for 10 characters (indexed from 0 to 9); `s` is a *constant* pointer to the first cell (character) of the array. Just prior to calling `strcpy`, the contents of all cells of `s` is undefined. The picture below shows both arguments of `strcpy`; These are `s` and the string `"Monkey"`:



The array `s` is not the collection of values $(\perp, \perp, \perp, \ldots)$, but the reference to the first element. Likewise, `"Monkey"` is not the collection of values (M , o , n , k , e , y , \0), but a pointer to the first of these values, M . When calling `strcpy`, these pointers are passed to the function `strcpy`. When the function `strcpy` is about to return, the array `s` will have been updated to contain the following values:



The array itself (the *pointer*) is not changed, it is still the same reference to the first cell of the array. When returning from the function `strcpy`, the (pointer to the) array `s` is passed to `printf` to print the copied string. Because the array is passed as a pointer, an array parameter may also be declared as t`*`. Here t is the type of the elements of the array. The first argument of `strcpy` can be declared `char *output`, the second as `char *input`, this is the notation used in the C-manual.

A disadvantage of passing the output array as an extra argument is that this array might not be long enough to hold the copied string. If the array s declared in main provided space for only three characters, the first three character M , o and n would end up in s, but the rest of the string would end up somewhere else (remember that the bound check is not performed). The results are unpredictable. That this can be dangerous was demonstrated by the infamous 'Internet-worm' [13], a program that invaded and crashed thousands of UNIX machines worldwide by abusing, amongst others, a missing bound check in a standard UNIX program.

To prevent strcpy from writing beyond the array upper bound, the length of the array must be passed explicitly as a argument. The function strncpy has a third argument to pass the length. It will never overwrite more elements. The function strncpy is generally preferred over strcpy. In cases when the programmer can prove that the array is large enough, strcpy can be used instead.

The functions strcat and strncat concatenate two strings. The concatenation is different from functional concatenation as it destroys one argument on the way. The first argument of strcat is an array containing the initial string, the second argument contains the string to be concatenated, and the concatenation is performed in the first argument. The first argument serves both as the source and as the destination of the function.

The function strncat is the safe counterpart of strcat. It has a third argument specifying how many elements to concatenate at most. All the str... functions discussed here are part of the string library. They are discussed in more detail in Appendix C.

5.5.3 An application of arrays and strings: argc and argv

Most of the programs that we have seen so far are written for one specific purpose. They compute the answer to just the one problem that they are intended to solve. The programs are not capable of finding solutions to even slightly different problems. This is too restricted. It is possible to make programs a bit more flexible by allowing the user of the program to pass some information to it. In SML, one would type the name of the main function and give various different arguments to the function. In C, we can use something similar, but a little more effort is required to make it work. The main function of a C program has hitherto been declared as having the following type:

```
int main( void )
```

This type states that no information (void) is passed to the main function and that an integer value should be returned. C permits us to declare the type of main as follows:

```
int main( int argc, char * argv[] )
```

This states that the function main takes two arguments:

argc is an integer that tells how many strings are passed to the program. The value of argc is at least 1.

`argv` is an array of strings. The first string (with index 0) represents the name of the program, for example `a.out`. The other strings are intended as proper arguments to the program. In total there are `argc` $-$ 1 such arguments. If `argc` equals 1, then no proper arguments are provided. The name of the program, `argv[0]`, is always present.

Here is how `argc` and `argv` can be used. The following C program echoes its arguments, with a space preceding each proper argument:

```
#include <stdlib.h>

int main( int argc, char * argv[] ) {
  int i ;
  printf("program %s", argv[0] ) ;
  for( i = 1; i < argc; i++ ) {
    printf(" %s", argv[i] ) ;
  }
  printf(".\n") ;
  return 0 ;
}
```

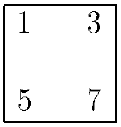
After compiling this program to an executable file, say `a.out`, we could execute it as follows:

```
a.out one two three !
```

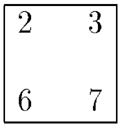
The program will then print the following line of text:

```
program a.out one two three !.
```

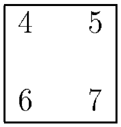
Exercise 5.4 Write a program that simulates the following game of cards. Player A holds three cards on which four numbers are printed as follows:



card 1



card 2



card 3

First, player B chooses a secret number between 0 and 7 (inclusive). Then player A asks player B whether the secret number is on card 1 and notes the answer (yes or no). Then player A asks whether the secret number is on card 2 and also notes the answer. The same happens with card 3. From the three answers, player A is able to work out what the secret number of Player B was.

5.6 Manipulating array bounds, bound checks

In the previous section it has been shown that three issues regarding the bounds of arrays in C are different from arrays in other languages:

1. There is no support to determine the number of elements of an array. So far we have seen two ways to solve this problem: by passing the length as an argument or by declaring some value in the array 'special', for example `\0` .

2. Bound checks are not enforced by the language. Functions must test explicitly on the bounds of arrays.

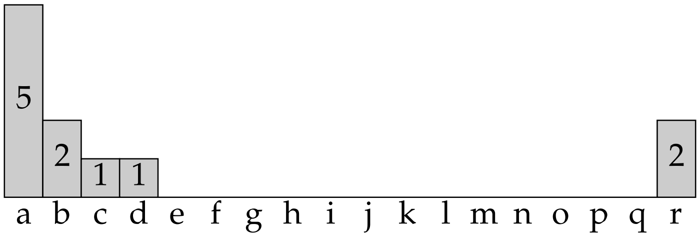
3. The number of elements of any particular array is fixed at compile time.

To this list we could add a fourth issue:

4. The lower bound of an array is always 0 in C.

This is also the case in SML, and even in our sequences. There are however other programming languages in which the lower bound of an array can be chosen freely; Pascal is an example. The choice to give only rudimentary support for arrays in C was made to improve execution speed. Programming convenience has been sacrificed for sometimes significant improvements in execution speed. This section shows how to use arrays with a lower bound that is not fixed to zero, and how to insert the proper bound checks. Section 5.7 will show a more flexible way to declare arrays, overcoming the limitation that the size of an array must be known at compile time.

As an example of using flexible array bounds we will implement a program that makes a histogram. A histogram is a bar graph that shows how often each value occurs in a sequence of values. Given a string, a histogram can be used to illustrate how often each character occurred in that string. The figure below shows the histogram of the string "abracadabra".



Making a histogram can be expressed conveniently in terms of sequences. A histogram of a sequence s is the sequence defined as follows:

$$\begin{array}{rcl} \text{histogram} & : & (I\!N \rightarrow A) \rightarrow (I\!N \rightarrow I\!N) \\ \text{histogram}(s) & = & \{\text{ord}(x) \mapsto n \mid x \in \text{range}(s) \wedge n = \#\{i \mid s(i) = x\}\} \end{array} \quad (5.7)$$

Here we assume that the function 'ord' maps characters to natural numbers. In SML, we are going to give a function `histogram` that counts characters in a particular range. The bounds of this range will be supplied as arguments. An array of characters (represented in SML as an array of strings) is analysed, and the number of times that each character is found is returned in an array indexed (indirectly) with characters. A useful auxiliary function is `inc` below: it delivers a new array, which is the same as the old one, except that a given array element has been incremented by one:

```
(* inc : int array -> int -> int array *)
fun inc s i = upd(s,i,sub(s,i) + 1) ;
```

The `histogram` function (below) starts by calculating the length `n` of the array `input`. The library function `array` creates a new array `empty` and initialises each element to 0. The size of the new array is one more than the difference between the upper bound `u` and the lower bound `l`. The `tally` function is then folded over the index domain of the array `input`, calling `inc` to tally the occurrences of each character in the input array `input`. The primitive SML function `ord` converts the first character of a string to its integer code.

```
(* histogram : char array -> int -> int -> int array *)
fun histogram input l u
    = let
          val n = length input
          val empty = array(u - l + 1,0)
          fun tally hist i = inc hist (ord(sub(input,i))-l)
      in
          foldl tally empty (0 -- n-1)
      end ;
```

Exercise ⋆ 5.5 Prove that the function `histogram` satisfies the specification of (5.7)

As was shown in the previous section on strings, C does not allow arrays to be returned as function values. Therefore, we will define the function `histogram` in such a way that the array in which the histogram is going to be written is passed as an argument. This argument is passed by reference because it is an array. This is the same method as used for the first argument of `strcpy`.

```
void histogram( char input[], int hist[], int l, int u ) {
  int i ;
  /*C Make an empty histogram*/
  for( i=0 ; input[i] !=  \0  ; i++ ) {
    /*C Incorporate input[ i ] in the histogram*/
  }
}
```

The first argument contains the text for which a histogram is to be made. Since the string is NULL terminated, the size of the array does not have to be passed as an argument. The for loop that iterates over the array stops as soon as the NULL character is recognised. The second argument is the array that will contain the histogram, and the third and the fourth arguments are the bounds of this array. To show how this function is going to be used, consider the following fragment of a main program:

```
#define lower   a
#define upper   z
#define number (upper - lower + 1)

int main( void ) {
  int hstgrm[ number ] ;
  histogram( "abracadabra", hstgrm, lower, upper ) ;
  print_histogram( hstgrm, lower, upper ) ;
  return 0 ;
}
```

The histogram hstgrm is declared in the function main. It is an array of number integers. The constant number is defined as one more than the difference between the encoding upper of the upper bound z and the encoding lower of the lower bound a . Assuming that the letters of the alphabet are encoded consecutively (which is the case for the common ASCII encoding, but not for the less common EBCDIC encoding of characters), number will have the value 26. The array hstgrm consists of 26 elements indexed from 0 to 25. Element 0 will be used to tally the number of a's, 1 to tally the b's, … and element 25 to tally the z's.

The first function to be called from main is histogram. Its first argument is the string "abracadabra" the remaining arguments are the array hstgrm, the lower bound lower, and upper bound upper of the histogram. The first action of histogram should be to initialise the array. This initialisation is performed conveniently by a for loop:

```
for( i=0 ; i<=u-l ; i++ ) {
  hist[i] = 0 ;
}
```

The variable i runs from the lower bound of the array 0 to u-l, which is one less than the number of elements of the array; each element is subsequently initialised to zero. After the initialisation, another for loop should iterate over the input. Each of the input elements of the input array is inspected in turn. According to the for schema of Chapter 3, the body of the for loop should contain the following assignment:

```
hist = /*hist with one added at index input[i]*/ ;
```

The functional version would make a new, updated array and use this new array for the next version. This is not strictly necessary since the previous version of the histogram is no longer needed. In C, the array hist is updated by incrementing the value of the appropriate cell of the array. Given a particular index, of the array element that needs to be incremented, the following statement will increment the appropriate array element:

```
hist[ /*index*/ ] ++ ;
```

To calculate the appropriate value of /*index*/ the character values obtained from the array input must be 'shifted to the left' by the lower bound l so that the letters a to z map onto the indices in the histogram 0 to 25. This shift operation is implemented by means of a subtraction on the characters (as explained in Chapter 2).

```
if( input[i] < l || input[i] > u ) {
  abort() ;
}
hist[ input[i] - l ] ++ ;
```

The if-statement verifies that the character `input[i]` is within the range `l` ... `u`. If it is not in range, the program will be aborted. This range check also catches a possible machine dependency of the type `char`, the possibility that some characters are represented by negative numbers. Any negative character causes the program to be aborted.

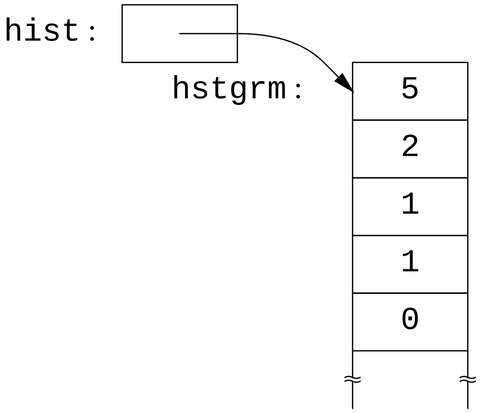
A complete program that creates a histogram and prints the result is given below. Arrays cannot be printed directly in C, instead they must be explicitly formatted. In this example, the histogram bars are formatted using lines of X-es.

```
#include <stdio.h>

void histogram( char input[], int hist[], int l, int u) {
  int i ;
  for( i=0 ; i<=u-l ; i++ ) {
    hist[i] = 0 ;                       /*Exercise 5.6*/
  }
  for( i=0 ; input[i] !=  \0  ; i++ ) {
    if( input[i] < l || input[i] > u ) {/*Exercise 5.6*/
      abort() ;
    }
    hist[ input[i] - l ] ++ ;           /*Exercise 5.6*/
  }
}

void print_xs( int n ) {
  int i ;
  for( i=0 ; i<n ; i++ ) {
    printf( "X" ) ;
  }
}

void print_histogram( int hist[], int l, int u ) {
  int i ;
  for( i=0 ; i<=u-l ; i++ ) {
    printf( "%c: %d ", i+l, hist[i] ) ; /*Exercise 5.6*/
    print_xs( hist[i] ) ;               /*Exercise 5.6*/
    printf( "\n" ) ;
  }
}
```

```
#define lower    a
#define upper    z
#define number  (upper - lower + 1)

int main( void ) {
  int hstgrm[ number ] ;
  histogram( "abracadabra", hstgrm, lower, upper ) ;
  print_histogram( hstgrm, lower, upper ) ;
  return 0 ;
}
```

Exercise 5.6 In the final histogram program above, there are seven places where an array is being indexed (on the five lines marked with the comment `/*Exercise 5.6*/`). Before one of these indexing operations, a bound check is performed. Why are there no bound checks before the other six index operations?

To complete the discussion, let us consider briefly how the program would execute. Just prior to the calls to `histogram`, the array `hstgrm` contains undefined values. As shown previously, a pointer to the array is passed to the function `histogram`, allowing it to update the histogram. This will ultimately lead to a complete histogram. When the function `histogram` is about to return, the array `hstgrm` will have been updated to contain the following values:



5.7 Dynamic memory

Arrays as discussed so far have two serious limitations:

1. The array size has to be fixed statically. In the example of the histogram, it was assumed that the string had only lowercase letters: the bounds were fixed at `a` and `z` .

2. The lifetime of an array that is declared in a function is limited.

These limitations can be overcome by using *dynamic memory*. Dynamic memory means that a block of store can be *allocated* during run time. This block of store can then be used to store data, such as an array or a struct. Once memory is allocated, it can be used as long as the program needs it. When the storage is no longer needed, it must be *deallocated* again. By maintaining a pointer to the allocated memory and the bounds of the array in a data structure, genuine dynamic arrays can be implemented elegantly in C.

5.7.1 The allocation of dynamic arrays

The histogram function had both limitations mentioned above: the array bounds were limited and the array had to be passed as an extra argument to the function. In the functional version, the array was also specified with fixed bounds. At first, we give an array type with arbitrary bounds in SML. The elements of the dynamic array are restricted to type integer.

```
type dynamic_array = int array * int * int ;
```

The type dynamic_array comprises a zero-based array, the desired lower bound, and the upper bound. To avoid confusion, we will use the term zero-based array for the proper, zero-based array. When we refer to a dynamic array, then this includes the dynamic_array struct as well as the zero-based array. The type of a dynamic array in C could be defined as follows:

```
typedef struct {
  int *data ;           /* The zero-based array */
  int  lb ;             /* Lower bound of dynamic array */
  int  ub ;             /* Upper bound */
} dynamic_array ;
```

The dynamic_array data structure holds a pointer to the zero-based array data as well as the bounds of the array in two integers lb and ub. The elements of the zero-based array are all of type int. Thus, the proper zero-based array is not part of the struct dynamic_array; only a pointer to the zero-based array is part of the struct.

To create an initialised dynamic array in SML we write:

```
(* array_alloc : int -> int -> dynamic_array *)
fun array_alloc l u = (array(u-l+1,0),l,u) ;
```

To create a dynamic array in C, the storage for it has to be *allocated*. This allocation operation is performed by calling the C library function calloc, defined in stdlib.h. The arguments passed to calloc are the number of cells needed and the size of each cell. A dynamic array with a given pair of bounds therefore can be allocated using the following function:

```
#include <stdlib.h>

dynamic_array array_alloc( int l, int u ) {
  dynamic_array da ;
  da.lb = l ;
  da.ub = u ;
  da.data = calloc( u - l + 1, sizeof( int ) ) ;
  if( da.data == NULL ) {
    abort() ;
  }
  return da ;
}
```

The lower and upper bounds `l` and `u` are arguments of the function `array_alloc`. The function stores the bounds in the `dynamic_array` structure and calls `calloc` to allocate the store for the zero-based array. To obtain the size of an element, C supports a built-in function `sizeof`, which returns the size in bytes of an element of the specified type. The following expression returns the number of bytes needed to store an integer:

```
sizeof( int )
```

The number of bytes needed to store a `dynamic_array` structure is returned by the expression:

```
sizeof( dynamic_array )
```

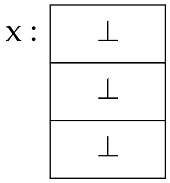
This amount of store would cover one pointer and two integers. The store allocated by `calloc` is guaranteed to be filled with zero-values. The result of `array_alloc` is a proper dynamic array.

The function `calloc` will normally return a pointer to a block of store that can hold the required data. In exceptional situations, when the system does not have sufficient space available, the function `calloc` will return a special pointer known as the `NULL`-pointer. No valid pointer ever has the value `NULL`. By testing if the value of `da.data` equals `NULL`, we can established whether the function `calloc` has succeeded in allocating the memory. The program is aborted if `calloc` has run out of space. Testing the value coming from `calloc` is recommended, as continuing a computation with a pointer that might be `NULL` will give random results: the program might give an arbitrary answer or might crash.

The function `array_alloc` shows that a dynamic array consists of two parts: the zero-based array, which holds the actual data; and the struct of type `dynamic_array`, which holds the bounds and a pointer to the zero-based array. These are separate data structures, which makes it possible to manipulate them separately. There is a danger in this, as one might think that just declaring a variable of type `dynamic_array` creates a proper dynamic array:

```
dynamic_array x ; /* WRONG */
```

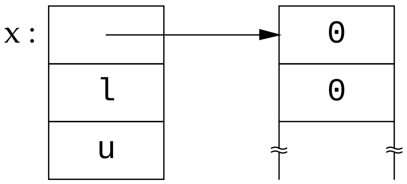
This does not work, as the declaration merely creates the space necessary to hold the pointer and the bounds. They are not properly initialised, nor is the zero based array allocated. In terms of the model of the store, the result of this declaration is:



The two fields storing the upper and lower bound are undefined, and the third field, which should point to a zero-based array, is also undefined. The correct way to use the dynamic array facility is by declaring and initialising the dynamic arrays:

```
dynamic_array x = array_alloc( l, u ) ;
```

The function `array_alloc` will return an initialised dynamic array structure. The first and second field will be initialised to `l` and `u` (representing the desired lower and upper bounds of the array), and the third field is allocated with a zero-based array of $u - l + 1$ elements. Here is a picture showing the results:



5.7.2 The extension of dynamic arrays

In the histogram program, a dynamic array must be extended when an element is encountered that is not in the range of the array. Extending arrays is an expensive operation, for in the general case, we may not assume that an array can be extended by claiming more space. The problem is that other dynamically allocated data structures may occupy that extra space already. The only safe way of extending an array is by copying its contents to a new area of store, that does allow for extra room. This problem is not specific to C, any language that offers arrays is faced with this difficulty.

In SML, the extension of a dynamic array is taken care of by the function `extend`, shown below:

```
(* extend : dynamic_array -> int -> int -> dynamic_array *)
fun extend (old,old_lb,old_ub) l u
    = let
         fun copy new i = upd(new,i-l,sub(old,i-old_lb))
      in
         (foldl copy (array(u-l+1,0)) (old_lb -- old_ub),l,u)
      end ;
```

The function `extend` takes an old array, and new lower and upper bounds `l` and `u`. It creates a new array which is large enough for the extended bounds and then folds the function `copy` over the domain of the old array, so as to copy the old array elements.

The C implementation closely follows the pattern of the SML function `extend`:

```
dynamic_array extend( dynamic_array old, int l, int u ) {
  dynamic_array new = array_alloc( l, u ) ;
  int i ;
  for( i=old.lb ; i<=old.ub ; i++ ) {
    new.data[ i - l ] = old.data[ i - old.lb ] ;
  }
  return new ;
}
```

The function extend takes an old array, and new lower and upper bounds l and u. It first allocates a new zero-based array that is large enough to handle the new bounds. The data elements of this new zero-based array are automatically cleared by array_alloc. The data from the old zero-based array are then transferred. Finally the new zero-based array is returned to the caller.

Exercise 5.7 Adapt the SML version of histogram so that it uses a dynamic array.

The type dynamic_array, together with the functions array_alloc and extend, implement an abstract data type for dynamic arrays (more details on implementing data abstractions are given in Section 8.4).

The dynamic array can be used to implement the function histogram properly (See section 5.7.3 for an explanation of the comments Leak!):

```
  dynamic_array histogram( char input[], int n ) {
    dynamic_array hist = array_alloc( input[0], input[0] ) ;
    int i ;
    for( i=0 ; i<n ; i++ ) {
      if( input[i] < hist.lb ) {
        hist = extend( hist, input[i], hist.ub ) ; /*Leak!*/
      } else if( input[i] > hist.ub ) {
        hist = extend( hist, hist.lb, input[i] ) ; /*Leak!*/
      }
      hist.data[ input[i] - hist.lb ] ++ ;
    }
    return hist ;
  }
```

The function histogram first allocates a one element dynamic array. We use the first character of the input as the initial lower bound and upper bound in order to create a singleton array. (It is assumed that there is at least one character in the input).

The for loop over the input array performs two bound checks: if the character is above the current bounds, the dynamic array is extended by stretching the upper bound; and if the character is below the lower bound, the dynamic array is extended to a new lower bound. After that, the character will fall within the bounds, and the zero-based array hist.data is updated. Note the statement:

```
  hist.data[ input[i] - hist.lb ] ++ ;
```

This has the same functionality as the statement (Section 3.3.4):

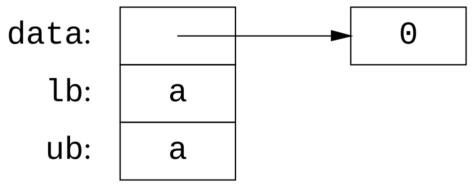
```
  hist.data[ input[i] - hist.lb ] =
                      hist.data[ input[i] - hist.lb ] + 1 ;
```

This demonstrates the advantage of using the ++ operator. The expression specifying which element is to be incremented occurs twice in the second statement, but only once in the first. This means that if the expression needs to be changed for some reason, it would require a consistent change of both copies of the expression in the latter form, while one change suffices for the first form.

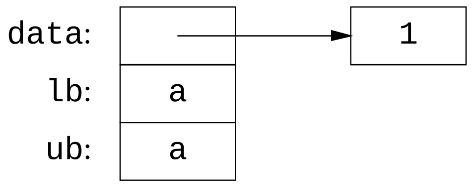
The `main` function using dynamic arrays is shown below.

```
int main( void ) {
  dynamic_array hist = histogram( "abracadabra", 11 ) ;
  int i ;
  for( i = hist.lb ; i <= hist.ub ; i++ ) {
    printf( "%c: %d\n", i, hist.data[ i - hist.lb ] ) ;
  }
  return 0 ;
}
```

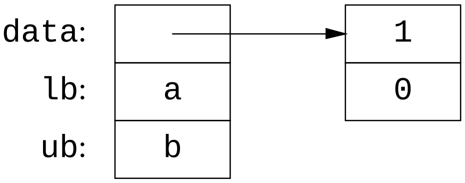
An execution trace shows how the dynamic array is used. Immediately after the function `array_alloc` is called, the dynamic array structure is filled with the following values:



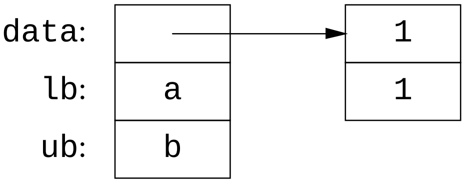
The upper bound and the lower bound are both `a` , and the data is a pointer to a single integer, which stores the value 0. The for loop is entered, and after the first iteration of the for loop the first character of the string has been tallied, which results in the following histogram:



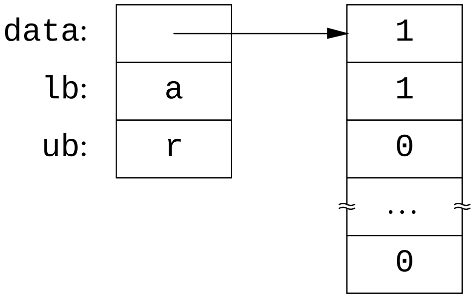
The bounds are unchanged, the element corresponding to `a` in the histogram has been incremented to reflect the single `a` that has been counted. In the next iteration of the for loop, the character `b` is encountered. The value `b` is higher than the upper bound; therefore, the dynamic array has to be extended. Immediately after the call to `extend`, the histogram will look as shown below. The upper bound is now `b` , the lower bound has not changed, and the histogram now consists of two integers. The first one is still 1, the second one has been initialised to 0.



After the second iteration of the for loop, the new array element has been incremented:

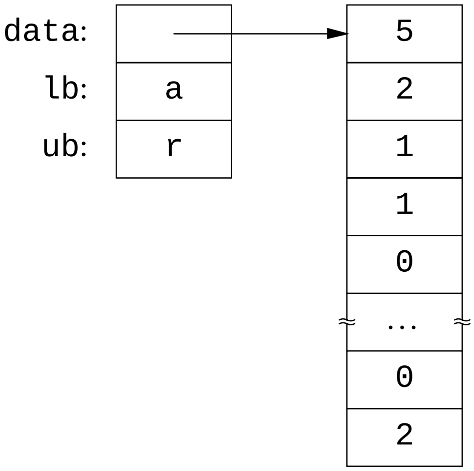


The third character of the input is `r`. This is higher than the upper bound, so the dynamic array has to be extended again. This time, the extension results in the following histogram:



The bounds have been set to `a` and `r`, and the zero-based array now contains 18 integers which store the histogram values for `a` up to `r`. The first two elements of the histogram have been copied from the previous version (they have both value 1). The other 16 cells of the array have been initialised to zeroes.

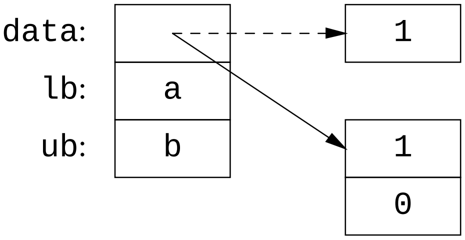
The rest of the input can now be processed, without extending the histogram, because all characters of "abracadabra" are in the range `a` … `r`. The end result of the function `histogram` is the following:



This represents a dynamic array with a lower bound `a`, an upper bound `r`, and a zero-based array of 18 integers with the values $5, 2, 1, 1, 0, \cdots 0, 2$.

5.7.3 The deallocation of dynamic arrays

Reviewing the dynamic array based histogram program of the previous section shows that there is something wrong with the use of the function `extend`. The function takes the `dynamic_array` structure `hist` as an argument and returns a new `dynamic_array` structure. This new structure is then written over the old structure `hist`. The figure below shows the state of the relevant areas of store at the point where `hist` has just been overwritten. The pointer to the old `data` is shown as a dashed pointer; its place has been taken by the pointer to the new data, shown as a solid pointer.



Destroying the only pointer to an area of store causes the area to become inaccessible. In C such an inaccessible area of store will not be reclaimed automatically. All functional languages have a garbage collector, and also some modern imperative languages such as Java, Modula-3 and Eiffel have a garbage collector. This is a component of the language implementation that detects when an area of the store is not longer accessible so that it can be reused. Unfortunately, C and C++ do not have garbage collectors, so the programmer must assume full responsibility for the management of the store.

When inaccessible areas of store are not reclaimed, a program may eventually run out of available space. This situation is known as a *memory leak* because store seems to disappear without warning.

The inaccessible store of the old dynamic array must be *deallocated* explicitly so that it can be reused later. This deallocation can be performed by calling the function `free`, with the pointer to the block of store to be freed as an argument. The function `free` will mark the block for reuse by `calloc` at a later stage. The following call thus deallocates the block of store that was referenced from `hist.data`:

```
free( hist.data ) ;
```

The new inner part of the for loop of the function `histogram` would read:

```
dynamic_array new ;
if( input[i] < hist.lb ) {
  new = extend( hist, input[i], hist.ub ) ;
  free( hist.data ) ;
  hist = new ;
} else if( input[i] > hist.ub ) {
  new = extend( hist, hist.lb, input[i] ) ;
  free( hist.data ) ;
  hist = new ;
}
```

Before the structure hist is overwritten with the new dynamic array new, the memory cells pointed to by hist.data are deallocated, so that the function calloc can reuse them later on.

It is tempting to perform this deallocation as part of the function extend, for example by modifying extend so that it deallocates the old zero-based array just prior to returning the new dynamic array:

```
dynamic_array extend( dynamic_array hist, int l, int u ) {
  dynamic_array new ;
  /*Make the new dynamic array*/
  free( hist.data ) ;
  return new ;
}
```

This implementation is particularly unclean because it is a halfway solution. Reading the function header, it seems that it is a pure function that takes one dynamic array and generates another. However, extend deallocates the old dynamic array, which should therefore not be used afterwards. So extend is not a pure function, because it actually destroys (part of) its input. A better solution is to redesign extend so that it announces that it relies on a side effect. By passing a pointer to the old dynamic array and letting extend update the old dynamic array, the old dynamic array cannot be accessed anymore, because it has been replaced by the new dynamic array. The code with this destructive behaviour is shown below:

```
void extend( dynamic_array * old, int l, int u ) {
  dynamic_array new = array_alloc( l, u ) ;
  int i ;
  for( i=old->lb ; i<=old->ub ; i++ ) {
    new.data[ i - l ] = old->data[ i - old->lb ] ;
  }
  free( old->data ) ;
  *old = new ;
}

dynamic_array histogram( char input[], int n ) {
  dynamic_array hist = array_alloc( input[0], input[0] ) ;
  int i ;
  for( i=0 ; i<n ; i++ ) {
    if( input[i] < hist.lb ) {
      extend( &hist, input[i], hist.ub ) ;
    } else if( input[i] > hist.ub ) {
      extend( &hist, hist.lb, input[i] ) ;
    }
```

```
      hist.data[ input[i] - hist.lb ] ++ ;
    }
    return hist ;
  }
```

The function `extend` takes a pointer to the dynamic array and updates this structure. To access the elements of `hist`, the arrow operator is used, since `hist` is now a pointer to a structure, as opposed to the structure itself. Even though `extend` relies on side effects and destructive updates, the function `histogram` is still a pure function. As will be discussed in more detail in Chapter 8, from a software engineering viewpoint it is important that functions have such a functional interface.

5.7.4 Explicit versus implicit memory management

The functions `calloc` and `free` manage memory *explicitly*. This is in contrast with the *implicit* memory management offered by languages with a garbage collector. When a data structure is used in SML, the store that is needed for the data structure is allocated automatically, that is, without the programmer being aware of it or it being visible to the programmer. When structures are no longer used, a garbage collector automatically reclaims the storage areas that are no longer in use so that they can be reused for storage. The garbage collector of a declarative language is implemented in such a way that only data structures that cannot be reached by the program are reclaimed. This means that *live* data is never reclaimed.

The explicit memory management in C does not have this safeguard. The C programmer decides when a block of store is to be reclaimed by calling the function `free`. If the C programmer makes a mistake and deallocates a block of store that is actually still in use, a dangling pointer is created somewhere in the program. As has been shown before, this will probably lead to random answers or a program crash. To prevent dangling pointers we should make sure that data lives longer than the pointer(s) to the data.

In the case of explicitly allocated memory, the lifetime is completely determined by the programmer. The lifetime of a block of dynamically allocated store ends when `free` is called. Therefore no pointers to that block of store should be alive when the block is deallocated. It is good practice to destroy data in the same function where the pointer to the data is destroyed (provided, that the pointer has not been copied!). This was the case in the last version of the histogram program and in an earlier version of the program where `free` was called from the function `histogram`. In later chapters, we will see that this combined deallocation of the block and destruction of the last reference to it is also the desired practice in other programs.

The problem of memory leaks is also related to the lifetime of pointers and data. In this case the rule is exactly the opposite: memory does leak away if the data has a lifetime that is longer than the pointer pointing to it, for if the pointer is destroyed before the data is deallocated, the data cannot be deallocated anymore. This is a vicious circle. If the pointer is destroyed before the memory is deallocated, we have a memory leak, while if the memory is deallocated before the pointer is destroyed, we have a dangling pointer. This is resolved by first deallocating the memory and destroying the pointer shortly afterwards.

Other languages, like C++, do have support to deallocate memory at the moment that pointers leave the scope. More advanced object oriented languages have built-in garbage collectors that completely relieve the programmer of the burden of managing the memory.

5.7.5 Efficiency aspects of dynamic memory

The reason why C uses explicit memory management is efficiency. From an execution time viewpoint, the allocation and deallocation of memory are costly operations. Allocating a block of memory takes probably one or two orders of magnitude more time than performing a simple operation like comparing two numbers or indexing an array. Because C programmers (are supposed to) know that memory allocation is time consuming, they will do everything possible to avoid unnecessary allocation operations.

In an ideal situation one would know a priori how large an array will have to be. In this case the array can be allocated once with the right size and then used without the need for extension. Information about the size of an array is often not available at the right time (for example when reading the data to be incorporated in the histogram from a file). Dynamic arrays provide a flexible solution, that can be highly efficient if used with care.

The way the dynamic array is used to construct a histogram is not expensive, at least if the input data is long enough. The reasoning behind this is that the number of allocation operations will never exceed the number of *different* characters in the input stream. Suppose that there are 70 different characters (letters, capitals, digits, punctuation marks, and so on); if the input length is a 10000 characters, the 70 allocation operations add negligible overhead.

However if the average length of the input data is about 7 characters, the cost of allocation will far exceed the cost of making the histogram. The real work performed is 7 index operations and increments, while the overhead is up to 7 allocation and deallocation operations.

The functions `calloc` and `free` provide a mechanism to build data abstractions like the dynamic array, but they should be used with care. There is nothing to protect the programmer from using these functions incorrectly (by deallocating memory at the wrong moment) or from using these functions too often. The programmer has the control over the memory management and should not expect any assistance from the compiler.

Using another language, for example C++, does not help to relieve this performance problem. Memory allocation is an expensive operation. Higher level languages do succeed in hiding the details of memory allocation though, as is successfully shown in SML. This does not mean that the penalty is not paid. It only means that the programmer has no longer any control when the penalty is paid, and henceforth does not have to worry about it.

5.8 Slicing arrays: pointer arithmetic

So far, we have considered arrays as single entities which were passed as a whole (albeit by reference). C has a mechanism that allows array slices to be passed. This mechanism is known as *pointer arithmetic*. To illustrate the use of pointer arithmetic, we will write a function that searches for the first occurrence of word in a text. In the specification below, both the word (w) and the text (t) are sequences:

$$\begin{array}{rcl} \text{search} & : & (\mathbb{N} \rightarrow \alpha \times \mathbb{N} \rightarrow \alpha) \rightarrow \mathbb{N} \\ \text{search}(w, t) & = & \text{min}\{\#t_1 \mid t = t_1 \frown w \frown t_2\} \end{array}$$

If the word w does not appear in the text t, the function min is asked to calculate the minimum of an empty set. In this case we shall assume that the number -1 is returned.

To give a search algorithm let us assume that we first try to match the word with the text, starting at the first position of the text. If this match fails, then we move on to the second position and see if we can match the word there. If the word occurs in the text then we will ultimately find it, and return the present starting position. Otherwise, we will exhaust the text, and return -1. Here is a schematic rendering of the first three steps of the search process, where we have chosen a word of length 3:

Step 1:

t_0	t_1	t_2	t_3	t_4	$\ldots$
w_0	w_1	w_2			

Step 2:

t_0	t_1	t_2	t_3	t_4	$\ldots$
	w_0	w_1	w_2		

Step 3:

t_0	t_1	t_2	t_3	t_4	$\ldots$
		w_0	w_1	w_2	

The implementation of the search algorithm in SML takes as arguments the word `w` and the text `t`. Both the word and the text are represented as an array of character. The function `search` also takes the current position `i` as argument. This indicates where a match is to be attempted:

```
(* search : char array -> char array -> int -> int *)
fun search w t i
    = if i > length t - length w
        then ~1
        else if slice (t,i,(i+length w-1)) = w
                then i
                else search w t (i+1)
```

The function `search` is tail recursive. The main structure of the C version of `search` is therefore easily written:

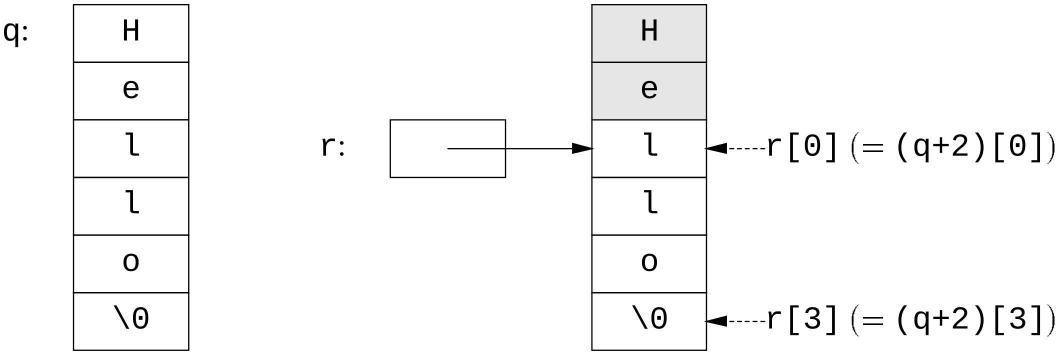
```
int search( char *w, char *t, int i ) {
  while( true ) {
    if( i > strlen( t ) - strlen( w ) ) {
      return -1 ;
    } else if( /*slice of t equals w*/ ) {
      return i ;
    } else {
      i = i + 1 ;
    }
  }
}
```

Apart from some inefficiencies, which we will resolve later, we need to fill in how to take the slice of `t`. The inefficient way to make a slice would be to allocate a block of memory that is large enough to hold the slice, and to copy the relevant cells of the array `t` into this new block of memory. C does however offer an alternative that does not require any cells to be copied. Instead, a different view on the same array is created.

To explain how this works, let us consider a simple example first. Here is a small main program with an array `q` of six characters. The array is initialised with the string `"Hello"`:

```
int hello( void ) {
  char q[6] = "Hello" ;
  char * r = q+2 ;
  return r[0] == q[2] ;
}
```

As explained before, `q` is really a *constant pointer* to the first of the six characters. Now comes the interesting bit: in C it is permitted to add an integer to a pointer, such that the result can be interpreted as a new pointer value. This is shown by the statement `r = q+2`. The figure below shows the array `q`, and it also shows the array `r`, which is merely another view on `q`.



The expression `(q+2)`, which 'adds' `2` to the array `q`, points to `q[2]`. Thus, `(q+2)` is an array where the element with index 0, `(q+2)[0]`, refers to `q[2]`. The element with index 3, `(q+2)[3]`, is the same element as the last element of `q`, `q[5]`.

So the highest index that can be used on the array (q+2) is 3. The elements that are shaded are still there, but they reside below the array. They can be accessed using negative indices. Thus, the expression (q+2)[-2] refers to the lowest element of the array q. The bounds of this array are -2 and 3, and the valid indices for (q+2) are -2, -1, 0, 1, 2, and 3. Any other index is out of bounds and will, because of the absence of bound checks, have undefined results.

The array (q+2) is almost a slice q(2 ... 5) of the array q. There are two differences:

1. (q+2) and q share the values of their cells. This is more efficient; but if either is updated, the other one is updated.

2. (q+2) still refers to the whole array q, the lower and upper parts are not physically detached.

Note that the picture above only suggests that q and q+2 are different arrays, whereas in reality they share the same storage space.

Returning to the word search function, how can we use pointer arithmetic to create a slice of the text and do so efficiently? Firstly, the sharing implied by pointer arithmetic is not a problem because the program that searches for a string does not update either the array or the slice. The second problem from the enumeration above is a real one: the comparison between the slice and the array w must only compare the section of the slice with indexes $0 \ldots \#w - 1$; it should ignore any cells below or above that range. The solution to this problem is to use the function strncmp, which compares only a limited part of an array. Calling strncmp with the start of the slice, and with the number of characters to compare will give the desired result. Consider the following test:

```
strncmp( t+i, w, strlen(w) ) == 0
```

The expression t+i results in a new view on t, shifted by i cells. So, as far as the function strncmp is concerned, the first element to compare is element t[i]. Since strncmp does not look below this index, it does not matter that the rest of the array is still there. Because no more than strlen(w) characters will be compared, the upper part of t will also be ignored.

Before giving the complete code of search, we remove some inefficiencies. Observe that on every iteration of the while loop, the function strlen is called to calculate the lengths of w and t. As these two lengths do not change, it is more efficient to calculate each length only once, and to store the result in a local variable. This leads us to the following complete and efficient implementation of search:

```
int search( char *w, char *t, int i ) {
  const int length_w = strlen( w ) ;
  const int length_t = strlen( t ) ;
  while( true ) {
    if( i > length_t - length_w ) {
      return -1 ;
    } else if( strncmp( t+i, w, length_w ) == 0 ) {
      return i ;
    } else {
      i = i + 1 ;
    }
  }
}
```

A sample `main` function to test the function `search` follows below. It assumes that the word is passed as the first argument to the program and that the text is the second argument.

```
  int main( int argc, char *argv[] ) {
    if( argc != 3 ) {
      printf( "Wrong number of arguments\n" ) ;
      return 1 ;
    } else {
      if( search( argv[1], argv[2], 0 ) != -1 ) {
        printf( "%s occurs\n", argv[1] ) ;
      } else {
        printf( "%s does not occur\n", argv[1] ) ;
      }
      return 0 ;
    }
  }
```

The process of searching something in a string is so common that the C library provides standard functions for searching in strings. For example, the function `strstr` implements the function `search` above. The function has a different interface. It has only two parameters, and they are in reverse order: the text comes first, followed by the word. The return value is also different, instead of returning the index of the element where the string was found, `strstr` returns a *pointer to* that element. So the return type of `strstr` is `char *` and not `int`. Instead of returning the index `i`, the function `strstr` returns the expression `s+i`, and instead of returning the integer `-1` on a failure, `strstr` returns the constant `NULL`.

Here is a revised version of the main program above that uses `strstr` instead of our own function `search`:

```
  int main( int argc, char *argv[] ) {
    if( argc != 3 ) {
      printf( "Wrong number of arguments\n" ) ;
      return 1 ;
    } else {
      if( strstr( argv[2], argv[1] ) != NULL ) {
        printf( "%s occurs\n", argv[1] ) ;
      } else {
        printf( "%s does not occur\n", argv[1] ) ;
      }
      return 0 ;
    }
  }
```

Above we have shown how addition to a pointer is used to shift an array, or to make an array slice. The addition of an array and an integer, results in a shifted view on that array. The inverse of this operation does also exist: two views on an array can be 'subtracted' to determine how far one view is shifted with respect to the other. The function `strstr` returns a pointer to the place where the string was found, but using this subtraction it is now possible to determine the index:

```
int main( void ) {
  char *q = "What a wonderful world!" ;
  char *r = strstr( q, "wonderful" ) ;
  char *s = strstr( q, "world" ) ;
  printf("Indexes %d %d, diff %d\n", r-q, s-q, s-r ) ;
  return 0 ;
}
```

The constant `q` points to the array of characters containing the string "What a wonderful world". The first call to `strstr` will return a pointer to the place where "wonderful" appears in `q`. The second call to `strstr` returns a pointer to where "world" appears. The `printf` statement prints three values:

- Firstly it prints the difference between `r` and `q`. They both point in the same array of characters, and as `q` points to the beginning, `r-q` is equal to the index of where `r` was pointing to, `7`.

- Secondly it prints the difference between `s` and `q`. This equals the index of the word "world" in `q`, which is `17`.

- The third number is `s-r`, both are indices in the array pointing to `q`, the subtraction will result in `10`, as `s` is pointing to `q[17]` and `r` is pointing to `q[7]`.

The arithmetic with pointer works just like ordinary arithmetic *as long as the pointers remain within the array bounds*. The single exception being that a pointer that is pointing just above the array is allowed, although this pointer should not be dereferenced. A number of interesting observations can now be made:

- The first element of an array `q` is indexed with `q[0]`, but as `q` is a pointer to the first element, it can also be accessed as `*q`.

- In an array `(q+2)`, the first element is accessed with `(q+2)[0]`, which is referring to the same cell as `q[2]` (refer to the picture in the beginning of this section). But in the previous bullet we observed that we could write `*(q+2)` instead of `(q+2)[0]`. Thus, `q[2]` and `*(q+2)` are actually referring to the same element. Indeed, in C, an expression of the form `q[i]` is *by definition* identical to the expression `*(q+i)`.

Pointer arithmetic is probably *the* concept for which C is (in)famous. Some people loathe it, as it can lead to programs that are completely incomprehensible, other programmers love it for the same reason. Pointer arithmetic is useful to create array slices. For example, divide and conquer type applications can set functions to work on two halves of an array by passing a pointer to the array itself and a pointer to the middle. Care should be taken however when *aliases* are created with pointer arithmetic, for example, when two slices (partially) overlap. As explained in Section 4.4.2, aliases are a problem when they are updated.

Pointer arithmetic works with arrays that are allocated dynamically in exactly the same way as with arrays that are declared in the program. Pointer arithmetic can also be used in conjunction with assignment statements. For example, if x is a pointer to some (part of) an array, then the operation x++ will cause x to point to the next cell. Again, this operation can be applied only as long as x points to the same data. To conclude this section we will show an idiomatic C function, which is used to copy a string:

```
void copystring( char *out, char *in ) {
  while( *out++ = *in++ ) {
    /* Nothing more to do */
  }
}
```

The while loop has a complicated conditional that contains three assignments. The first assignment is the = sign, it assigns one character from the in array to the out array. The other two assignments are the ++ operators, which shift both in and out one position further. To finish it off, the loop terminates if the character that is copied happens to have the same representation of false, that is the \0 character. We do not recommend to use this style of code, but you will find this style of coding in real C programs.

5.9 Combining arrays and structures

Arrays and structs can be combined freely. It is possible to construct an array of structures or a structure with one or more arrays as its components. Details of these data types are discussed using a personnel database as a running example.

In the previous chapter, we discussed the data type employee, which was designed to record information about an employee of some company. Here we will adapt the example in two ways. Firstly, slightly different information about the employee is stored; the records of the database will now contain more detailed information. Secondly, the information for a number of employees is stored; we are now actually building a database. The personnel database is intended for use by a company that sells bicycles. The information to be maintained is the name of the employee, the salary, the year of birth, and the number of bicycles sold over each of the past five years. The company has 4 employees. The manager of the company needs a program that increases the salary of each employee who sold more than 100 bicycles on average over the last five years. Here is the SML solution to the problem:

```
type employee  = (string * real * int * int array) ;
type personnel = employee array ;

(* mean : int array -> real *)
```

```
fun mean s
    = let
         val n = length s
         fun add_element sum i = sum + real(sub(s,i))
      in
         foldl add_element 0.0 (0 -- n-1) / real(n)
      end ;

(* enough_sales : employee -> bool *)
fun enough_sales (name, salary, birth, sales)
    = mean sales > 100.0 ;

(* payrise : employee -> real -> employee *)
fun payrise (name, salary, birth, sales) percent
    = (name, salary * (1.0+percent/100.0), birth, sales) ;

(* update_single : personnel -> int -> personnel *)
fun update_single p i
    = if enough_sales (sub(p,i))
         then upd(p,i,payrise (sub(p,i)) 10.0)
         else p ;

(* increase : personnel -> personnel *)
fun increase p = let
                    val n = length p
                 in
                    foldl update_single p (0 -- n-1)
                 end ;
```

The function mean from the beginning of the chapter is reused here, but with a different argument type. Here we are computing the mean of an array of integers instead of reals. The function enough_sales checks if the employee has sold enough bicycles to meet the salary increment criterion. The function update_single takes an employee and increases the salary with 10% if sufficiently many bicycles have been sold; increase will increase the salaries of all employees who sold enough bicycles. It will not alter the salary of employees who have not sold enough bicycles.

When implementing this problem in C, the first issue to consider is how to represent the data. As before, the SML tuple is represented by a struct, and the SML array is represented by a C array:

```
#define n_name     10
#define n_sales     5
#define n_personnel 4

typedef struct {
  char   name[n_name] ;
```

```
    double salary ;
    int    year_of_birth ;
    int    sold[n_sales] ;
  } employee ;

  typedef employee personnel[n_personnel] ;
```

The first three fields of the type employee specify the name (consisting of a maximum of 10 characters), the salary, and the year of birth of the employee. The fourth field is an array of 5 integers, used to store the sales over the past five years. The second type, personnel, defines an array of employee. The definition might seem to be turned inside out, since the name of the type to be defined appears in the middle of the typedef. However, typedefs follow the same syntax as variable declarations. A variable x that will hold 4 employees would have been declared:

```
  employee x[4] ;
```

The C implementations of mean and enough_sales are not difficult to write (see below). The functions update_single and increase are more interesting to implement. Increase runs a foldl over the range 0 … n-1. As has been shown before in Chapter 3, this is implemented with a for loop. The operation to be performed in the for loop is update_single:

```
  personnel update_single( personnel p, int i ) {
    if( enough_sales( p[i] ) ) {
      p[i] = payrise( p[i] ) ;
    }
    return p ;                       /* INCORRECT C */
  }

  personnel increase( personnel old, int n ) {
    int i ;
    personnel new = old ;            /* INCORRECT C */
    for( i=0 ; i<n ; i++ ) {
      new = update_single( new, i ) ; /* INCORRECT C */
    }
    return new ;                     /* INCORRECT C */
  }
```

As discussed before, C does not allow array assignment or returning an array as a function result. This means that this code fragment is illegal C. However, it does serve to explain what to do next. Solving one problem at a time, let us look at update_single first. This function passes its input array by reference. It destructively updates the input array if the i-th employee deserves a payrise. Therefore, update_single is not a pure function. We should acknowledge this in the type of the function by changing the result type to void:

```
  void update_single( personnel p, int i ) {
    if( enough_sales( p[i] ) ) {
      p[i] = payrise( p[i] ) ;
    }
```

```
}

personnel increase( personnel old, int n ) {
  int i ;
  personnel new = old ;             /* INCORRECT C */
  for( i=0 ; i<n ; i++ ) {
    update_single( new, i ) ;
  }
  return new ;                      /* INCORRECT C */
}
```

The function `update_single` is not a pure function, but in spite of that, the function `increase` *is* a pure function (if it were syntactically correct): it takes an array, and returns another, new array that has been modified so that the salaries have been updated. The destructive update of the function `update_single` has been hidden. This use and hiding of destructiveness is used extensively when designing modular programs.

To implement the function `increase` in syntactically correct C, one has to make a decision how the array `new` is to be returned. Two solutions are possible:

1. Pass an extra array to the function `increase` that serves as the output array. This is the way the function `histogram` was implemented previously.

2. Decide that the old array will no longer be necessary and reuse this array to pass the result back. This is the way that the problem with `update_single` was solved.

Here is a first attempt to complete the first solution.

```
void increase( personnel old, int n, personnel new ) {
  int i ;
  new = old ;                      /* INCORRECT C */
  for( i=0 ; i<n ; i++ ) {
    update_single( new, i ) ;
  }
}
```

This solution is not entirely correct, because arrays cannot be assigned. The correct way to assign one array to another is to use a for loop as follows:

```
void increase( personnel old, int n, personnel new ) {
  int i ;
  for( i=0 ; i<n ; i++ ) {
    new[i] = old[i] ;
  }
  for( i=0 ; i<n ; i++ ) {
    update_single( new, i ) ;
  }
}
```

Here is the second solution to implementing `increase`. This version destroys the old database.

```
void increase( personnel old, int n ) {
  int i ;
  for( i=0 ; i<n ; i++ ) {
    update_single( old, i ) ;
  }
}
```

Both implementations of `increase` are valid, but they have different semantics. The first produces a new, amended version of the database; the second version overwrites the old one. The first one is useful if the company keeps a backup of the database, whereas the second one is more efficient if the old one is no longer needed. Which version is best depends on knowledge about the intended use of the program.

The C functions needed to handle the database are shown completely below:

```
double mean( int s[], int n ) {
  int i ;
  int sum = 0 ;
  for( i=0 ; i < n ; i++ ) {
    sum = sum + s[i] ;
  }
  return (double)sum/n ;
}

bool enough_sales( employee e ) {
  return mean( e.sold, n_sales ) > 100.0 ;
}

void payrise( employee *e, double percent ) {
  e->salary = e->salary * (1 + percent/100) ;
}

void update_single( personnel p, int i ) {
  if( enough_sales( p[i] ) ) {
    payrise( &p[i], 10 ) ;
  }
}

void increase( personnel p, int n ) {
  int i ;
  for( i=0 ; i<n ; i++ ) {
    update_single( p, i ) ;
  }
}
```

In this version, we have chosen to destructively update the old database. To make the program consistent, the function payrise has also been replaced by its destructive counterpart. It is also worthwhile to notice that the function mean has been changed slightly. The function mean of the beginning of this chapter calculated a value of the type double, when given an array of values of type double. The current version of mean has an array with integers as an argument, but it returns a double. Somewhere along the line types must be coerced. We have chosen to perform the type coercion when returning the value:

```
return (double)sum/n ;
```

It would have been equally valid to sum the values in a double precision sum and to perform the typecast when adding elements to the sum.

To complete the program, a significant amount of extra code is needed to perform the output. The functions print_personnel and print_employee print the database and its records respectively:

```
void print_employee( employee e ) {
  int i ;
  printf( "%s, %f, %d, [ ", e.name, e.salary,
                            e.year_of_birth ) ;
  for( i=0 ; i<n_sales ; i++ ) {
    printf( "%d ", e.sold[i] ) ;
  }
  printf( "]\n" ) ;
}

void print_personnel( personnel p, int n ) {
  int i ;
  for( i=0 ; i<n ; i++ ) {
    print_employee( p[i] ) ;
  }
}

int main( void ) {
  personnel p = {
    { "John" , 18813.00, 1963, {  80,  90,  75, 20, 69 } },
    { "Mary" , 19900.00, 1946, {  72,  83,  75, 18, 75 } },
    { "Bob"  , 12055.45, 1969, { 120, 110, 100, 99, 99 } },
    { "Alice", 15133.50, 1972, { 200, 230,  75, 11, 35 } }
  } ;
  increase( p, n_personnel ) ;
  print_personnel( p, n_personnel ) ;
  return 0 ;
}
```

The function print_employee is called with a structure as its argument. In C, since structures are passed by value, this implies that the structure is copied. Copying is not necessary (because the function is only reading data), so often C programmers will pass a pointer to the structure instead. This forces the use of call by reference for passing the structure as a parameter. The result is a more efficient function:

```
void print_employee( employee *e ) {
  int i ;
  printf( "%s, %f, %d, [ ", e->name, e->salary,
                          e->year_of_birth ) ;
  for( i=0 ; i<n_sales ; i++ ) {
    printf( "%d ", e->sold[i] ) ;
  }
  printf( "]\n" ) ;
}
```

The call to the function `print_employee` is changed accordingly:

```
print_employee( &p[i] ) ;
```

Although more efficient, this implementation is less clear, because a user of the function `print_employee` cannot be sure that the structure is not changed anymore.

5.10 Multi-dimensional arrays with fixed bounds

C has proper support for multi-dimensional arrays *with fixed boundaries*. Such an array is denoted with a sequence of bounds; one for each dimension. A two-dimensional array for example has two bounds, and is thus an array of arrays. Usually, a `typedef` is given for a multidimensional array, which fixes the bounds. Here is the `typedef` for a 3×5 matrix of integers:

```
#define ROW 3
#define COL 5
typedef int matrix[ROW][COL] ;
```

A multi-dimensional array can be initialised by listing the desired values for each element. To initialise the 3×5 matrix we would list the desired values for the first row $(11, 12, \ldots)$, then those for the second row $(21, 22, \ldots)$ etc:

```
matrix m = { {11,12,13,14,15},
             {21,22,23,24,25},
             {31,32,33,34,35} } ;
```

A multi-dimensional array with fixed bounds can be passed to a function. Here is a function to print a `matrix`:

```
void print_matrix( matrix m ) {
  int i,k ;
  for( i = 0; i < ROW; i++) {
    printf( "row %d:", i ) ;
    for( k = 0; k < COL; k++) {
      printf( "\t%d", m[i][k] ) ;
    }
    printf( "\n" ) ;
    }
  }
```

The expression `m[0][0]` refers to the top left element, and the expression `m[ROW-1][COL-1]` refers to the bottom right element.

If the boundaries of the dimensions in a multi-dimensional array cannot be determined statically, one should use an array containing pointers to further arrays. Consult Kernighan and Ritchie for a further elaboration of this point [7, Page 110].

5.11 Summary

The following C constructs were introduced:

Array types Array types are denoted using square brackets. Static arrays must have a compile time constant size. Dynamic arrays can be created using the function `calloc`, which has two parameters, the size of an element and the number of elements to allocate, it returns a *pointer* to the first element of the array.

Array index Arrays are indexed with square brackets: a`[`i`]`, where a is the array and i is the (integer) index. The first element of an array has index 0. The same notation can be used to identify an element to be overwritten. a`[`i`] =` e overwrites the i-th element of a with e.

Array bounds C does not provide array bound checking. It is wise to either be able to argue that array bounds cannot be exceeded or put explicit checks in a program to make sure that the bounds are not exceeded.

Arrays as function arguments and result In C, an array is not a set of values, but a *pointer to* the set of values. The consequence is that when passing an array as an argument, the array is *passed by reference*. Arrays should not be returned as a function result, as only the pointer is returned, and the set of values might cease to exist when the function terminates. Arrays allocated explicitly can safely be passed around.

Assignment to whole arrays It is not permitted to assign a new contents to an entire array at once. A loop should be used to achieve this.

Pointer arithmetic Values can be added to a pointer, effectively slicing an array.

The programming principles that we have encountered in this chapter are:

- An array is a sequence, that is, a mapping from natural numbers to some set of values.

- Working with arrays gives rise to a common programming error, the *off by one error*; it is particularly easy to overrun the upper bound of an array by mistaking the length for the upper bound.

- Beware of memory leaks, that is, use `free` on all data structures allocated by `calloc`.

- Make sure that a data structure is truly redundant before freeing it. Using `free` too early results in a dangling pointer.

- Attempt to deallocate memory in the same place where the last reference to a block is destroyed. This results in neat functions.

- Use `void` returning functions if the function does its useful work by a side effect. Such functions are often called procedures. Do not use side effects in proper functions. This ensures that functions and procedures are always clearly identified.

- When side effects are necessary for efficiency reasons, try to hide them in an auxiliary function and provide a functional interface to the outside world.

Many of the sequences in this chapter that have been implemented using arrays could have been implemented with lists instead. The next chapter discusses the implementation of lists in C. It also compares the two implementations of sequences from the point of view of their efficiency.

5.12 Further exercises

Exercise 5.8 The last example program assumes that the company has 4 employees and 5 years of history. Modify the program to use dynamic arrays so that an arbitrary number of employees and years of history can be maintained. (a further extension, storing the database on a file, can be found in Chapter 7)

Exercise 5.9 Rewrite your card simulation program from Exercise 5.4 so that it plays a real game. When playing, the user should choose a secret number and the program should try to 'guess' it. Here is sample dialogue with such a program. The input of the user is underlined; the text produced by the program is in plain font.

```
Does the number appear on the first card: 1 3 5 7?  y
Does it appear on the second:             2 3 6 7?  n
And on the third:                         4 5 6 7?  n
Your secret number must be: 1!
```

Exercise 5.10 In this exercise, create a function that prints a table of the months of a year with the number of days in each month. Furthermore, write a function that when given a particular date in a year calculates what date it will be exactly one week later.

(a) Write a C function `print_three` to print the first three characters of a string. The function should have one argument (the string) and its return type should be `void`.

(b) Give the C type definition of a structure `month` with two fields: an integer `days` and a character string `name`. The string should have an appropriately chosen maximum length. Use this declaration to create an array of `month` called `leap`. Initialise the entries of the array with the number of days in each month and the *full* name of the month. You may assume that February has 29 days.

(c) Create a function `print_year` that prints a table of the months of a year with the number of days in each month. Your function should take one argument, the array of `month`. Its return type should be `void`. Furthermore, print the total number of days in the year. This should be computed by adding the days in all the months. Here is a fragment of the output that your function `print_year` should produce:

```
Jan. has 31 days
Feb. has 29 days
/*... rest of months*/
Dec. has 31 days
This year has 366 days
```

(d) Given a particular date in a year, for example February 24th, write a C function that calculates what date it will be exactly one week later. In a leap year, the answer should be March 2nd. Thus you will thus need to know how many days each month has. Use the table of part(b) for this purpose. Design a suitable interface for your function.

Exercise 5.11 The bisection method of the previous chapters can be used not only to search for the roots of a function but it can also be used to search efficiently for some special item in non-numerical data.

(a) Modify `extra_bisection` from Section 4.5 to work with discrete rather than continuous data. That is replace all types `double` by `int` and reconsider the use of `eps`, `delta` and `absolute`.

(b) Write a `main` function to call your discrete bisection function such that it will search for an occurrence of a string in a sorted array of strings. Use the arguments to your `a.out` to search for another occurrence of `a.out` to test your code. For example, the following command should produce the number `5`, and it should make only two string comparisons.

```
a.out a.1 a.2 a.3 a.4 a.out b.1
```

(c) Generalise the bisection function such that it can assume the role of the discrete as well as the continuous bisection. Then rewrite your program to use the generalised bisection rather than the discrete.

Exercise 5.12 If you use a machine with a UNIX like operating system you might try to write the following program that determines how much address space your program is allowed to use. A standard UNIX function, `brk`, attempts to set the highest address that your program may use (the 'break'). The function returns either `0` to indicate that the break was set successfully, or `1` to signal that the break could not be set that high. Use the bisection function of the previous exercise to find the highest break that you can use (assume that your system allows pointers and integers to be mixed, and that the break is somewhere between `0` and `0x7FFFFFFF`).

Exercise 5.13 A program is needed to make sure that the exam results of the first year Computer Science students at your University are correctly processed.

- **(a)** Define a two dimensional array for storing the exam results (integers in the range $0 \ldots 100$) of the first year Computer Science students. Make sure that you choose appropriate upper bounds on the array(s) for the number of students in your class and the number of modules in your course.
- **(b)** Write a function to compute the sum of all the scores in the exam results table for a particular student and another function to count the number of non-zero scores for a particular student.
- **(c)** Write a function to print the exam results as a nicely formatted table, with one line per student and one column per module:
  - Each row should be preceded by its row number.
  - Rows containing only zeroes should not be printed.
  - At the end of each row, print the sum, the number of non-zero scores and the average of the scores in the row.
- **(d)** Write a main function to create a table with some arbitrary values. Then call your print function on the table.
- **(e)** What measures have you taken to make sure that the exam results are processed and averaged correctly?

Exercise 5.14 A particular spread sheet program has the following features:

- **(a)** The program knows about a fixed maximum number of work sheets.
- **(b)** Each work sheet has a name (of a fixed maximum length), and a date and time of when it was last used.
- **(c)** Each work sheet is a rectangular 2-dimensional array of cells with bounds that will not exceed certain fixed maximum values.
- **(d)** The are four kinds of cells: a formula (represented as a string of a fixed maximum length), an integer, a real number, or a boolean.
- **(e)** The spread sheet program should be able to tell what kind of cell it is dealing with.
- **(f)** Each cell has a flag stating whether it is in use or not.

Design the C data structures required to support all these features by giving `#define` and `typedef` declarations. *Do not write the rest of a program that might use the data structures.*

Exercise 5.15 A magic square [8] of order n is a square in which the numbers $1, 2 \ldots n^2$ are arranged in the following way:

- Each number appears exactly once.
- The sum of the numbers in each row is the same.
- The sum of the numbers in each column is the same.
- The sum of the numbers in each of the two main diagonals is the same.
- All sums above are the same.

Here is a magic square of order 3 as an example. The sums of rows, columns and main diagonals of this square are all equal to 15.

6	1	8
7	5	3
2	9	4

Write a C program to print magic squares of order $1, 3, 5 \ldots 17$.

Chapter 6

Lists

The preceding chapters discussed non-recursive data structures. Chapters 2 and 3 used scalar data types such as `int`, `double`, `bool`, the enumerated types and functions. Chapter 4 used non-recursive structures to build tuples and algebraic data types using the C language elements `struct` and `union`. The resulting data structures are strictly hierarchical. These data structures are characterised by their simplicity and their inflexibility: once defined, the structure cannot be changed, although the components of the structure can be given arbitrary values.

Chapter 5 made a first step towards a more flexible data structure, the array. An array was introduced as an implementation of a sequence. One of the major problems with arrays is that they are expensive to extend and to contract. The list provides an alternative implementation of a sequence. Lists are easy to extend and contract, but lists also have disadvantages. The present chapter will see a development of a recursive data type by discussing the list. We will also compare lists to arrays. We are careful not to change the representation of data when moving from the functional to the imperative domain.

6.1 Lists of characters

The list is one of the fundamental data types of functional programming. Several examples have been discussed where a recursive algorithm operates on an arithmetic sequence. Such a sequence is just a list of integers. So far, the C implementations could be made without lists, mainly because arithmetic sequences were represented in the C implementations as for-statements. However, there are many problems that require flexible data structures such as lists, so their C implementation will be discussed. To simplify the discussion, a list of characters will be defined and used in this chapter. In Chapter 8, the approach will be generalised to polymorphic lists.

All functional languages provide lists as primitives. It is also possible to define a list explicitly. Here is the definition in SML of an algebraic data type with two constructors `Nil` and `Cons`, which is effectively a list:

```
datatype char_list = Nil
                   | Cons of (char * char_list) ;
```

A list of two characters "Hi" can now be written as:

```
(* list_hi : char_list *)
val list_hi = Cons("H",Cons("i",Nil)) ;
```

In C, a list of characters is be defined as shown below.

```
typedef struct list_struct {
  char                list_head ;
  struct list_struct  * list_tail ;
} *char_list ;
```

The char_list definition uses the pointers of C to implement a list structure. The type char_list is a pointer to a structure with two fields. The first field is called list_head and is of type char. The second field of the list structure has the name list_tail and is a pointer to something of the type struct list_struct. This type refers back to the structure being defined; it is a recursive data type that we are defining, so we should expect an element of recursion in the definition.

Each definition of a structure (or a union) may have an identifier, say this_identifier, immediately following the keyword struct. The pair struct this_identifier can then be used to refer to the type. This is the only construction that allows the definition of a recursive type, because C does not allow the use of a type identifier that is not yet defined. Hence, it would be illegal to define the character list in terms of a char_list.

In a functional program, storage allocation is automatic, but in a C program data structures must be allocated explicitly. The logical place to put list structures is on the heap. Here is a function cons that allocates a structure of the right size on the heap and then fills in the appropriate fields of the structure. This is a typical, idiomatic C function.

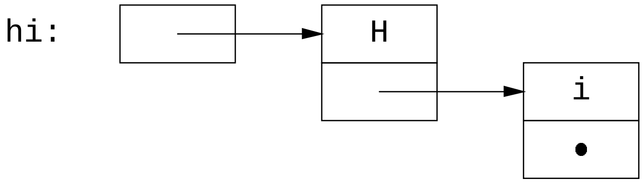
```
#define list_struct_size sizeof( struct list_struct )

char_list cons( char head, char_list tail ) {
  char_list l = malloc( list_struct_size ) ;
  if( l == NULL ) {
    printf( "cons: no space\n" ) ;
    abort( ) ;
  }
  l->list_head = head ;
  l->list_tail = tail ;
  return l ;
}
```

The function malloc allocates space on the heap. It works just like the function calloc discussed in Section 5.7, except that it allocates a block of memory just large enough to hold one value, in this example of the size of a list_struct. The contents of this cell are not initialised to 0 by malloc; instead, the value of the cell is undefined. In this case, the size of the cell is sizeof(struct list_struct), which refers to the size of the structure list_struct. The two assignment statements initialise the head and tail fields of the data structure.

Every list must be properly terminated so that functions operating on a list can find out where the list ends. The C convention for indicating the end of any list or recursive data structure is to use a null-pointer, denoted NULL. The SML value list_hi shown above can be implemented in C as follows:

```
char_list list_hi( void ) {
  char_list hi = cons(  H , cons(  i , NULL ) ) ;
  return hi ;
}
```

When executed, the function list_hi creates the heap structure shown below:



A box represents a data structure of type list_struct, and an arrow represents pointer of type char_list. The local variable hi points at the first cell of the list. The null pointer that indicates the end of the list is shown as a •; it represents the NULL value.

6.1.1 List access functions: head and tail

When accessing the elements of a list, it is convenient to have access functions for the components of the structure. This gives rise to the following two SML functions:

```
(* head : char_list -> char *)
fun head (Cons(x,xs)) = x ;

(* tail : char_list -> char_list *)
fun tail (Cons(x,xs)) = xs ;
```

These functions are readily implemented in C:

```
char head( char_list l ) {
  if( l == NULL ) {
    abort() ;
  }
  return l->list_head ;
}

char_list tail( char_list l ) {
  if( l == NULL ) {
    abort() ;
  }
  return l->list_tail ;
}
```

Often C programmers tend not to use functions for head and tail, but to inline the code directly, using `l->list_head` and `l->list_tail`. For two reasons inlining is slightly more efficient. Firstly, the function call overhead is avoided. Secondly, the test on the end of the list is avoided (`l == NULL`), which is safe if the programmer ‘knows’ that the list is not empty, similar to avoiding bound checks on arrays, see Chapter 5. The disadvantage of inlining is however that the structure of the data is exposed in several places in the program. It will be impossible to hide implementation details, a feature which will be discussed in Chapter 8. Therefore, it is better to leave inlining to the compiler.

Exercise 6.1 Write a C procedure `print_list` to print the characters of a `char_list` as a string. Use a while-statement.

Exercise 6.2 The SML data type `char_list` above represents an algebraic data type, with recursive use of the type `char_list`. The C version `char_list` shown above uses the conventional C notation `NULL` to denote the end of the list, instead of using an equivalent of the SML equivalent `Nil`. Show how the C constructs `union`, `enum`, and `struct` can be used to create a literal equivalent in C of the algebraic data type `char_list` (see also Chapter 4)

The correspondence between a selection of useful list primitives as they appear in functional languages and the newly defined list functions and constants in C is summarised below:

function	lang.	SML type or C prototype
`Cons`	SML	`char -> char_list -> char_list`
	C	`char_list cons( char, char_list )`
`Nil`	SML	`char_list`
	C	`char_list NULL`
`head`	SML	`char_list -> char`
	C	`char      head( char_list )`
`tail`	SML	`char_list -> char_list`
	C	`char_list tail( char_list )`

The following sections will see a graded series of example problems that use lists. This will create, amongst others, the implementations of the following functions:

function	lang.	SML type or C prototype
`length`	SML	`char_list -> int`
	C	`int       length( char_list )`
`nth`	SML	`char_list -> int -> char`
	C	`char      nth( char_list, int )`
`append`	SML	`char_list -> char_list -> char_list`
	C	`char_list append( char_list, char_list )`
`filter`	SML	`(char->bool) -> char_list -> char_list`
	C	`char_list filter(bool (*pred)(char),char_list)`
`map`	SML	`(char->char) -> char_list -> char_list`
	C	`char_list map( char (*f)( char ), char_list )`

6.2 The length of a list

The basic list abstraction can be put to work in the implementation of a few recursive list functions. The SML function `length` computes the length of a list, which corresponds to $\#s$ on a sequence.

```
(* length : char_list -> int *)
fun length Nil          = 0
  | length (Cons(x,xs)) = 1 + length xs ;
```

C has no pattern matching, so explicit calls to list access functions are required. To prepare for the transition to C, the `length` function must be rewritten to use access functions and a conditional instead of pattern matching:

```
(* length : char_list -> int *)
fun length x_xs = if x_xs = Nil
                     then 0
                     else 1 + length (tail x_xs) ;
```

The corresponding C implementation is now constructed using the basic technique from Chapter 2.

```
int length( char_list x_xs ) {
  if ( x_xs == NULL ) {
    return 0 ;
  } else {
    return 1 + length( tail( x_xs ) ) ;
  }
}
```

Exercise 6.3 Write a tail recursive version of `length` in SML and give the corresponding C implementation that uses a while-statement.

6.3 Accessing an arbitrary element of a list

Another useful function, nth, accesses an arbitrary element from a list. This corresponds to the access operator on sequences $s(\cdot)$.

```
(* nth : char_list -> int -> char *)
fun nth (Cons(x,xs)) 0 = x
  | nth (Cons(x,xs)) n = nth xs (n-1) ;
```

The pattern matching can be removed from nth. This yields an SML-function that can be directly implemented in C.

```
(* nth : char_list -> int -> char *)
fun nth x_xs n = if n = 0
                    then head x_xs
                    else nth (tail x_xs) (n-1) ;
```

The function nth is tail recursive; this makes it straightforward to use the while-schema from Chapter 3:

```
char nth( char_list x_xs, int n ) {
  while( n != 0 ) {
    x_xs = tail( x_xs ) ;
    n-- ;
  }
  return head( x_xs ) ;
}
```

Exercise 6.4 When trying to access a non-existent list element, the SML definition of nth raises the exception Match, but the C implementation of nth produces an undefined result. Modify the C version of nth so that it aborts with an error message when a non-existent list element is accessed.

6.4 Append, filter and map: recursive versions

The last three generally useful list processing functions, append, filter, and map, are functions that are not tail recursive. It is possible to develop efficient C versions using a technique known as *open lists*. Below, we will first discuss three simple recursive (and inefficient) versions; the next section discusses the optimisation technique.

6.4.1 Appending two lists

The function append corresponds to concatenation of sequences. When given two separate lists, it creates a new list from the concatenation of the two lists.

The recursive SML definition is:

```
(* append : char_list -> char_list -> char_list *)
fun append Nil          ys = ys
  | append (Cons(x,xs)) ys = Cons(x,append xs ys) ;
```

Exercise 6.5 Rewrite the recursive definition of append to use list primitives and conditionals rather than pattern matching.

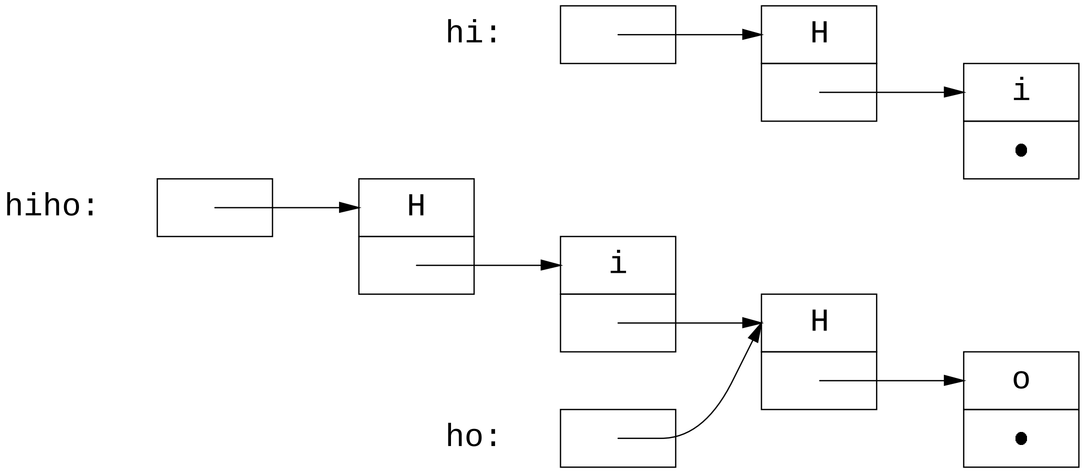
The function append can be translated into a recursive C equivalent as shown below. How to create a version that uses a while-loop is discussed in Section 6.5:

```
char_list append( char_list x_xs, char_list ys ) {
  if( x_xs == NULL ) {
    return ys ;
  } else {
    return cons( head( x_xs ),
                 append( tail( x_xs ), ys ) ) ;
  }
}
```

When applied to the two argument lists, the append function copies the first list and puts a pointer to the second list at the end of the copy. The copying is necessary, as append cannot be sure that the first argument list is not going to be needed again. To illustrate this important point, consider a concrete example of using append. This is the function list_hi_ho below. It appends the two lists hi and ho into the result list hiho.

```
char_list list_hi_ho( void ) {
  char_list hi = cons(  H , cons(  i , NULL ) ) ;
  char_list ho = cons(  H , cons(  o , NULL ) ) ;
  char_list hiho = append( hi, ho ) ;
  return hiho ;
}
```

Executing the function list_hi_ho creates the two heap structures shown below:



The local variables hi, ho, and hiho are each shown to point to the relevant parts of the structures. The list pointed to by the variable hi has been copied by append. The other list, pointed at by ho, is shared. The function list_hi_ho returns the result list hiho. The list pointed at by hi is by then no longer accessible, the list pointed at by ho is accessible as part of the result list. The heap storage for the inaccessible hi list can be reclaimed. The storage occupied by the ho list cannot be reclaimed, because it is still accessible. In general, it is difficult to determine which heap structures will be redundant and which structures are still in use. Consequently reusing heap data structures should be done with care. See also Section 6.8.

6.4.2 Filtering elements from a list

Filtering is a generally useful operation over lists. When given a predicate pred and an input list, the filter function selects only those elements for which the predicate returns true:

```
(* filter : (char->bool) -> char_list -> char_list *)
fun filter pred Nil
    = Nil
  | filter pred (Cons(x,xs))
    = if pred(x)
         then Cons(x,filter pred xs)
         else filter pred xs ;
```

Exercise ⋆ 6.6 Give the specification of filter in terms of sequences.

The filter function can be written directly in C, using the techniques that have been developed earlier in this chapter:

```
char_list filter( bool (*pred)( char ), char_list x_xs ) {
  if ( x_xs == NULL ) {
    return NULL ;
  } else {
    char x = head( x_xs ) ;
    char_list xs = tail( x_xs ) ;
    if( pred( x ) ) {
      return cons( x, filter( pred, xs ) ) ;
    } else {
      return filter( pred, xs ) ;
    }
  }
}
```

Let us now use filter to select all digits from a list of characters. The SML version is:

```
(* filter_digit : char_list -> char_list *)
fun filter_digit xs = filter digit xs ;
```

The function `filter_digit` uses an auxiliary function `digit` as the predicate:

```
(* digit : char -> bool *)
fun digit x = x >= "0" andalso x <= "9" ;
```

A higher order function such as `filter` is a powerful abstraction. It encapsulates the full mechanism required to traverse the input list, whilst offering complete freedom in the choice of an appropriate predicate. Instead of filtering out digits, any other (type correct) predicate can be supplied as an argument to filter. For the software engineer, abstractions such as `filter` are attractive, as they allow one problem to be solved once and for all. Solutions to more complicated problems can be built on that basis.

In C, `filter` can be used for the same purpose, that is, to filter out elements from the list, such as digits or characters greater than the letter 'A', and so on:

```
char_list filter_digit( char_list xs ) {
  return filter( digit, xs ) ;
}
```

Here is the C version of `digit`:

```
bool digit( char x ) {
  return x >=  0  && x <=  9  ;
}
```

An interesting problem arises when the predicate needs some extra information. Consider filtering out elements greater than a certain 'pivot' value `p`. The SML code would be written like this:

```
(* filter_greater : char -> char_list -> char_list *)
fun filter_greater p xs
    = let
         fun greater_p x = x > (p:char)
      in
         filter greater_p xs
      end ;
```

The pivot `p` is passed by `greater_p` directly to the operator `>`. The function `greater_p` is a partially applied version of the comparison operator `>`, as it already knows which right operand to use, `p`. The use of a local function definition is essential here, since it encapsulates the value of `p`. If we were to lift the definition of `greater_p` out of the `let` construct, the value of `p` would be unknown, for it would be out of its defining scope.

In C, local function definitions are not permitted. The solution to this problem, as discussed in Chapter 4, will be used here also. Instead of encapsulating the pivot `p` in the local function definition, as in the SML version, the C version passes the pivot explicitly to a new version of `filter` with an extra argument. Unfortunately, this means that the C version of `filter` has to be modified. The modifications consist of adding an extra argument `arg` to both `filter` and the predicate `pred`:

```
char_list extra_filter( bool (*pred)( void *, char ),
                        void * arg, char_list x_xs ) {
```

```
    if ( x_xs == NULL ) {
      return NULL ;
    } else {
      char x = head( x_xs ) ;
      char_list xs = tail( x_xs ) ;
      if( pred( arg, x ) ) {
        return cons( x, extra_filter( pred, arg, xs ) ) ;
      } else {
        return extra_filter( pred, arg, xs ) ;
      }
    }
  }
```

The original C version of `filter` was not general enough. To see `extra_filter` in action, consider again the problem of filtering out elements greater than the pivot `p` from a list of characters.

```
  char_list filter_greater( char p, char_list xs ) {
    return extra_filter( greater, &p, xs ) ;
  }
```

The address of the pivot `p` is passed as the second argument to `extra_filter`, which in turn will pass it to the predicate `greater`. The latter dereferences its pointer argument `arg`:

```
  bool greater( void *arg, char x ) {
    char * c = arg;
    return x > *c ;
  }
```

The function `greater` accesses the pivot value by dereferencing the pointer `arg`, which points to the pivot value `p`.

6.4.3 Mapping a function over a list

Another powerful higher order abstraction of list operations is to map a function over a list, that is, to apply a certain function `f` to all elements of a list. Here is the SML version of `map`:

```
  (* map : (char->char) -> char_list -> char_list *)
  fun map f Nil          = Nil
    | map f (Cons(x,xs)) = Cons(f x,map f xs) ;
```

Exercise ⋆ 6.7 Give the specification of `map` in terms of sequences.

Exercise ⋆ 6.8 Prove that the SML-function `map` shown above satisfies the specification given in Exercise 6.7.

The map function can be written directly in C:

```
char_list map( char (*f)( char ), char_list x_xs ) {
  if( x_xs == NULL ) {
    return NULL ;
  } else {
    char x = head( x_xs ) ;
    char_list xs = tail( x_xs ) ;
    return cons( f( x ), map( f, xs ) ) ;
  }
}
```

Exercise 6.9 Generalise map to extra_map in the same way as filter has been generalised to extra_filter.

This concludes the first encounter of lists in C. The primitives append, map, extra_map, filter, and extra_filter may be inefficient still. This issue will be revisited in the next section. All implementations are modular, so they would be appropriate building blocks for many list processing applications in C programs.

6.5 Open lists

The conceptually simple operation of making a copy of a list is surprisingly difficult to implement efficiently. In this section, a list copy function will be discussed to illustrate a useful optimisation technique that can be applied in a C implementation but not in the (pure) functional world. An SML function to copy a list is:

```
(* copy : char_list -> char_list *)
fun copy Nil          = Nil
  | copy (Cons(x,xs)) = Cons(x,copy xs) ;
```

After rewriting this function without pattern matching, a recursive C implementation can be derived using the while-schema:

```
char_list copy( char_list x_xs ) {
  if( x_xs == NULL ) {
    return NULL ;
  } else {
    return cons( head( x_xs ), copy( tail( x_xs ) ) ) ;
  }
}
```

Although this implementation is efficient in time (the list copy will take a time proportional to the length of the list), the implementation is not tail recursive. This means that the implementation requires stack space proportional to the length of the list, which is undesirable.

The technique available for turning a non-tail recursive function into a tail recursive one reverses the sequence of events. In this case, it would mean one of two possibilities:

- To traverse the list x_xs from right to left and to build the result list up from right to left
- To traverse and construct from left to right.

Both options go against the flow of one list in order to go with the flow of the other. Here is the first solution:

```
(* copy  : char_list -> char_list -> char_list *)
fun copy  accu Nil
    = accu
  | copy  accu (Cons(x,xs))
    = copy  (append accu (Cons(x,Nil))) xs ;

(* copy : char_list -> char_list *)
fun copy xs = copy  Nil xs ;
```

The tail-recursive function copy uses an accumulating argument to assemble the result list while traversing the input list. Each new input element is attached at the end of the result list using append. This makes copy inefficient, because it creates a completely new list for every element of the original input list. If the input list contains n elements, the total number of Cons cells created is $1 + 2 + 3 + \ldots n = n(n+1)/2$. Thus by writing a tail recursive function, we have made the problem worse rather than better. Why then have we then gone through all this trouble? The answer is that the calls to append are not really necessary. With some programming effort, we can improve the tail recursive version of copy such that it will only allocate n Cons cells, whilst still being tail recursive. The solution that we will obtain is efficient, as it does precisely the amount of work that one would expect, using only a constant amount of stack space. However, there is a fly in the ointment: the efficient solution cannot be programmed in SML. Fortunately it can be written nicely in C.

Exercise 6.10 Write a version of copy that traverses the input list from the right, using foldr. Then translate the result into C.

Consider the following append based C implementation of the tail recursive version of copy:

```
char_list copy( char_list xs ) {
  char_list accu = NULL ;
  while( xs != NULL ) {
    accu = append( accu, cons( head( xs ), NULL ) ) ;
    xs = tail( xs ) ;
  }
  return accu ;
}
```

This implementation is inefficient because of the use of the append function. When we discussed the append function earlier it was noted that append has to copy its first argument, so that the (pointer to) the second argument can be attached at the end of the copy. We stated that this is necessary because, in general, one does not know whether the first argument is shared. It is not safe to change a potentially shared argument. However, in this case, we do know that the first argument accu of append is not shared.

Inspecting the code of copy above, we see that initially the accumulator represents the empty list. Appending a new list to the empty list merely returns the new list; therefore, the second time around the while loop, the first argument to append is the list that was created during the first round. Therefore, this list is not shared, and it is safe to change it. The same reasoning applies to subsequent iterations, so that throughout the execution of the while loop, we can be sure that the first argument of append is always a list that is not shared. Thus, it can be changed safely.

The efficient version of our list copy problem uses this information by applying the following *open list* technique: by remembering where the previous iteration has appended an element, the next element can be appended at that point, without requiring further list traversal. The name 'open list' stems from the fact that at each stage, the result list is still open ended, as more elements are yet to be added to the right.

There are two ways to remember this last element, but neither exist in the functional world. The first method is relatively clean, but requires some code duplication. The second method is less clear, but more efficient. Both methods are used in real C programs, so they are discussed below.

6.5.1 Open lists by remembering the last cell

A relatively clean way to complete the implementation of the list copy function is to remember where the last element has been appended to the output list. To do this, there must be a cell that can be identified with the last element. The copy function will now be completed in three steps. As the first step, the while-statement will be *unrolled* once. A while-statement can always be unrolled based on the following observation:

```
                                 if( condition ) {
                                   statement ;
while( condition ) {               while( condition ) {
  statement ;              ≡         statement ;
}                                  }
                                 }
```

The result of unrolling is that the first iteration of the loop is executed before the while. The statements of the first iteration, now separate from the statements of subsequent iterations, can be optimised using the knowledge that all variables have their initial values. This information will help to create an efficient function.

The while-statement in copy above can be unrolled, yielding the following function. It looks complicated because of the code duplication, but these complications will be optimised away shortly:

```
char_list copy( char_list xs ) {
  char_list accu = NULL ;
  if( xs != NULL ) {
    accu = append( accu, cons( head( xs ), NULL ) ) ;
    xs = tail( xs ) ;
    while( xs != NULL ) {
      accu = append( accu, cons( head( xs ), NULL ) ) ;
      xs = tail( xs ) ;
    }
  }
  return accu ;
}
```

The second step towards a solution is to look closely at the fourth line where the first of the two assignments is done:

```
accu = append( accu, cons( head( xs ), NULL ) ) ;
```

At this point, the list accu is guaranteed to be NULL. Thus the statement at line 4 may safely be replaced by:

```
accu = cons( head( xs ), NULL ) ;
```

This removes the first use of append. To remove the second use of append, the third and final step introduces a new variable last which remembers where the last element has been attached to the result list:

```
char_list copy( char_list xs ) {
  char_list accu = NULL ;
  char_list last ;
  if( xs != NULL ) {
    last = accu
         = cons( head( xs ), NULL ) ;
    xs = tail( xs ) ;
    while( xs != NULL ) {
      last = last->list_tail
           = cons( head( xs ), NULL ) ;
      xs = tail( xs ) ;
    }
  }
  return accu ;
}
```

Here, we are using a convenient shorthand available in C for assigning the same value to several variables at once:

```
last = accu
     = cons( head( xs ), NULL ) ;
```

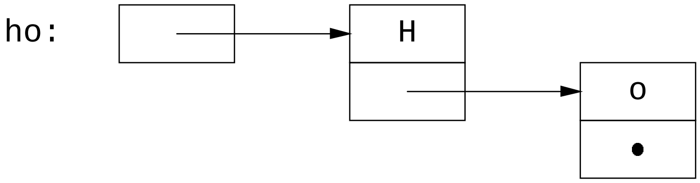
The statement above has exactly the same meaning as the two statements below, because the second assignment is used as an expression (explained in Chapter 3):

```
accu = cons( head( xs ), NULL ) ;
last = accu ;
```

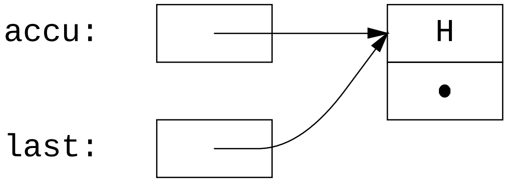
To illustrate the working of the open list based function `copy`, consider the following C function:

```
char_list list_ho( void ) {
  char_list ho = cons(  H , cons(  o , NULL ) ) ;
  char_list copy_ho = copy( ho ) ;
  return copy_ho ;
}
```

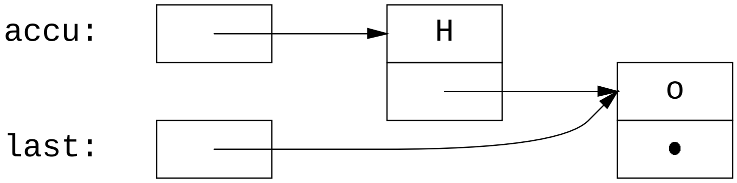
The list `ho` is created first:



After creation, the list is copied to `copy_ho`. The first step makes a copy of the first `list_struct` cell:



The next step remembers, through the use of the variable `last`, where the previous step has put its data and attaches the copy of the next cell at this point:



The resulting C implementation is perhaps a bit complicated but it is efficient. The price one often has to pay for efficiency is loss of clarity and conciseness. As a general rule, this is acceptable provided a relatively complex function has a well defined and clean interface so that it can be used without much knowledge of its internals. Clearly, this is the case with the list copying function.

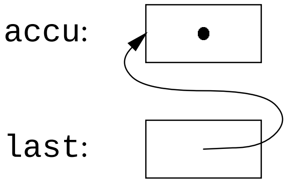
6.5.2 Open lists by using pointers to pointers

The first method to complete the `copy` function from the previous section remembers a pointer to the last cell added. In this section, we describe another method.

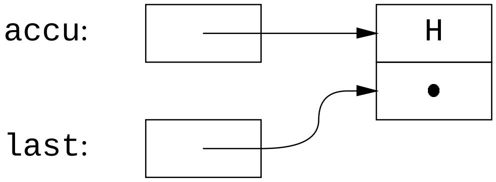
It uses a pointer that remembers which pointer to overwrite in order to append the next cell. This is a more advanced use of pointers and results in a concise and idiomatic C function. The complete solution is shown below; let us consider the relevant aspects of the solution. The type of the variable `last` that remembers which field to overwrite is `char_list *`: it points to a pointer in the result list. This result list is constructed starting from `accu`. Thus the pointer `last` initially points at `accu`. Subsequently, `last` will refer to the `tail` field of the last cell of the result list. The address-of operator & is instrumental here.

```
char_list copy( char_list xs ) {
  char_list accu = NULL ;
  char_list *last = &accu ;
  while( xs != NULL ) {
    char_list new = cons( head( xs ), NULL ) ;
    *last = new ;
    last = &new->list_tail ;
    xs = tail( xs ) ;
  }
  return accu ;
}
```

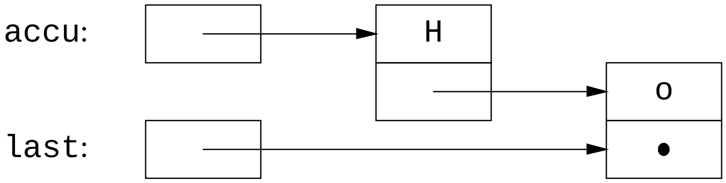
To help understand how the above function `copy` works, consider the diagrams below. It shows three stages of the process of copying the list produced by `cons( H ,cons( o ,NULL))`. Initially the result list is empty, and `accu` contains a NULL pointer. At this stage `last` points to `accu`:



The first assignment statement of the while-statement then creates a new `cons` cell, and the second assignment-statement stores this pointer via `*last` at the end of the list, which is initially in `accu`. The third statement then updates `last` so that it points to the new tail of the list, the element `tail_list` of the new node. The states of `accu` and `last` are shown graphically below:



The next iteration creates a new cell `o` and uses `last` to place the new cell at the end of the list. The variable `last` is updated to point to the end of the list again, which results in the following state:



This is the last iteration of the loop, so the function returns the newly created list.

This completes the description of two efficient implementations of copy based on open lists. These two implementations are functionally identical (as they are identical to the naive implementation that would use a call to append). The implementations are also good building blocks: The internals of the two implementations may be non-functional, but the functions provide a clean interface to their callers without side effects.

6.5.3 Append using open lists

When the append function was introduced earlier in this chapter, we noted that the efficiency could still be improved by making the function tail recursive. The structure of append is similar to that of copy. Here is a comparison between the two:

```
fun append  Nil ys                  fun copy  Nil
        = ys                                = Nil
  | append (Cons(x,xs)) ys            | copy (Cons(x,xs))
        = Cons(x,append xs ys) ;            = Cons(x,copy xs) ;
```

The differences between copy and append are few:

Argument: append has an extra argument ys.

End of input: where copy returns the empty list Nil, append returns its extra argument list ys.

Therefore, taking the efficient loop-based C implementation of copy from Section 6.5.1 and making changes corresponding to the differences noted above should give an efficient C implementation of append. Here is the result, annotated with the changes:

```
char_list append( char_list xs,
                  char_list ys ) {   /*added char_list ys*/
  char_list accu = ys;              /*replaced NULL by ys*/
  char_list last ;
  if( xs != NULL ) {
    last = accu
         = cons( head(xs), ys ) ;   /*replaced NULL by ys*/
    xs = tail( xs ) ;
    while( xs != NULL ) {
      last = last->list_tail
           = cons( head(xs), ys ) ; /*replaced NULL by ys*/
      xs = tail( xs ) ;
    }
  }
  return accu ;
}
```

Programming with open lists can be rewarding if the functions are derived in a systematic fashion. However, it is easy to make mistakes, so care should be taken when using this technique.

Exercise 6.11 Write a version of `append` that uses the advanced pointer technique for open lists from Section 6.5.2.

Exercise 6.12 Page 191 shows a recursive version of `map`. Implement a version of `map` that uses open lists.

Exercise 6.13 Page 188 shows a recursive version of `filter`. Implement a version of `filter` that uses open lists.

6.6 Lists versus arrays

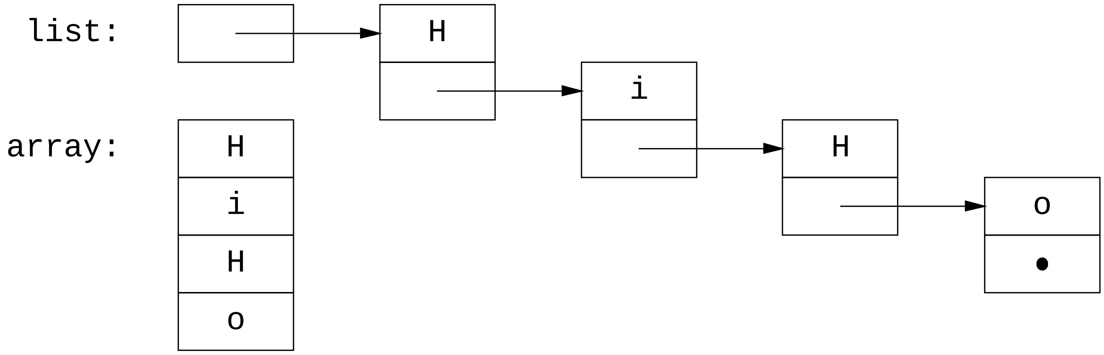
The previous chapter discussed the array and, in this chapter, we have introduced lists. From a mathematical point of view, arrays and lists are the same. Both represent a sequence, that is, an ordered set of elements. From an algorithmic point of view, there are differences. The table below gives the cost of each of the 'abstract' operations that can be performed on a sequence of n elements.

operation	representation	
	list	array
create a sequence of n elements	$\mathcal{O}(n)$	$\mathcal{O}(n)$
extend with new first element	$\mathcal{O}(1)$	$\mathcal{O}(n)$
extend with new arbitrary element	$\mathcal{O}(n)$	$\mathcal{O}(n)$
remove first element	$\mathcal{O}(1)$	$\mathcal{O}(n)$
remove arbitrary element	$\mathcal{O}(n)$	$\mathcal{O}(n)$
access first element	$\mathcal{O}(1)$	$\mathcal{O}(1)$
access arbitrary element	$\mathcal{O}(n)$	$\mathcal{O}(1)$

As an example, the last entry in the table 'access arbitrary element' states that a list requires an amount of work proportional to n, whereas an array can deliver an arbitrary element in constant time. To illustrate this point, consider the C function `list_array` below:

```
void list_array( void ) {
  char_list list = cons( H ,cons( i ,
                   cons( H ,cons( o ,NULL)))) ;
  char array[4]  = { H , i , H , o };
}
```

The store arrangement for the data structures allocated for `list` and `array` are:



The array is stored as a contiguous block. Therefore the array requires less space, because no pointers are necessary to connect the elements of the data structure, as for lists. More importantly, it is possible to calculate offsets from the beginning of the array and to use these offsets to access an arbitrary element directly. On the other hand the fact that a list is not stored as a contiguous block, but as a chain of blocks, makes it inexpensive to attach a new element to the front of the list. An array would have to be copied in order to achieve the same effect.

In designing algorithms that operate on a sequence, the appropriate data representation must be selected. Depending on the relative importance of the various operations, the list or the array will be more appropriate. There are other data structures that may be used to represent a sequence, such as streams, queues, and trees. These will have different characteristics and may therefore be more appropriate to some applications. Streams are discussed in Chapter 7 and trees are the subject of a number exercises at the end of this chapter. A discussion of the more advanced data structures can be found in a book on algorithms and data structures, such as Sedgewick [12].

6.6.1 Converting an array to a list

To study the difference between an array and a list more closely, we consider the problem of transferring the contents of an array into a list and the conversion from a list into an array. As before, the elements of both data structures will be characters to simplify the presentation.

The following SML program converts an array to a list. It first creates a list of all possible index values of the array and then accesses each array element using the appropriate index value.

```
(* array_to_list : char array -> char_list *)
fun array_to_list s
    = let
          val l = 0
          val u = length s - 1
          fun subscript i = sub(s, i)
      in
          map subscript (l--u)
```

```
        end ;
```

The lower bound of the array `s` is `l=0`, and the upper bound is `u=length s - 1`. Thus, the range of possible index values is `l` … `u`.

The function `array_to_list` is inefficient for several reasons. Firstly, it creates the list of index values as an intermediate data structure, which is created and then discarded. Secondly, it uses the non-tail recursive function `map`, which causes the solution to require an amount of stack space proportional to the length of the array.

There are alternative solutions that do not suffer from these problems. To avoid the intermediate list, one could write a directly recursive solution. This will be left as an exercise. We will explore a second alternative that yields the optimal efficiency.

Exercise 6.14 Rewrite `array_to_list` as a non-tail recursive function and translate the result into a recursive C function.

To create an efficient, iterative version of the function `array_to_list`, the for-schema is needed, because an expression of the form `(l -- u)` must be translated. However, the for-schema cannot be applied directly to the function `array_to_list`, because it does not contain an application of either `foldl` or `foldr`; it uses `map` instead. There is a formal relationship between `foldr` and `map` that will be of use to us. Here is a reminder of what `map` and `foldr` accomplish on a list (of three elements to keep the example simple). Here `f` is a unary function, and $\oplus$ is a binary operator such that $x \oplus ys = f\,x :: ys$.

$$\begin{array}{llllllll}
\texttt{map}\ f & (x_1 & :: & (x_2 & :: & (x_3 & :: & []))) \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & (f\ x_1 & :: & (f\ x_2 & :: & (f\ x_3 & :: & []))) \\
= & | & | & | & | & | & | & | \\
 & (x_1 & \oplus & (x_2 & \oplus & (x_3 & \oplus & []))) \\
 & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
\texttt{foldr}\ \oplus\ [] & (x_1 & :: & (x_2 & :: & (x_3 & :: & [])))
\end{array}$$

The relationship illustrated above is when given that $x \ \oplus \ ys = f \ x \ :: \ ys$ then we have:

$$\texttt{map}\ f\ xs \;=\; \texttt{foldr}\ \oplus\ []\ xs \tag{6.1}$$

Exercise $\star$ 6.15 Prove (6.1) for finite lists by induction over the length of the list `xs`.

Using (6.1) the SML function `array_to_list` is transformed to:

```
(* array_to_list : char array -> char_list *)
fun array_to_list s
    = let
          val l = 0
          val u = length s - 1
          fun sub_cons i xs = Cons(sub(s, i), xs)
       in
          foldr sub_cons Nil (l -- u)
       end ;
```

The right folding decreasing for-schema yields the C function:

```
char_list array_to_list( char s [], int n ) {
  int l = 0 ;
  int u = n - 1 ;
  char_list list = NULL ;
  int i ;
  for( i = u; i >= l; i-- ) {
    list = cons( s[i], list ) ;
  }
  return list;
}
```

The C function array_to_list efficiently transfers the contents of an array of characters into a list of characters. The reason why an efficient for-loop based version could be developed is because the cost of indexing an arbitrary element of the array is $\mathcal{O}(1)$. Thus, accessing the elements of the array from left to right or from right to left is immaterial. In the next chapter, similar examples will appear that do not have this property. In one of these examples, text will be read from a stream, which must happen strictly from left to right. The consequences of this seemingly innocuous property will be far reaching.

6.6.2 From a list to an array

The dual problem of transferring an array into a list is creating an array from a list. This is what the foldl based SML function below achieves:

```
(* list_to_array : char_list -> char array *)
fun list_to_array xs
    = let
          val n = length xs
          val u = n - 1
          val s = array(n, " ")
          fun update s i = upd(s,i,nth xs i)
       in
          foldl update s (0 -- u)
       end ;
```

The function list_to_array starts by creating a new array s of the right size. The lower bound of the array is assumed to be 0; the upper bound u is then one less than the length n of the list xs. Subsequently, the expression foldl update s (0 -- u) runs trough the permissible index range, calling upon the auxiliary function update to first access the required elements from the list xs and then to update the array s. The new values are obtained by indexing the list using the list index function nth.

The foldl version of the function array_to_list can be transformed directly into a C implementation by the for-schema. After simplification, the result is:

```
char * list_to_array( char_list xs ) {
  int n = length( xs ) ;
  char * array = malloc( n ) ;
  int i;
  if( array == NULL ) {
    printf( "list_to_array: no space\n" ) ;
    abort( ) ;
  }
  for( i = 0; i < n; i++ ) {
    array[i] = nth( xs, i ) ;
  }
  return array;
}
```

The implementation of list_to_array is not efficient. The reason is that the nth function accesses an arbitrary element. This is an $\mathcal{O}(n)$ operation on a list (see the table on page 198). For each element of the list xs, the nth function must start again at the beginning, counting list elements until it has reached the desired element. This is wasteful, as list_to_array traverses the list sequentially from left to right. However, it should be possible to remember which point the previous iteration of the for-loop has reached, so that the current iteration can access the next list element.

A new version of list_to_array can be written based on this idea. It has a different structure because the index range and the list of values must be traversed simultaneously. The new version could have been written in a 'fold' form as before, but not without some complications. The explicit recursive variant is more appropriate in SML:

```
(* list_to_array : char_list -> char array *)
fun list_to_array xs
    = let
          val n = length xs
          val s = array(n, " ")
          fun traverse s i Nil
              = s
            | traverse s i (Cons(x,xs))
              = traverse (upd(s,i,x)) (i+1) xs
      in
          traverse s 0 xs
      end ;
```

The explicitly recursive variant of list_to_array above creates a fresh array like its predecessors. It then calls upon the auxiliary function traverse to traverse the index range and the list of values simultaneously. Each recursive invocation of the auxiliary function causes the array s to be updated with a new value.

The explicitly recursive form of the new list_to_array allows for the multiple argument while-schema to be applied. After simplification, this yields the following C function:

```
char * list_to_array( char_list xs ) {
  int n = length( xs ) ;
  char * array = malloc( n ) ;
  int i = 0 ;
  if( array == NULL ) {
    printf( "list_to_array: no space\n" ) ;
    abort( ) ;
  }
  while( xs != NULL ) {
    array[i] = head( xs ) ;
    i = i+1 ;
    xs = tail( xs ) ;
  }
  return array;
}
```

For each iteration, both the index i range and the list of values xs are advanced.

The C function list_to_array efficiently transfers the contents of a list of characters into an array of characters because the list is traversed from left to right. At each step the head and the tail of a list are accessed, these are both efficient operations ($\mathcal{O}(1)$).

The moral of the story is that it is a good idea to 'go with the flow'. If a data structure supports some operations more efficiently than others, one should use the efficient ones. In the case of lists, this means a traversal from left is preferred above a traversal from the right.

Exercise 6.16 C implements a string as an array of characters. Implement a new string data type and a library of functions that represent a string as a list of characters. Implement functions for stringcat, stringcmp, stringncmp, and stringlen. Also provide functions to convert between array and list based strings. Can you store the \0 character in the list based string?

Exercise 6.17 Storing a single character in each cons cell makes the list of characters representation uneconomical in terms of its space usage. Reimplement the string library so that it uses a list of arrays, using the following C data structure:

```
typedef struct string {
  char data[ 32 ] ;
  int length ;
  struct string *next ;
} *string ;
```

Each cell can store up to 32 characters, the number of characters stored in a cell is specified in the `length` field. Strings of more than 32 characters are stored in cells that are linked via their `next` fields.

6.7 Variable number of arguments

It is sometimes useful to be able to write a single function that can be applied to a variable number of arguments. Good examples of such functions are `printf` and `scanf`.

In SML and other functional languages, there are two ways to write such functions. If all arguments have the same type, they can be collected in a list, and passed to the function. If the arguments have different types, an Algebraic Data Type can be used to define which combinations of types are legal, and used to pass the arguments. (This in addition to *overloaded* functions, which we will not discuss here.) As an example, the standard function `sum` computes the sum of a series of numbers, which are passed in a list.

The same solution could be used in C, but since C does not have the syntax to support easy creation and matching of lists and algebraic data types, this strategy results almost always in unreadable programs. To allow the programmer to pass a variable number of arguments, C has a built-in mechanism for *variable arguments* lists, also known as *vararg* lists.

The concept of a variable argument list in C is simple. Instead of specifying the arguments precisely in a prototype, one must specify them with the ellipsis notation `...`, which stands for 'any additional arguments of unspecified type'. Thus, the prototype of a function `sum` that would add a certain number `n` of values would read:

```
int sum( int n, ... ) ;
```

After this declaration, the following calls are legal:

```
sum( 3, 5, 1, 9 ) == 15    &&
sum( 2, 314, 52 ) == 366   &&
sum( 0 ) == 0
```

The `...` notation in the prototype allows the programmer to indicate that any number of arguments may be passed to this function. To use these arguments, the function call must specify how many actual arguments there are and the function definition must be written such that the arguments can be retrieved from the argument list. Arguments are retrieved one at the time via an *argument list pointer* and using three functions that operate on this pointer.

The type of the argument list pointer is `va_list`, which stands for variable argument list. This type is declared in the include file `stdarg.h`. A variable of type `va_list` must be initialised with a call to `va_start`. Successive elements can be retrieved with calls to `va_arg`, while the function must end with a call to `va_end`. As an example of its use, the function `sum` is defined below with a variable argument list:

```
#include <stdarg.h>
```

```
int sum( int n, ... ) {
  int i, a ;
  int accu = 0 ;
  va_list arguments ;              /* Declaration */
  va_start( arguments, n ) ;    /* Initialisation */
  for( i=0 ; i<n ; i++ ) {
    a = va_arg( arguments, int ) ;   /* Retrieval */
    accu = accu + a ;
  }
  va_end( arguments ) ;                 /* Ending */
  return accu ;
}
```

The declaration of the pointer to the argument list, `arguments`, is straightforward. The initialisation is slightly unusual, `va_start` has two arguments: the argument pointer to be initialised, and the *last proper argument* of the function, `n` in this case. The result of calling `va_start` will be that the argument pointer refers to the first of the variable number of arguments. The call to `va_arg`, in the line marked `Retrieval`, will take one argument from the list. `va_arg` has two arguments: the argument pointer, and the *type* of the argument to be retrieved. In this case all parameters are integers, therefore the type is `int`. Finally, when the arguments have been processed, `va_end` is called to close the list of arguments.

To understand the program better, we can trace the execution. The type `va_list` is in reality some kind of pointer type. As usual in C, the declaration of a pointer leaves the pointer in a undefined state. The call to `va_start` initialises the pointer. This might seem odd, as a function cannot modify its argument as a side effect. Indeed, `va_start` is not a function, but a *macro*. Macros are explored in detail in Chapter 8. For now it suffices to know that this particular macro leaves a *pointer* to the first argument in the variable `arguments`.

The call to `va_start` has two arguments, the pointer to the argument list, and the *type* of the parameter. Types cannot be passed to functions, therefore `va_arg` is, like `va_start`, a macro. The value of `va_arg` is of the specified type, and it returns the value of the argument assuming that the argument passed has that type. It then advances the argument pointer so that it points to the next argument.

The type of the arguments is not verified. If a `double` is passed to `sum`, this function will retrieve an integer, and end up with a random value. Also, arguments are not coerced, like otherwise when you pass the constant `2` to a function expecting a `double`. The variable argument list is not a type safe feature, this in contrast with the use of safe algebraic data types in SML.

A function that has a variable argument list with mixed types is the function `printf`. This function has a string as its first argument, and then a list other values: strings, integers, floating point values, and so on. We can now explain why the type specified in the format string, must match the type of the value passed as the argument. The function `printf` parses the format string to determine which types the arguments have. It can only retrieve the arguments if the types match.

This process is outlined below:

```
void printf( char *format, ... ) {
  va_list ptr ;
  int i ;
  va_start( ptr, format ) ;
  for( i=0 ; format[i] !=  \0  ; i++ ) {
    if( format[i] ==  %  ) {
      switch( format[i+1] ) {
        case  d  : {
          int d = va_arg( ptr, int ) ;
          /*print the integer d*/
          break ;
        }
        case  f  : {
          double f = va_arg( ptr, double ) ;
          /*print the double f*/
          break ;
        }
        /*other cases*/
      }
      i++ ;
    } else {
      /*print format[i]*/
    }
  }
  va_end( ptr ) ;
}
```

When a `%d` format is encountered, an argument of the type `int` is retrieved from the argument list. From then on `printf` knows that the argument `d` must be rendered as an integer. Similarly, when a `%f` format is encountered, a floating point number is retrieved, and so on. The function cannot check that it was called with the appropriate types, it must hope for the best. The `vararg` facility is convenient but it should be used with care.

6.8 Store reuse

As we have seen on a number of occasions, store allocation in C requires care. Most list processing functions in this chapter allocate storage (the exceptions being `head`, `tail`, `length`, and `nth`). None of the functions deallocate or reuse storage. A program built using such greedy primitives, must, unless it is short, soon exhaust any available space. In the following, we give some general guidelines on how to approach allocation. To do this, we must make some assumptions about the context in which we are working.

The most common form of deallocation has been discussed before in Chapter 5. When a function deletes the last reference to a block of store, then this block of store must be deallocated. Using this policy we can adapt list_hi_ho on page 187. The reference hi to the cons-cell containing the H will cease to exist at the end of the function; therefore, the cons-cell containing the H must be deallocated. However, this cons-cell contains the last reference to the next cell in the list (the i). The whole list must be deallocated, for which we will make a function free_list.

The deallocation function free_list below works as follows. Every cell must be deallocated in turn, but once a cell has been deallocated, the pointer to it is a dangling reference (the cell may have been destroyed). Therefore, the next cell in the list must be read *before* a cell is deallocated:

```
void free_list( char_list xs ) {
  while( xs != NULL ) {
    char_list last = xs ; /* Remember which cell to free */
    xs = tail( xs ) ;     /* First advance xs in the list */
    free( last ) ;        /* Before freeing the head */
  }
}
```

The function free_list can now be called from list_hi_ho in order to deallocate the list hi:

```
char_list list_hi_ho( void ) {
  char_list hi = cons(  H , cons(  i , NULL ) ) ;
  char_list ho = cons(  H , cons(  o , NULL ) ) ;
  char_list hiho = append( hi, ho ) ;
  free_list( hi ) ;
  return hiho ;
}
```

Note that the list pointed to by ho should not be freed; append did not copy the tail part of the newly created list, but reused the ho part. Proper deallocation depends on the precise interface provided by functions. It is therefore vital to document not only the functionality of functions, but also how they manage the store. In the case of append, the function has two arguments: one of the arguments is copied (the first list), while *an extra reference* to the second list is being created. This extra reference is the reason why, on exit from list_hi_ho, there is still a reference in existence to the list pointed to by ho.

In the case of list_hi_ho, the number of references to all objects can be predicted while writing the program. Often this prediction is difficult or even impossible. When this is the case, the number of references has to be counted at run-time instead. Each data structure that is allocated has an extra field, called the *reference count*. The reference count is then maintained as follows:

- The reference count is initialised to 1.
- Each time a pointer to a cell is duplicated, the reference count is incremented by one.
- Each time a pointer to a cell is destroyed, the reference count is decremented by one. If the reference count reaches zero, the cell can be deallocated.

In principle, reference counting is a simple idea. Using reference counting, the function append should increment the reference count of the second argument and would leave the reference count of the first argument unchanged. The function list_hi_ho should decrement the reference counts on hi and ho prior to returning, which will cause the list hi to be deallocated because there is no other reference to the Hi list.

Note that list_hi_ho should actually increment the reference count of hiho prior to returning (because an extra copy, the return value, is made) and decrement it immediately afterward (because hiho will be destroyed upon return). These two cancel each other out and do not have to be implemented. This saves some increments and decrements that obfuscate the program code. However, because of these optimisations, it is difficult to make it work correctly: an increment or decrement is easily forgotten and such omissions wreak havoc on the data structures. Furthermore, there may be a significant overhead associated with the maintenance of the reference counts, both in terms of execution time and in terms of the extra space required to maintain the counts. In practice, reference counts are therefore used on data items that are large and allocated infrequently. Explicit deallocation is used for smaller units of data.

6.9 Summary

The following C constructs were introduced:

Recursive types Structures need to contain a pointer to a value of their own type in order to create a linked list. It is not legal to refer forward to a type in C, so instead, the struct must be named, this name is then used in combination with the keyword struct to refer to the type:

```
typedef struct s {
  /* other member declarations */
  struct s * n ;
} t ;
```

This defines a struct with, amongst others, a member called n, which is a pointer to a value of type struct s, which is its own type. The typedef defines t to be the type struct s.

NULL pointer The NULL pointer is a pointer that does not point to anything. It is used to signal error conditions (functions returning NULL), and it is used to signal the end of lists.

Memory allocation The function malloc allocates memory, like calloc, but memory is not initialised, and only one cell of the specified size is allocated. The function free releases memory.

Variable argument list It is possible to define functions with a variable number of arguments (such as printf).

This chapter discusses the implementation of a sequence as a list. The programming principles that we have seen in action revolve around the use of pointers to create efficient list manipulation functions.

- It is important to choose the right data structure. The main advantage of the list over the array is that lists are more easily extended and contracted. The main disadvantage of the list is that the cost of accessing an element is not uniform.

- Lists are a standard facility of functional programming languages. Type safe monomorphic lists can be built easily and efficiently in C.

- The standard repertoire of list processing functions such as `length`, `nth`, `append`, `filter`, and `map` can be built in C using the recursive definitions of these list operators. The extra argument technique from Chapter 4 to implement partially applied functions offers the flexibility and ease of higher order functions.

- The implementation of lists in C makes use of access functions since pattern matching is not provided. If used consistently, access functions help to provide clear and concise programs. Using access functions does not have to be costly, as compilers may automatically inline the body of such small functions.

- The C programmer must carefully consider the issue of allocating, reuse, and deallocating the storage used for list cells. Premature deallocation or reuse causes programs to go disastrously wrong. Late deallocation or reuse may cause programs to exceed the available space and crash.

- Programming by difference is a useful technique that requires the programmer to study differences between two functions and then to systematically implement these differences in more complicated versions of the functions.

- The C programmer has to take care that operations on lists do not inadvertently walk beyond the end of the list. It is wise to build checks into a program, so that if this happens, a sensible warning is produced. Functional programs are checked automatically.

- The list processing functions can be implemented efficiently by using tail recursion with an accumulating argument. The *open list* technique was introduced for accumulating a list. We have seen two implementations of this technique:

  - The first implementation maintains a pointer to the cell that has been allocated last. The next cell can then be linked via the appropriate field of the last cell at constant cost.

– The second implementation remembers, not a pointer to the last cell, but a pointer to the appropriate field of the last cell. This typical C programming style gives rise to an idiomatic and efficient implementation of open lists.

All C implementations of the standard list processing functions in this chapter offer a purely functional interface to their callers.

6.10 Further exercises

Exercise 6.18 Give the specification of reversing a sequence and implement an efficient C function `reverse` to operate on the type `char_list`.

Exercise 6.19 In this exercise, binary trees will be implemented.

(a) Give an equivalent definition in C of the following SML data type:

```
datatype tree = Branch of (tree * tree)
              | Leaf   of int ;
```

(b) Write a C function `mkLeaf` to create a leaf node of the `tree` type. The function `mkLeaf` should have the following prototype:

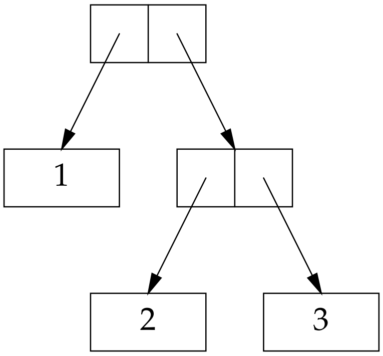
```
tree_ptr mkLeaf( int leaf ) ;
```

Also write a function `mkBranch` with the prototype below to create an interior node.

```
tree_ptr mkBranch( tree_ptr left, tree_ptr right ) ;
```

(c) Write a recursive C function to print a tree as follows:

- For each leaf, print its key value.
- For each branch, print its left branch and print its right branch, separated by a comma and enclosed in parentheses. As an example, the tree below should be printed as `(1,(2,3))`.



(d) A binary tree can be rotated by swapping the left and right fields at every interior node. For example:

```
rotate (Branch(Leaf(1),Leaf(2)))
      = Branch(Leaf(2),Leaf(1)) ;
rotate (Leaf(0))
      = Leaf(0) ;
```

First write a function `rotate` in SML. Then write a C function with exactly the same functionality as the SML function, that is, make sure that the original tree is not destroyed.

Exercise 6.20 It is interesting to try to simulate SML style pattern matching in C. In this exercise we are going to develop a general system to support this. Let us look again at the definition of a character list and a typical SML function, `head`, that uses pattern matching:

```
datatype char_list = Nil
                   | Cons of (char * char_list) ;

(* head : char_list -> char *)
fun head (Cons(x,xs)) = x ;
```

Firstly, the `head` function needs to analyse its argument so as to check that it is of the form `Cons(...,...)` and not of the form `Nil`. Secondly, the `head` function has to bind the value of the first component of the `Cons` constructor to the variable `x`. It then also binds the value of the second component to the variable `xs`, even though the latter is not used on the right hand side of the definition of `head`. Our task is to provide support in C for these two activities: case analysis and binding.

(a) Study the following two type definitions carefully and relate the various elements of the definitions to the two activities identified above.

```
typedef enum { Bind, Data } tree_tag ;

typedef struct tree_struct {
  tree_tag tag ;
  union {
    struct tree_struct ** bind ;   /* Bind */
    struct data_struct {           /* Data */
      char key ;                   /* Data */
      int size ;                   /* Data */
      struct tree_struct ** data ; /* Data */
    } comp ;                       /* Data */
  } alt ;
} * tree_ptr ;
```

(b) Write two functions `mkBind` and `mkData` to allocate the appropriate version of the `tree_struct` on the heap. Use the `vararg` facility so that the function takes an arbitrary number (possibly zero) of sub trees, and calloc to allocate sufficient memory to hold the pointers to the sub trees. The required prototypes of the functions are shown below.

```
tree_ptr mkBind( tree_ptr * b ) ;
tree_ptr mkData( char k, int s, ... ) ;
```

(c) Write a function match with the prototype given below:

```
bool match( tree_ptr pat, tree_ptr exp ) ;
```

The function should compare the two trees and return true if they match, false otherwise. Two trees pat and exp match if:

- Both have tag Data and if both have the same key and size. In addition, all subtrees must match.
- If the argument pat has tag Bind. In that case, the field bind should be set to remember the value of exp. You may assume that there are no nodes with tag Bind in the trees offered as the second argument to match.

(d) Write a main function to create some sample patterns and expressions to test your match function.

Chapter 7

Streams

The previous chapters discussed two data structures for storing sequences of data items: lists and arrays. A third kind of sequence is the stream. An example of a stream is the sequence of characters being typed by the user as input to a program, or conversely, the characters being printed on the screen.

In this chapter, streams are discussed, and they are compared with the two other sequences. A stream has two distinguishing features:

- A stream can *only* be accessed sequentially. That is, after accessing element x_i of the stream, only the next element x_{i+1} can be accessed. If x_i is needed later on, it must be remembered explicitly.
- The number of items in a stream is not known beforehand. At any moment, the stream might dry up.
- In C and SML streams work by side effects. That is, reading a character from a stream uses a function that returns a character (the result), and updates some state to record that this character has been read. Writing a character deposits the character on the stream, and updates some state to record this.

In a functional language, a stream is sometimes captured via a list abstraction. That is, a certain list does not have a predetermined value, but contains all characters that come from the stream. When the stream ends, the list is terminated (with `[]`). Using a list is slightly deceptive, as it 'remembers' all previous characters in the stream.

Most programming languages, including C and SML, offer a set of low level primitives that allow a program to open a certain stream, access the elements of the stream sequentially, and then finally close the stream. The elements cannot be accessed in an arbitrary order, as is the case with other sequences. Should the programmer wish to use the elements of a stream in a different order than sequentially, or should the programmer wish to use them more than once, the elements of the stream must be stored for later use. Streams can be large, so it is often impractical to keep the entire contents of a stream in the store. The main issue in this chapter is how to access a stream such that the least amount of store is needed to remember stream elements.

As an example of the use of streams, four programs will be developed. The first program counts the number of sentences in a stream. A straightforward version of this program remembers the entire stream. An optimised version of the program will remember just one element of the stream. The second program will calculate the average length of sentences in a stream. This program also comes in two versions, one that remembers the entire stream and the other that remembers only a single element. The third program counts the number of times a certain word appears in a stream. This program needs to remember a number of elements from the stream, where the number of elements is determined by the length of the word. The last program will sort the elements of a stream. This program must remember all elements of the stream.

7.1 Counting sentences: stream basics

We will now write a program to count the number of sentences in a text. The solution has two aspects. The first is how to define a sentence. We will assume that a sentence is a sequence of characters that ends with a full stop. The number of sentences will be taken as the number of full stop characters `.` in the text:

$$\begin{array}{rcl} \text{sentence\_count} & : & (\mathbb{N} \rightarrow A) \rightarrow \mathbb{N} \\ \text{sentence\_count}(s) & = & \#\{i \mapsto s(i) \mid i \in \text{domain}(s) \wedge s(i) = \texttt{.} \} \end{array} \qquad (7.1)$$

As a first solution, we can use two functions. One function captures the stream in a list, and another function counts the number of full stops in a list. For the second we use the SML function `full_stop_count` shown below. Please note that, whilst we used a monomorphic type `char_list` in Chapter 6, here we use the standard SML type `list` with the usual polymorphic operators, such as `length` and `filter`.

```
(* full_stop_count : char list -> int *)
fun full_stop_count cs
    = let
         fun is_full_stop c = c = "."
      in
         length (filter is_full_stop cs)
      end ;
```

The second aspect of the solution is to capture the input stream. In SML streams are accessed using two functions. The predicate `end_of_stream(stream)` determines whether the end of a stream has been reached. For as long as the predicate yields false (and there are thus elements to be accessed from the stream), the function `input(stream,n)` can be used to access the actual characters from the stream. The argument `stream` identifies the stream. The argument `n` specifies how many characters should be accessed. The function `input` has a side effect on the stream, in that it makes sure that a subsequent call to input will cause the next character(s) to be accessed. A complete SML function to transfer all characters from a stream into a list could be written as follows:

```
(* stream_to_list : instream -> char list *)
fun stream_to_list stream
    = if end_of_stream stream
         then []
         else input(stream,1) :: stream_to_list stream ;
```

Note that `stream_to_list` must rely on a side effect on the stream. This can be seen if we consider what would happen if `input` and `end_of_stream` had been pure functions. In this case, the stream would not change from one recursive call of `stream_to_list` to the next; therefore the test `end_of_stream` would always give the same result (either always true, or always false). In the case `end_of_stream` always yields false, `stream_to_list` would return the empty list, otherwise it would never terminate. Neither of these are the intended behaviour of `stream_to_list`, so it must do its useful work with a side effect.

The first solution to the problem of counting sentences in a text combines the two functions `stream_to_list` and `full_stop_count` as follows:

```
(* sentence_count : instream -> int *)
fun sentence_count stream
    = full_stop_count (stream_to_list stream) ;
```

The function `sentence_count` is elegant but inefficient: It remembers the entire contents of the stream, while it only needs a single element of the stream. The solution to this problem is to combine counting full stops with accessing stream elements. We discuss this optimisation in the next section, after having introduced the C equivalents of the functions `full_stop_count`, `stream_to_text`, and `sentence_count`.

Using the techniques of Chapter 6, we can develop the C version of `full_stop_count`. Here is the result:

```
int full_stop_count( char_list cs ) {
  int stops = 0 ;
  while( cs != NULL ) {
    if( head( cs ) ==  .  ) {
      stops++ ;
    }
    cs = tail( cs ) ;
  }
  return stops ;
}
```

A stream in C has the type `FILE *`. To access the elements of a stream, C offers the primitive `getc( stream )`. If there is at least one character available, `getc` will return that element. Otherwise it will return the integer `EOF` (which stands for **E**nd **O**f **F**ile). Note that `getc` results in an integer; a small subset of these integers denote legal character values that can be read. The other integers are used to signal special 'out of band' values for example, the end of the file. It is important to know that, when the result of `getc` is stored in a variable of type `char`, the value `EOF` might be lost, as it is *not* a character.

A function stream_to_list that uses getc to create a list from the standard input can now be implemented in C:

```
char_list stream_to_list( FILE * stream ) {
  int c = getc( stream ) ;
  if( c == EOF ) {
    return NULL ;
  } else {
    return cons( c, stream_to_list( stream ) ) ;
  }
}
```

The function stream_to_list processes one character at a time. This character is then stored as the head of the result list using the cons function. The rest of the input is read by a recursive invocation of stream_to_list. The stream_to_list function returns an empty list using the NULL pointer when it encounters the end of the input.

We can now give a naive, but elegant, C function sentence_count to count the number of full stops in a text:

```
int sentence_count( FILE * stream ) {
  return full_stop_count( stream_to_list( stream ) ) ;
}
```

The given C solution to the sentence count problem would work, but only for small streams. The solution is inefficient because it requires an amount of store proportional to the length of the stream. There are two reasons why this is the case. Firstly, the function stream_to_list uses an amount of stack space proportional to the length of the stream. Secondly, the entire stream is transferred into the store during the computation. We will solve both efficiency problems using the techniques that were developed in Chapters 3 and 6.

7.1.1 Efficiently transferring a stream to a list

The stream_to_list function will recurse over the input, so the stack size required for this program is proportional to the size of the input stream. This space efficiency problem can be solved using tail recursive functions and open lists in the same way as the efficiency problems of the copy and append functions were solved in Chapter 6. The body of stream_to_list is a conditional. To compare the function copy to stream_to_list, we must rewrite copy so that it also uses a conditional. Furthermore, the functions in Chapter 6 were all specialised to operate on the data type char_list. Here we use the standard SML type list. The two functions are:

```
fun stream_to_list stream        fun copy xs
    = if end_of_stream stream        = if xs = []
      then []                          then []
      else input(stream,1) ::          else head xs ::
           stream_to_list stream ;          copy (tail xs) ;
```

The two functions have the same structure. The main difference is that the list `xs` plays the same role in `copy` as the stream does in `stream_to_list`. In detail, the differences are:

The arguments. The function `copy` has an explicit argument representing the list being accessed. The function `stream_to_list` relies on the SML primitive `input` to remember at which point the current file is being read.

The termination condition. The test of 'end of input' for `copy` checks whether the end of the list has been reached using `xs = []`. The corresponding test `end_of_stream stream` in `stream_to_list` checks whether the end of the stream has been reached.

The next element. While the end of the input has not been reached, `copy` constructs a new list element using the next element of the input list `xs` while `stream_to_list` uses the character it has just accessed using `input(stream,1)`.

Side-effects. The `copy` function is pure, that is, it has no side effect. The `stream_to_list` function is impure, since it has a side effect on the stream.

Making changes to the open list version of `copy` that reflect the above differences yields the following efficient C version of `stream_to_list`. The changes have been annotated in the comments:

```
char_list stream_to_list( FILE * stream ) {      /* name  */
  char_list accu = NULL ;
  char_list last ;
  int c;                                     /* c declaration */
  if( (c = getc(stream) ) != EOF ) {       /* test on end */
    last = accu
         = cons( c, NULL ) ;                /* c not head.. */
    /* no statement here because getc side effects       */
    while( (c = getc(stream) ) != EOF ) {  /* test on end */
      last = last->list_tail
           = cons( c, NULL ) ;              /* c not head.. */
      /* no statement here because getc side effects     */
    }
  }
  return accu ;
}
```

Please note the use of assignments in expressions here to save the character just read for later use. This is idiomatic C and also quite clear.

Exercise 7.1 Rewrite `stream_to_list` without assignments in expressions and comment on the resulting duplication of code.

Exercise 7.2 Another implementation of `stream_to_list` would use pointers to pointers, as in the second form of `copy` and `append`. Implement `stream_to_list` using pointers to pointers.

The function stream_to_list efficiently transfers text from a stream into a list. This avoids the problem of the unbounded amount of stack space in our initial solution to the sentence count problem.

7.1.2 Avoiding the intermediate list

The SML and C solutions of the sentence count problem transfer the entire contents of the stream into the store before the function full_stop_count can count the number of full stops. This is not necessary: the elements of the stream could be read step by step, while counting the full stops at the same time. So far, our efforts to make stream_to_list efficient are necessary, but not sufficient. For as long as one of the functions involved in the solution causes inefficiency, the entire solution will be inefficient.

Here is an efficient version of sentence_count. It uses an auxiliary function count to do the real work. This SML function shows how the two processes of reading and counting have been merged:

```
(* sentence_count : instream -> int *)
fun sentence_count stream
    = let
         fun count stops
             = if end_of_stream stream
                  then stops
                  else if input (stream, 1) = "."
                          then count (stops+1)
                          else count stops
      in
         count 0
      end ;
```

The auxiliary function count is efficient because:

- count is tail-recursive.
- count does not create intermediate list structure.

Exercise ⋆ 7.3 Prove that the efficient version of sentence_count is equivalent to the inefficient version of the previous section.

The efficient C version of sentence_count is:

```
int sentence_count( FILE * stream ) {
  int stops = 0 ;
  while( true ) {
    int c = getc( stream ) ;
    if( c == EOF ) {
      return stops ;
    } else if( c ==  .  ) {
      stops++ ;
      }
    }
  }
```

Removing the need for the intermediate list is not only more efficient in space, but it also makes the program run faster.

7.1.3 IO in C: opening files as streams

A C program may operate on several streams at the same time. Three predefined streams are always available: `stdin` for accessing the standard input stream, `stdout` for accessing the standard output stream, and `stderr` which is a stream for writing error messages.

Further input and output streams can be made available to the program using the function `fopen`, which creates a stream so that it reads from, or writes to a file. This function has two parameters, the first parameter is the name of the file to be opened, the second parameter is the mode. The two most frequently used modes are `"r"` (which results in a stream from which input can be read) and `"w"` (which results in a stream to which output can be written). Other modes, amongst others for appending to files, are specified in the C manual [7]. The function `fopen` will return a stream (of type `FILE *`, defined in `stdio.h`), if the file cannot be opened, it will return the value NULL.

When the program is finished with a stream, the function `fclose` should be called to close the stream.

We have seen two functions operating on streams so far: `printf` which printed to the standard output and `getc` which reads a character from a specified stream. They are part of two families of functions for reading data from streams, and writing data to streams.

Input

There are three more functions to read from an input stream: `getchar`, `scanf`, and `fscanf`. The function `getchar()`, reads a character from the standard input stream, it is the same as `getc( stdin )`. The function `scanf` reads a number of data items of various types from the standard input stream and stores the values of these data items (`scanf` stands for 'scan formatted'). The function `scanf` can be used to read strings, integers, floating point numbers, and characters. The first argument of `scanf` specifies what needs to be read; all other arguments specify where the data should be stored. As an example of scanf, consider the following call:

```
int i, n ;
char c ;
double d ;
n = scanf("%d %c %lf", &i, &c, &d ) ;
```

This section of code will read three data items: an integer (because of the `%d` in the format string), which is stored in `i`; a character (denoted by the `%c` in the format string), which will be stored in the variable c; and a floating point number, which will be stored in d, indicated by the %lf (more about this %-sequence later on). The return value of scanf is the number of items that were successfully read. Suppose the input stream contains the following characters:

```
123 q 3.14
```

The variable i will receive the value 123, c will receive the value q , and d will receive the value 3.14. The function call returns 3 to the variable n because all items were read correctly. Suppose the stream had contained instead:

```
123/*3.14
```

Now the scanf would have read the 123 into i, read / into c, and returned 2, as the asterisk is not (part of) a valid floating point number. When scanf fails, the next use of the stream will start at the first character that did not match, the asterisk in this example.

Any character in the format string of scanf that does not belong to a %-sequence must be matched exactly to characters in the input stream. The exception to this rule is that spaces in the format string match any sequence of white space on the input stream. Consider the following call as an example:

```
int i, n ;
char c ;
n = scanf( "Monkey: %d %c", &i, &c ) ;
```

This will give the value 2 to the variable n if the input is:

```
Monkey: 13       q
```

It will return 0 to n if the input is:

```
Money: 100000
```

In the latter case, the characters up to Mon will have been read, the next character on the stream will be the e from Money.

The function scanf returns any number of results to its caller through its pointer arguments. It is convenient to do so in C, but, as we have already seen, it is a type-unsafe mechanism: the compiler cannot verify that the pointers passed to scanf are pointers of the right type. The following call will be perfectly in order according to the C compiler:

```
int i, n ;
char c ;
double d ;
n = scanf("%lf %d %c", &i, &c, &d ) ;
```

However, scanf will attempt to store a floating point number in the integer variable i, an integer in the character variable c, and a character in the double variable d. Therefore, this program will execute with undefined results.

The way floating point numbers are accessed is a source of problems. A %f-sequence reads a *single* precision floating point number, while %lf reads a *double* precision floating point number. Until now we have ignored the presence of single precision floating point numbers of type float, but scanf must know the difference. Writing a single precision floating point into the store of a double precision number does not give the desired result. Another common error using scanf is to forget the & in front of the arguments after the format string, causing the value to be passed instead of a pointer, most likely resulting in the function scanf crashing.

The function scanf always accesses the standard input stream stdin. The function fscanf has an extra argument that specifies the stream to be accessed. Similar to getchar and getc, scanf(...) is identical to fscanf(stdin, ...). An example of its use is given after the description of the output functions.

Output

Similar to the family of input functions, getchar, getc, scanf, and fscanf, there is a family of output functions: putchar, putc, printf and fprintf. The companion of getchar to put characters on the output stream stdout is called putchar. Here is a sequence of six statements that is equivalent to printf("Hello\n"):

```
putchar(  H  ) ; putchar(  e  ) ; putchar(  l  ) ;
putchar(  l  ) ; putchar(  o  ) ; putchar(  \n  ) ;
```

The function call putchar(c) is equivalent to a call to printf("%c", c). The function putc(c, stream) allows a character c to be output to the named stream. Thus putchar(c) is equivalent to putc(c, stdout). Finally, the function fprintf has an extra argument that specifies to which stream to write: printf(...) is identical to fprintf(stdout, ...).

An Input/Output Example

The following program fragment reads an integer and a double precision floating point number from the file “input.me” and writes the sum to the file “result”:

```
int i, n ;
double d ;
FILE *in  = fopen( "input.me", "r" ) ;
if( in != NULL ) {
  FILE *out = fopen( "result", "w" ) ;
  if( out != NULL ) {
    n = fscanf( in, "%d %lf", &i, &d ) ;
    if( n == 2 ) {
      fprintf( out, "%f\n", d+i ) ;
    } else {
      fprintf( stderr, "Wrong format in input\n" ) ;
    }
    fclose( out ) ;
  }
  fclose( in ) ;
}
```

Exercise 7.4 Give SML and C versions of a function `list_to_stream` to transfer the contents of a list of characters to a stream.

7.2 Mean sentence length: how to avoid state

The previous problem counted the number of sentences in a text. A slightly more complicated problem is not only to count the number of sentences, but to calculate the mean sentence length:

$$\begin{aligned}\text{mean\_sentence\_length} &: (\mathbb{N} \rightarrow \mathbb{A}) \rightarrow \mathbb{N}\\ \text{mean\_sentence\_length}(s) &= (\#s) \text{ div } \text{sentence\_count}(s) \qquad (7.2)\end{aligned}$$

As we have already developed an efficient function `sentence_count` to count the number of sentences, all we have to do is write a function `character_count` to count the number of characters in a stream. Then we may divide the number of characters by the number of sentences. If our function `sentence_count` is a good building block, we should be able to reuse it for this purpose.

Here is an efficient function `character_count`, which is basically a simplified version of `sentence_count`:

```
(* character_count : instream -> int *)
fun character_count stream
    = let
         fun count chars
             = if end_of_stream stream
                  then chars
                  else let
                          val c = input (stream, 1)
                       in
                          count (chars+1)
                       end
      in
         count 0
      end ;
```

The mean sentence length would be given by the SML function below:

```
(* mean_sentence_length : instream -> int *)
fun mean_sentence_length stream
    = (character_count stream) div
      (sentence_count stream) ;
```

Unfortunately, executing this program will not produce the expected result. Either the function `mean_sentence_length` will produce the answer 0 or it will fail with a 'divide by zero' exception. How can this be? The two building blocks `sentence_count` and `character_count` are not faulty. The reason is that the function `input`, which is used by both, has a side effect and modifies the stream. Depending on the order in which the two operands of the `div` operator are evaluated, either character_count accesses the entire stream, so that when sentence_count is evaluated, it finds the stream empty, or vice versa. This shows the danger of using functions that have side effect: they make it difficult to create good building blocks. It depends on the circumstances whether a function that relies on side effects has the desired behaviour or not. The C versions of sentence_count and character_count also modify the stream as a side effect, so there is no point in trying to pursue development on the present basis.

The mean sentence length problem could be solved by transferring the entire stream into a list and then to count the characters and the sentences. Such a solution was dismissed earlier because of the inherent inefficiency and also the impossibility of processing arbitrary large streams. A better solution is to combine the activities of sentence_count and character_count into a single function that accesses the elements of the stream just once.

Such a combined function can be developed, since the structures of sentence_count and character_count are so similar. The SML functions can be merged, whilst taking care not to duplicate the elements they have in common:

```
(* mean_sentence_length : instream -> int *)
fun mean_sentence_length stream
    = let
         fun count chars stops
             = if end_of_stream stream
                  then chars div stops
                  else if input (stream, 1) = "."
                          then count (chars+1) (stops+1)
                          else count (chars+1) stops
      in
         count 0 0
      end ;
```

The auxiliary function count inspects each character of the input stream in turn and counts the number of full stops and characters in parallel. This means that the input stream does not have to be remembered, and the implementation can discard each character of the stream after it has been inspected.

The version of mean_sentence_length above can be transformed into an efficient C function without difficulty:

```
int mean_sentence_length( FILE * stream ) {
  int chars = 0 ;
  int stops = 0 ;
  int c ;
  while( true ) {
    c = getc( stream ) ;
    if( c == EOF ) {
      return chars / stops ;
    } else if( c ==  .  ) {
      stops++ ;
    }
```

```
      chars++ ;
    }
  }
```

This is an efficient function to calculate the mean sentence length. Many functions that consume a stream can be written using a while loop that iterates over the characters of the input stream, accumulating all the required information.

7.3 Counting words: how to limit the size of the state

The programs developed for the previous two examples operated on the stream on a per character basis. Programs often act on larger units of data. An example of such a problem is the word-count problem. Given a text (for example the text of this book), we may wish to count how often a certain word (for example, the word "cucumber") occurs in the text (the answer is 11 times).

The number of occurrences of the word w in the text t is given by the function below, where both the word and the text are represented by a sequence of characters:

$$\begin{array}{rcl} \text{word\_count} & : & (\mathbb{N} \to \alpha \times \mathbb{N} \to \alpha) \to \mathbb{N} \\ \text{word\_count}(w, t) & = & \#\{(t_1, t_2) \mid t = t_1 \frown w \frown t_2\} \end{array} \qquad (7.3)$$

An algorithm to count all the occurrences of the word `w` would try to match it to each position in the text `t`. Using lists to represent both the word and the text, successful matches can be counted as follows in SML:

```
(* word_count : char list -> char list -> int *)
fun word_count ws []      = 0
  | word_count ws (t::ts) = if match ws (t::ts)
                               then 1 + word_count ws ts
                               else     word_count ws ts ;
```

The `word_count` function requires a subsidiary function `match` to check whether a word matches the text at a particular position. The `match` function compares the characters from the word and the text. It yields false if either the text contains too few symbols or if a mismatch is found:

```
(* match : char list -> char list -> bool *)
fun match []      t       = true
  | match (w::ws) []      = false
  | match (w::ws) (t::ts) = w = (t:char) andalso match ws ts ;
```

Exercise ⋆ 7.5 Given a word w, a text t, and (7.3). Prove that: word_count$(w, t) =$ `word_count w t`

As a first step, we use the naive approach: transfer the contents of a stream into a list and then count the number of occurrences of the word "cucumber":

```
(* main : instream -> int *)
```

```
fun main stream =
  let
     val word = explode "cucumber"
     val text = stream_to_list stream
  in
     word_count word text
  end ;
```

Exercise 7.6 Rewrite `word_count` above as a tail recursive function using an accumulating argument and translate the result into a loop based C implementation.

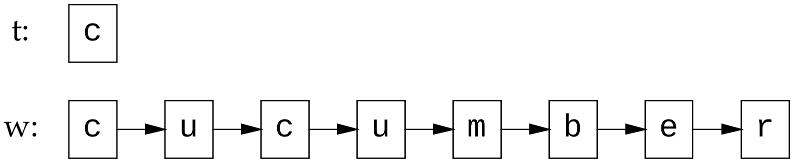
Exercise 7.7 Give an efficient C implementation of `match`.

Exercise 7.8 Write a `main` program, using `word_count` and `stream_to_list`, to count the number of occurrences of the word "cucumber" on the `stdin` stream:

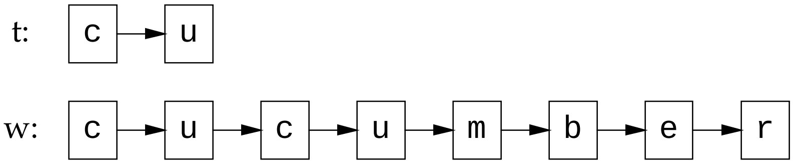
7.3.1 Using a sliding queue

As we have seen in the previous sections, first transferring the contents of a stream into a list and then manipulating the list is not ideal. For the word count problem, we need a *queue* to buffer elements from the input stream: each time a new character of the stream is needed, it is entered to the rear of the queue. Each time a character has ceased to be useful, it is deleted from the front of the queue. The net effect of this is that the queue appears to advance over the text and that the word shifts along with the queue.

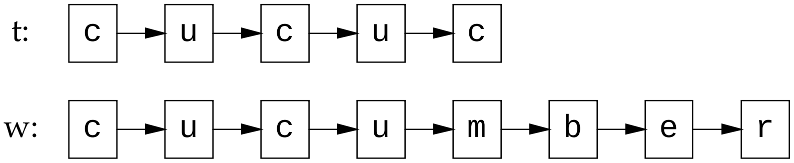
To illustrate this process suppose that the text contains the characters "cucucumbers" and that we have seen that the first character of the text matches the first character of the word "cucumber". This situation is shown graphically below. Here the portion of the text that has been accessed so far is labelled `t`, and the word is labelled `w`. The front of the queue is to the left of the picture, and the rear of the queue is to the right.



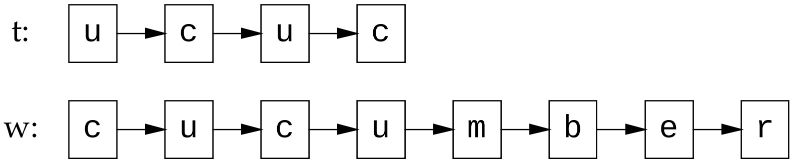
The word and the text are shown with corresponding matching characters vertically aligned. Now the second character of the text must be accessed for it to be compared to the second character of the word. The character `u` enters the queue at the rear (right).



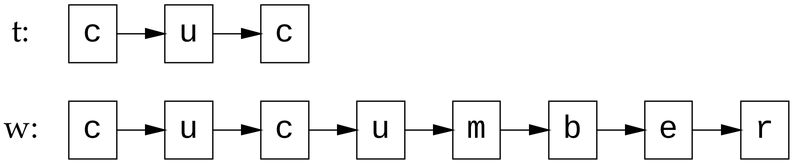
After accessing another three characters a mismatch occurs, because the text reads "cucuc…" and the word reads "cucum…". This causes the `match` function to return false to `word_count`. The queue of elements from the text is now in the following state:



The function `count` makes the next step; it advances the position of the queue in an attempt to match "u…" with "c…". The character `c` at the front of the queue has now left.



The match now fails immediately. The queue is advanced yet again, so that now also the character `u` at the front of the queue disappears.



Now we see the benefits of using a queue sliding over the text. The matching of the characters `c` , `u` and `c` can be done immediately; there is no need to access new elements from the stream because they have been remembered.

7.3.2 Implementing the sliding queue in SML

An implementation of the sliding queue should satisfy the following requirements:

- New elements should be appended to the rear of the queue.

- Accessing the queue should be supported efficiently.
- It should be possible to discard elements from the front of the queue as soon as they are no longer needed.

The sliding queue can be implemented on the basis of a list. We will first study this in SML and then give an optimised implementation in C. The SML datatype `queue` that we use for the queue is a tuple consisting of an input stream and a list:

```
datatype  a queue = Queue of (instream *  a list) ;
```

The actual elements that are part of the queue are stored in the list. We are aiming to always keep a certain number of elements in the list. In the context of the word count problem, we know in advance that we will be needing as many elements as there are elements in the word that we are counting. To support the transfer of elements from the text, we need to remember with which stream we are dealing.

Initially, our program will have to create a queue with a number of characters from the stream entered into the queue. Here is an SML function to do this:

```
(* create : instream -> int -> char queue *)
fun create stream n
    = Queue (stream, stream_to_list stream n) ;
```

The function `create` tries to access `n` characters from the stream and stores these in the list. We are using here a modified version of `stream_to_list` which reads no more than `n` characters. The identity of the stream must be remembered for future use.

Exercise 7.9 Give an efficient C version of `stream_to_list` which reads no more than a given number `n` of elements.

To fetch the contents of the queue, we define the SML function `fetch`:

```
(* fetch :  a queue ->  a list *)
fun fetch (Queue (stream, list)) = list ;
```

When advancing the queue we must make sure that a new character is automatically entered when an old character is removed. New characters must be transferred from the stream involved. Here is the SML function `advance` taking care of this:

```
(* advance : char queue -> char queue *)
fun advance (Queue (stream, list))
    = if end_of_stream stream
         then Queue (stream, tail list)
         else Queue (stream, tail list @
                                        [input (stream, 1)] ) ;
```

Appending a newly accessed stream element to the rear of the queue is relatively expensive as it requires the use of the standard SML append operator `@`. In the optimised C version of this solution, we will use the open list technique rather than an append operation.

Finally, we need a predicate is_empty to signal when no further elements are available:

```
(* is_empty :  a queue -> bool *)
fun is_empty (Queue (stream, list)) = list = [] ;
```

To use these four primitives, the function word_count should be rewritten to operate on a sliding queue, rather than on a list.

```
(* word_count : char list -> char queue -> int *)
fun word_count ws ts
    = if is_empty ts
         then 0
         else if match ws (fetch ts)
                 then 1 + word_count ws (advance ts)
                 else     word_count ws (advance ts) ;
```

The function match does not need to be altered, as the result of applying fetch to the queue is a list. To complete the SML queue based solution of the word count problem, we give a main function:

```
(* main : instream -> int *)
fun main stream =
  let
     val word = explode "cucumber"
     val text = create stream (length word)
  in
     word_count word text
  end ;
```

7.3.3 Implementing the sliding queue in C

The sliding queue implementation in SML uses the append operator to enter new elements from the stream into the queue. We have seen that this is an inefficient operation. The same operation can be implemented efficiently if we use open lists. To develop a C data structure and a set of functions that implement the sliding queue with open lists, we need to keep track of the end of the list for the queue operations. Therefore, we define the equivalent of our SML type queue with a provision for keeping track of the beginning as well as the end of the list.

```
typedef struct queue_struct {
  FILE      * queue_stream ;
  char_list queue_first ;
  char_list queue_last ;
} * char_queue ;
```

As in SML, our C program will have to be able to create an initial queue with a number n of elements already entered into the list:

```
#define queue_struct_size sizeof( struct queue_struct )
```

```
char_queue create( FILE * stream, int n ) {
  char_queue q   = malloc( queue_struct_size ) ;
  if( q == NULL ) {
    printf( "create: no space\n" ) ;
    abort( ) ;
  }
  q->queue_stream= stream ;
  q->queue_first = stream_to_list( stream, n ) ;
  q->queue_last  = find_last( q->queue_first ) ;
  return q ;
}
```

The code of `create` is straightforward, except that we need to keep track of the last element of the list. There are two possible ways of implementing this. Firstly, we could modify `stream_to_list` so that it delivers the required pointer. This should not be difficult as the open list version of `stream_to_list` already keeps track of this information. The disadvantage is that `stream_to_list` would then have to return two, rather than just one, pointers, and this is not conveniently done in C.

The second way of getting the pointer to the last element of the list is simply by writing a new function `find_last` to do that. This is the alternative that we are favouring here. It has the disadvantage that the list is created first and then traversed just to find the end. However, since we may expect this list to be short, this should not be a problem. Furthermore, the extra traversal only happens once, during initialisation.

```
char_list find_last( char_list current ) {
  char_list previous = NULL ;
  while( current != NULL ) {
    previous = current ;
    current  = tail( current ) ;
  }
  return previous ;
}
```

The function `find_last` uses two ‘chasing’ pointers: `current` points at the current cons cell and `previous` at the one before. As soon as `current` reaches the end of the list (the `NULL` pointer), `previous` will be pointing at the last cons cell of the list. The function `find_last` returns the `NULL` pointer if the list is empty.

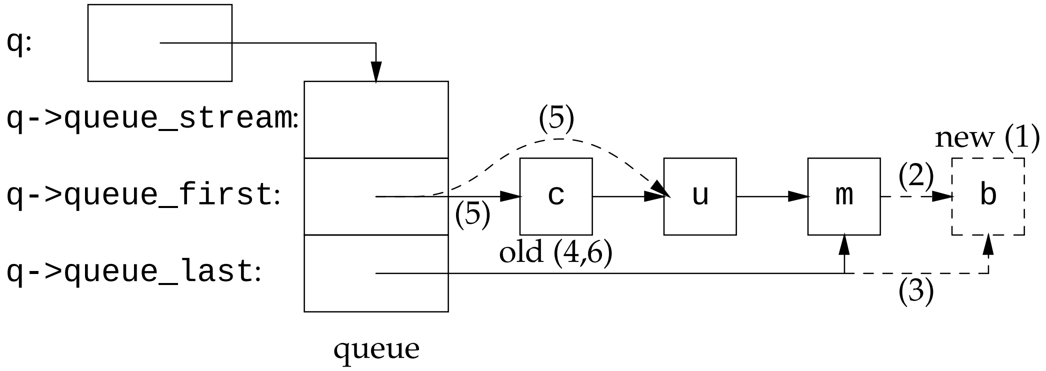
The function to fetch the contents of the queue is implemented in C as follows:

```
char_list fetch( char_queue q ) {
  return q->queue_first ;
}
```

To advance the queue, we will need to use the pointer to the last cell of the list, so that the append operation can be supported through the open list technique. To make `advance` efficient, we will write it to modify the old queue, hence having a side effect. To remind us of this choice, the return type of `advance` is `void`.

```
void advance( char_queue q ) {
    char_list old, new;
    int c = getc( q->queue_stream ) ;
    if( c != EOF ) {
      new = cons( c, NULL ) ;                        /* 1 */
      q->queue_last->list_tail = new ;               /* 2 */
      q->queue_last = new ;                          /* 3 */
    }
    old = q->queue_first ;                           /* 4 */
    q->queue_first = q->queue_first->list_tail ;   /* 5 */
    free( old ) ;                                    /* 6 */
  }
```

The working of `advance` is shown graphically below. The state of the queue as it exists when `advance` is called is drawn as solid lines. We are about to add the character `b` to the queue and to remove the character `c` . The dashed lines represent the changes made to the bookkeeping of the queue. The new element is put into a singleton list `new` (1). This list is then connected to the tail of the last cell of the queue (2). The `queue_last` field is redirected to point at the new cell (3). Removing the character `c` from the queue involves redirecting the `queue_first` field to point at the cell holding the letter `u` (5). The space occupied by the old cell must be freed, which requires us to keep a pointer `old` to the cell (4) and when all modifications have been done to free it (6).



Here is the predicate `is_empty` in C:

```
  bool is_empty( char_queue q ) {
    return q->queue_first == NULL ;
  }
```

Finally, `word_count` needs to be rewritten to use the queue instead of the list:

```
  int word_count( char_list ws, char_queue ts ) {
    int accu = 0 ;
    while( ! is_empty( ts ) ) {
      if( match( ws, fetch( ts ) ) ) {
        accu++ ;
      }
      advance( ts ) ;
    }
    return accu ;
  }
```

Exercise 7.10 Implement a main program to initialise the queue and count the number of occurrences of some word in a text.

7.3.4 Counting words using arrays

The process of creating a queue from a stream of characters as shown in the previous section is effective, but its efficiency could still be improved. For each each iteration of `word_count`, a list cell has to be allocated, and another cell has to be freed. Allocation of cells is usually expensive, and if performance is important, it is better avoided. Another data structure might be more appropriate for buffering the input stream, the array. Firstly, an array is allocated all at once rather than piecemeal, as is the case for the cells of a list. Secondly, an array offers more efficient access to arbitrary elements than a list. Below, we will show how arrays can be used to implement the word count problem.

Let us begin by revising the queue data type to use an array instead of a list. The SML version of the data type is shown below. The actual elements that are part of the queue are now stored in an array. We aim to always keep a certain number of elements in the queue. This is an important consideration as arrays are not easily extended. If we can allocate an array of the right size at the beginning, the array based queue will never have to do any storage allocation or deallocation. We cannot guarantee that all elements of the array will always contain useful information, so we maintain the index of the last element that is actually valid. We could have chosen to maintain a count of the valid elements instead. This turns out to be less convenient, as we will see.

```
datatype  a queue = Queue of (instream * int *  a array) ;
```

Here is the equivalent data structure in C:

```
typedef struct queue_struct {
  FILE * queue_stream ;
  int  queue_valid ;
  char * queue_array ;
} * char_queue ;
```

To create the queue, we use the `stream_to_list` function to transfer the first `n` elements of the stream to a list. Then we use the function `list_to_array` from Chapter 6 to create the array. As the function `create` is only used once (during initialisation), it should not matter that it creates a small intermediate list structure. It is important to make functions efficient that are often used. Spending effort on functions that are rarely used does not pay. This tradeoff is often referred to as the 90%/10% rule: a program will often spend 90% of its time in only 10% of its code. Therefore spending effort to optimise this 10% is worthwhile. Optimising the remaining 90% is rarely worthwhile.

```
(* create : instream -> int -> char queue *)
fun create stream n
    = let
         val list  = stream_to_list stream n
         val valid = length list - 1
         val array = list_to_array list
      in
         Queue (stream, valid, array)
      end ;
```

In C, the same function is:

```
#define queue_struct_size sizeof( struct queue_struct )

char_queue create( FILE * stream, int n ) {
  char_list list  = stream_to_list( stream, n ) ;
  char_queue q    = malloc( queue_struct_size ) ;
  if( q == NULL ) {
    printf( "create: no space\n" ) ;
    abort( ) ;
  }
  q->queue_stream = stream ;
  q->queue_valid  = length( list )-1 ;
  q->queue_array  = list_to_array( list ) ;
  return q ;
}
```

Fetching the contents of the queue is straightforward:

```
(* fetch :  a queue ->  a array *)
fun fetch (Queue (stream, valid, array)) = array ;
```

The C version of `fetch` is:

```
char * fetch( char_queue q ) {
  return q->queue_array ;
}
```

Exercise 7.11 We might need an extra function `valid` to tell us how many valid elements there are in the queue. Give these functions in C and SML.

Advancing the queue with an array based implementation is now slightly complicated, as we will need to shift the contents of the array. Here is the SML version of `advance`:

```
(* advance : char queue -> char queue *)
fun advance (Queue (stream, valid, array))
    = let
         fun shift i = if i < valid
```

```
                         then sub (array, i+1)
                         else input (stream, 1)
      in
         if end_of_stream stream
            then Queue (stream, valid-1, tabulate (valid, shift))
            else Queue (stream, valid, tabulate (valid+1, shift))
      end ;
```

Here we see that maintaining the index of the last valid element of the array is indeed convenient. If a counter had been used instead, several more increments and decrements would have to be programmed. The C version of the shifting `advance` is:

```
void advance( char_queue q ) {
  int c = getc( q->queue_stream ) ;
  int i ;
  for (i = 0; i < q->queue_valid; i++) {
    q->queue_array[i] = q->queue_array[i+1] ;
  }
  if( c == EOF ) {
    q->queue_valid -- ;
  } else {
    q->queue_array[q->queue_valid] = c ;
  }
}
```

The function `advance` needs to make as many steps as there are characters in the array. Shifting the array is an expensive operation if the array is large. A more efficient but more complicated version of `wordcount` is discussed in Exercise 7.13.

The predicate to signal that no further elements are available requires some care. As the `valid` component of the queue data type maintains the index of the last valid element, a value of `0` for `queue_valid` means that one element of the array is still available. Thus a value of `-1` signals that the queue is empty. Here is the SML version of the predicate:

```
(* is_empty :  a queue -> bool *)
fun is_empty (Queue (stream, valid, array)) = valid = ~1 ;
```

In C the code is:

```
bool is_empty( char_queue q ) {
  return q->queue_valid == -1 ;
}
```

Exercise 7.12 Rewrite the SML and C versions of `word_count` and `match` so that they become suitable for array based queues.

Exercise 7.13 The functions above shift all the elements in the array one position to the left on every advance. This operation is increasingly expensive for long words. An alternative is to use a *cyclic buffer*, where the text is 'wrapped around'. The `advance` operation overwrites the oldest character in the buffer, and the beginning of the buffer is defined to start at the next character. If we denote the start of the buffer b to be i, then the contents is $b(i), b(i+1), \ldots b(n-1), b(0), b(1), \ldots b(i-1)$. The next element to be overwritten is $b(i)$, whereupon i is incremented. Devise a solution that uses a cyclic buffer.

7.4 Quicksort

The final example of this chapter discusses the problem of sorting the contents of a stream. The problem here is that it is impossible to sort a stream efficiently without having access to its entire contents. This time, no optimisations apply that only keep part of the stream in the store. The problem of efficient sorting is by itself interesting and worthy of study.

To sort data, it should be possible to compare two arbitrary data elements and to decide which of the two should come first in a sorted sequence. When given a sequence s of length $n+1$ and an operation $\leq$ defined on the elements of the sequence, the sorted sequence sort(s) is:

$$\begin{array}{rcl} \text{sort} & : & (\mathbb{N} \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \\ \text{sort}(s) & = & \{0 \mapsto s(i_0), \ldots n \mapsto s(i_n)\} \\ & & \text{where } \{i_0, \ldots i_n\} \text{ is a permutation of } \{0, \ldots n\} \\ & & \text{and } s(i_0) \leq s(i_1) \leq \ldots s(i_n) \end{array}$$

An elegant divide and conquer algorithm to sort a sequence is quicksort [3]. The recursive specification of quicksort on sequences is given below. Here we are using the filter function as defined in Exercise 6.6. The predicates are written as sections; for example $(\cdot < p)$, when applied to some value x, is the same as $x < p$.

$$\begin{array}{rcl} \text{qsort} & : & (\mathbb{N} \rightarrow \alpha) \rightarrow (\mathbb{N} \rightarrow \alpha) \\ \text{qsort}(s) & = & \begin{cases} \exists p \in \text{range}(s). & \\ \text{qsort}(\text{filter}(\cdot < p, s)) & \\ \qquad {}^\frown\text{filter}(\cdot = p, s) {}^\frown & \\ \text{qsort}(\text{filter}(\cdot > p, s)), & \text{if } \#s > 0 \\ s, & \text{otherwise} \end{cases} \end{array}$$

Some element p of the sequence is used as a 'pivot'. The three filters partition the elements less than the pivot, those equal to the pivot (of which there is at least one), and the elements which are greater than the pivot into three separate sets. The elements of the first and third partitions are sorted recursively. The two sorted partitions are finally joined with the pivot partition to form the result.

Quicksort can be implemented using both lists and arrays. The list based implementation is elegant, but not efficient. The array based implementation is complicated, but more efficient. Both implementations will be discussed in the following sections.

7.4.1 Quicksort on the basis of lists

The specification of quicksort uses filters which can be translated directly into SML:

```
(* qsort : char list -> char list *)
fun qsort []      = []
  | qsort (p::xs) = let
                       fun less_eq x = x <= (p:char)
                       fun greater x = x >  (p:char)
                    in
                       qsort (filter less_eq xs)
                           @ [p] @
                       qsort (filter greater xs)
                    end ;
```

Local function definitions are required to capture the value of the pivot `p` for the benefit of the comparison operators (See Section 6.4.2).

The C version of `qsort` follows the SML version closely. The extra argument version of `filter` is used to pass the pivot `p` to the comparison operators <= and >.

```
char_list qsort( char_list p_xs ) {
  if( p_xs == NULL) {
    return NULL ;
  } else {
    char      p  = head( p_xs ) ;
    char_list xs = tail( p_xs ) ;
    char_list ls = extra_filter( less_eq, &p, xs ) ;
    char_list gs = extra_filter( greater, &p, xs ) ;
    return append( qsort( ls ),
                   append( cons( p, NULL ),
                           qsort( gs ) ) ) ;
  }
}
```

We cannot define local functions in C so the auxiliary functions `less_eq` and `greater` are defined as separate functions:

```
bool less_eq( void * arg,  bool greater( void * arg,
              char x ) {                   char x ) {
  char * p = arg ;           char * p = arg ;
  return x <= * p ;          return x > * p ;
}                          }
```

The C version of `qsort` above is incomplete: the functions `extra_filter`, `cons`, and `append` all allocate cells, but the store is not deallocated. Therefore, this program needs to be amended so that the temporary lists are deallocated.

Exercise 7.14 Amend the code for quicksort so that redundant lists are deallocated. Assume that `append( x, y )` copies the list `x`, `cons` creates one cell, and `extra_filter` creates a whole fresh list.

Exercise 7.15 Write a function `main` in both SML and in C to call `qsort` with the following short list of characters:

$$E\ ,\ C\ ,\ F\ ,\ B\ ,\ A\ ,\ C\ ,\ G$$

The list version of quicksort is elegant and compact, and it was not particularly difficult to derive. Unfortunately, it is not efficient because of the large number of intermediate lists it creates. There are two sources of inefficiency. Firstly, for each invocation of `qsort`, the input list is traversed twice, creating two separate partitions in the form of separate lists. Each element (except the first) of the input list will end up in one of the two partitions. Thus each call to `qsort` effectively copies its entire input list during the partitioning phase. Secondly, the calls to `append` create copies of the lists as produced by the recursive calls to `qsort`.

In the next section we turn to the array version of `qsort` with a view of avoiding all copies of intermediate data structures. The price that has to be paid for this improved efficiency is considerable complication in partitioning the input data. An unexpected advantage is that the `append` operation becomes redundant when using an array.

7.4.2 Quicksort on the basis of arrays

Arrays and lists are both representations of sequences. The difference between the two representations is in the cost of the operations that can be applied to the sequence. The quicksort algorithm partitions the input sequence into two separate sequences. Implementing this efficiently using lists is difficult, since it requires changing the linkage of the cells of the list. Moving data around in an array is easier.

Consider, as an example, the following operation of swapping two elements of an array. The SML version accesses the elements of the array `data` at positions `i` and `j` and creates a new array with the values at positions `i` and `j` exchanged:

```
(* swap :  a array -> int -> int ->  a array *)
fun swap data i j
    = let
         val data_i = sub (data,  i)
         val data_j = sub (data,  j)
         val data   = upd (data,  j, data_i)
         val data   = upd (data , i, data_j)
      in
         data
      end ;
```

In a purely functional language, accessing an element of an array has a time complexity of $\mathcal{O}(1)$ and updating an element is an $\mathcal{O}(n)$ operation. In C, both accessing and updating an array element is $\mathcal{O}(1)$, because the array can be overwritten. An efficient C version will access the old values at i and j and update them as follows:

```
void swap( char data[], int i, int j ) {
  char data_i = data[i] ;
  char data_j = data[j] ;
  data[i] = data_j ;
  data[j] = data_i ;
}
```

The function swap has a side effect on the array data. The void return type advertises that swap does its work by a side effect.

Let us now consider how an array based version of qsort could be developed. Firstly, the input data is needed in the form of an array. Secondly, the array must be partitioned such that one partition contains the array elements not exceeding the pivot and another partition contains the elements greater than the pivot. Elements equal to the pivot will occupy a partition of their own. Thus, an important element of the solution is to maintain various indices in the array, in order to keep tabs on the partitions involved.

Each call to qsort works on a section of the array, delineated by two indices l and r (for *left* and *right*). The function qsort will leave all other array elements undisturbed. The partitions within the range l to r will be further delineated using index variables j and i. The elements less than or equal to the pivot are at positions l to j; those greater than or equal to the pivot are at positions i to r. The intervening elements (from j+1 to i-1) are equal to the pivot and require no further sorting. Here is a diagram showing the partitions and the role of variables delineating the partitions:

$\leq$ l	$\ldots$	$\leq$ j	$=$	$\ldots$	$=$	$\geq$ i	$\ldots$	$\geq$ r

The SML version of qsort using arrays is:

```
(* qsort : char array -> int -> int -> char array *)
fun qsort data l r
    = if l >= r
         then data
         else let
                 val p       = l
                 val data_p  = sub (data, p)
                 val i       = l
                 val j       = r
                 val (data , i , j )
                             = partition p data_p data l i j r
                 val data    = qsort data   l  j
                 val data    = qsort data   i  r
              in
                 data
              end ;
```

The partition function receives the index position p of the pivot, the pivot itself data_p, the input array data, the boundaries l and r of the section of the array to be partitioned, and the initial values of i and j. The partition function returns the data array with the elements between l and r partitioned, and it also returns the boundaries of the new partitions as i and j . The array data is passed to a recursive qsort to order the elements between l and j . The resulting array data is passed to the second recursive invocation of qsort. The latter orders the elements between positions i and r and returns this result as data .

The C solution follows the functional solution closely, except where the call to partition is concerned. Here the functional version returns, along with the new array data , two indices i and j . A C function can be made to return a data structure containing these elements, but it is not idiomatic C to do so. Instead, one would pass the address of the variables i and j to the partition function so that it may update these. The operator & is used to deliver the address of the variables. The partition function already has a side effect (it updates the data array), so nothing is lost by making it update the variables i and j as well.

```
void qsort( char data [], int l, int r ) {
  if( l < r) {
    int p = l ;
    char data_p = data[p] ;
    int i = l ;
    int j = r ;
    partition( p, data_p, data, l, &i, &j, r ) ;
    qsort( data, l, j ) ;
    qsort( data, i, r ) ;
  }
}
```

The idea of an efficient partitioning is to move the elements of the array by exchanging two elements at a time. This consists of a number of steps:

up The up sweep starts at the left most position (initially l) and compares the elements it finds with the pivot. It keeps moving to the right until an element is found that is greater than the pivot.

down The down sweep starts at the right most element of the partition (initially r) and keeps moving to the left until an element is found that is less than the pivot.

swap While the up and down sweeps have not met, swap the last elements encountered by both up and down, as they are both in the wrong partition. Restart the up sweep and down sweep from the next positions.

stop The process stops as soon as the up and down phases meet.

The figure below illustrates the partitioning process on a sample array of 7 elements (indexed by 0 to 6).

step	[0]	[1]	[2]	[3]	[4]	[5]	[6]	Comments
1.	‘E’	‘C’	‘F’	‘B’	‘A’	‘C’	‘G’	initial state
	i						j	initial positions `i=0` and `j=6`
2.	‘E’	‘C’	‘F’	‘B’	‘A’	‘C’	‘G’	function `up`
		i					j	‘E’ ≤ ‘E’ so `i` moves right
3.	‘E’	‘C’	‘F’	‘B’	‘A’	‘C’	‘G’	function `up`
			i				j	‘C’ ≤ ‘E’ so `i` moves right
4.	‘E’	‘C’	‘F’	‘B’	‘A’	‘C’	‘G’	function `up`
			i				j	‘F’ > ‘E’ so `i` cannot be moved
5.	‘E’	‘C’	‘F’	‘B’	‘A’	‘C’	‘G’	function `down`
			i			j		‘G’ > ‘E’ so `j` moves left
6.	‘E’	‘C’	‘F’	‘B’	‘A’	‘C’	‘G’	function `down`
			i			j		‘C’ ≤ ‘E’ so `j` cannot be moved
7.	‘E’	‘C’	$\boxed{\text{‘C’}}$	‘B’	‘A’	$\boxed{\text{‘F’}}$	‘G’	function `partition` at `i < j`
			i			j		the ‘F’ and the ‘C’ are swapped
8.	‘E’	‘C’	‘C’	‘B’	‘A’	‘F’	‘G’	function `partition`
				i	j			`i` moves right, `j` moves left
9.	‘E’	‘C’	‘C’	‘B’	‘A’	‘F’	‘G’	function `up`
					i,j			‘B’ ≤ ‘E’ so `i` moves right
10.	‘E’	‘C’	‘C’	‘B’	‘A’	‘F’	‘G’	function `up`
					j	i		‘A’ ≤ ‘E’ so `i` moves right
11.	‘E’	‘C’	‘C’	‘B’	‘A’	‘F’	‘G’	function `up`
					j	i		‘F’ > ‘E’ so `i` cannot be moved
12.	‘E’	‘C’	‘C’	‘B’	‘A’	‘F’	‘G’	function `down`
					j	i		‘A’ ≤ ‘E’ so `j` cannot be moved
13.	$\boxed{\text{‘A’}}$	‘C’	‘C’	‘B’	$\boxed{\text{‘E’}}$	‘F’	‘G’	function `partition` at `p < j`
					j	i		exchange ‘E’ (pivot) and ‘A’
14.	‘A’	‘C’	‘C’	‘B’	‘E’	‘F’	‘G’	function `partition`
				j		i		move `j` one step further and stop

The `partition` function has to do a lot of work, so we should expect it to be complicated. The code below exhibits the main structure, that is, an up sweep and a down sweep precede the main decision making. The complications arise because there are four separate cases to consider. The first case continues the recursion (when the up sweep and the down sweep have not met). The other three cases terminate the recursion, depending on the value of the final elements, the pivot has to be swapped with either element, or no swap has to take place at all. In any case the pivot element ends up ‘sandwiched’ between the two partitions. The SML implementation reads:

```
(* partition : int -> char -> char array ->
               int -> int -> int -> int ->
               (char array * int * int) *)
fun partition p data_p data l i j r
    = let
          val i  = up   data_p data i r
         val j  = down data_p data l j
      in
         if i  < j
            then partition p data_p (swap data i  j )
                           l (i +1) (j -1) r
            else if i  < p
                    then (swap data i  p, i +1, j )
                    else if p < j
                            then (swap data p j , i , j -1)
                            else (data, i , j )
      end ;
```

The structure of the `partition` function matches the requirements of the general while-schema, so the C implementation is:

```
void partition( int p, char data_p, char data[],
                int l, int *i, int *j, int r ) {
  while( true ) {
    *i = up(   data_p, data, *i, r ) ;
    *j = down( data_p, data, l, *j ) ;
    if( *i < *j ) {
      swap( data, *i, *j ) ;
      (*i)++ ;
      (*j)-- ;
    } else if( *i < p ) {
      swap( data, *i, p ) ;
      (*i)++ ;
      return ;
    } else if( p < *j ) {
      swap( data, p, *j ) ;
      (*j)-- ;
      return ;
    } else {
      return ;
    }
  }
}
```

Using the pointers to `i` and `j` is slightly clumsy, this will be solved in Exercise 7.16. The `up` and `down` functions sweep along the array and compare the elements to the pivot. The `up` function stops when either it encounters the boundary `r` or it encounters an element that exceeds the pivot. In SML:

```
(* up : char -> char array -> int -> int -> int *)
fun up data_p data i r
    = if i < r andalso sub (data, i) <= (data_p:char)
         then up data_p data (i+1) r
         else i ;
```

The `up` and `down` functions are symmetric: the `down` function sweeps downwards until it encounters either the boundary l or an element that is less than the pivot:

```
(* down : char -> char array -> int -> int -> int *)
fun down data_p data l j
    = if l < j andalso sub (data, j) >= (data_p:char)
         then down data_p data l (j-1)
         else j ;
```

The up and down functions can be translated into pure C functions using the while schema:

```
int up( char data_p, char data[], int i, int r ) {
  while( i < r && data[i] <= data_p ) {
    i++ ;
  }
  return i ;
}


int down( char data_p, char data[], int l, int j ) {
  while( l < j && data[j] >= data_p ) {
    j-- ;
  }
  return j ;
}
```

These two functions use the short circuit semantics of the &&-operator: the right hand operand is not evaluated if the left hand side evaluates to false. The expression in up that tests the loop termination checks whether the index i is within the array bounds, before actually indexing in the array data. Consider the && with its argument reversed:

```
  while( data[i] <= data_p && i < r )
```

Although the && operator is logically commutative, the result of this expression might be different since the expression data[i] might cause the program to crash before the bound check is performed.

The array version of qsort is now complete. Here is a main program to call qsort on a small SML list:

```
(* main : char array *)
val main = let
              val list = explode "ECFBACG"
              val l    = 0
              val r    = length list - 1
              val data = list_to_array list
           in
              qsort data l r
           end ;
```

Here is the corresponding C version:

```
int main( void ) {
  int l = 0 ;
```

```
    int r = 6 ;
    char data[] = "ECFBACG" ;
    qsort( data, l, r ) ;
    printf( "%s\n", data ) ;
    return 0 ;
  }
```

Exercise 7.16 The `partition` function is used only once in `qsort`. Substitute the body of the C version of `partition` in that of `qsort` and show that this removes the need for manipulating the address of the variables `i` and `j`.

We have now two versions of quicksort: one to sort a list of characters and one to sort an array of characters. Using the functions `stream_to_list` and `stream_to_array`, it is easy to build a function that sorts the contents of a stream. Such a function would not be useful, as one often is not interested in just the sorted characters of a stream. It would be more interesting to sort the lines or numbers contained in a stream. It is not difficult to extend the quicksort versions for such purposes.

Exercise 7.17 Write a program that reads a file, and sorts it on a line by line basis.

7.5 Summary

The following C constructs were introduced:

Streams The type to denote a stream in C is `FILE *`. The stream is stored externally, it can be input by the user, from another program, or from a file. A stream is a linear sequence, like an array and a list, but it can usually only be accessed sequentially.

The main programming principles that we have seen in this chapter are:

- The re-use of volatile information, such as that accessed from a stream, requires careful planning. Often, algorithms require the information from a stream to be accessed a number of times. This is inefficient, and it pays off to spend effort in trying to devise equivalent algorithms that glean the relevant information from a stream whilst making a single pass.

- The primitives provided with the standard data types are often of a low level of abstraction. In this chapter, we have built functions that provide a buffered view of a stream. It is often a good idea to invest in a collection of building blocks (functions and data structures) that deliver additional services, in this case buffering. In a sense, the buffering facilities mitigate one of the restrictions that streams impose, the sequential access. Within certain limits, the elements of a stream can be accessed in an order that is not quite sequential. Enlarging the size of the buffer weakens the limited access to the point where the buffer is capable of containing the entire stream. In that case the sequential access restriction on the stream is completely hidden. The price to pay for this flexibility is an amount of store capable of holding the entire contents of the stream.

- Side effects can be useful and allow for highly efficient programs to be written. However, a function that performs a side effect should advertise this, because care must be taken when such a function is used as a building block. Hidden side effects can lead to obscure errors.

- The open list technique has been used to develop an efficient function for reading the contents of a stream into a list. It has also been used in our implementation of the buffering technique. Here a separate data structure maintains the pointers necessary to implement the open list technique.

- If two functions with a similar structure operate one after the other on the same data structure, then these functions can often be merged. There is an important tradeoff here: the two individual functions are less specialised and therefore better building blocks than the merged result. However, the merged function is often more efficient, because the data structure (for example a stream) does not need to be stored.

- The quicksort algorithm has been presented in two forms. The first version is elegant but inefficient and uses lists. The inefficiency is caused entirely by the profligate use of store. To counter this misdemeanour an array based version was developed. This version does not resemble the elegant list based version in any way. The understanding obtained from the list based version was used to good effect in the creation of the array based version. The efficiency of the array based version is entirely attributable to the use and re-use of the original array containing the data.

7.6 Further exercises

Exercise 7.18 The database of employees (Section 5.9 and Exercise 5.8) was stored in memory. A real data base would be stored on a file. Design functions which will allow the employee database to be stored on a file. Use the following file format: the members of the structure are terminated by hash signs (#), each new structure starts on a new line. For example:

```
John#18813#1963#80#90#75#20#69#
Mary#19900#1946#72#83#75#18#75#
Bob#12055#1969#120#110#100#99#99#
Alice#15133#1972#200#230#75#11#35#
```

Exercise 7.19 Quicksort is not the only sorting method. Another method is bubble sort. It compares every element in an array of data to every other element. If two elements are out of order, then they are swapped. If there are n data elements, bubble sort will always make $n-1+n-2+\ldots+1 = \frac{n^2-n}{2}$ comparisons. Quicksort only makes this many comparisons in the worst case, it makes fewer comparisons on average. Write a C function `bubble_sort` and test it by sorting the list of strings supplied through `argc` and `argv`.

Exercise 7.20 Modify one of the sorting functions from this chapter to sort strings rather than characters. Then use your sort function to sort the lines of a file, representing the contents of a line as a string.

Exercise 7.21 Sensitive information is often encrypted such that it is difficult for unauthorised persons to intercept that information. An ancient method for encryption is the *Caesar cipher*. If a letter in the original message or *plaintext* is the n-th letter of the character set then replace this letter by the $n+k$-th letter in the encrypted message or *cipher text*. The key to encrypting and decrypting the message is then the number k. For example if the plain text is "Functional C" and the key is 4, then the cipher text is "Jyrgxmsrep$G", for the fourth letter after F is J, and so on. Write a program to implement the Caesar cipher method, taking the text to be processed from `stdin` and writing the output to `stdout`. The program should take a single argument: $+k$ for encryption or $-k$ for decryption. Would you be able to prove for all values of k that decryption after encryption is equivalent to do nothing at all, or, for people familiar with UNIX terminology:

```
(a.out +k | a.out -k) = cat
```

Exercise 7.22 The encryption of Exercise 7.21 is weak. If you know that the plain text was written in, say, the English language, then you would expect the most frequently occurring letter in the cipher text to represent the letter 'e', since this is the most frequently occurring letter in English text. It should thus not be difficult to guess what the encryption key k would be. Your guess could be confirmed by also looking at the next most frequently occurring letter and so on, to see if they all agree on the key value. Frequency tables for many natural languages are widely available. Try to get hold of such a table for your native language and write a program to analyse a cipher text produced by your Caesar cipher program so as to discover the key with as much certainty as possible.

Chapter 8

Modules

All programs developed so far were small. They typically consisted of no more than 10 functions. Real programs are larger (thousands of functions). Large programs must be designed in such a way that the code can easily be inspected, maintained, and reused. This process of organising is generally known as modularisation.

Modules are parts of the program that perform some specific function together. During the design of a program, the solution is split into modules. These modules use each other according to some well defined interface. Once the interface and the functionality of a module are defined, modules can be inspected, designed, compiled, tested, debugged, and maintained separately. Additionally, modules can be reused in other programs where a similar functionality is required.

SML has a sophisticated module mechanism. It is completely integrated with the language. An SML *structure* is a collection of types and functions; a *signature* describes the interface of a structure, and a *functor* operates on structures to create a new structure. SML modules support everything mentioned above, with the flexibility of a polymorphic type system.

The module mechanism of C is rather different. It is probably the most crude module mechanism of any programming language. The C module mechanism is implemented by a separate program, the C preprocessor. The C preprocessor takes a C program and prepares it for the C compiler. This preprocessor has no knowledge of the syntax of C, but handles the program as text instead (indeed, the C preprocessor can be used for many other purposes). The C module mechanism consequently lacks the sophistication of a real module system, as provided by SML.

The first section of this chapter describes the basic structure of the C module system, and henceforth, of the C preprocessor. After that, we discuss the concept of global variables as an important aspect of modularisation. Global variables can be used to store state information that remains accessible across function calls. When used correctly, global variables can be an asset. However, it is easy to abuse them and obscure code. We show how global state can be stored differently, leading to a cleaner interface. This is akin to an *object oriented* style of programming. After this, we show how modules can also be generalised, by discussing the counterpart of polymorphism.

8.1 Modules in C: files and the C preprocessor

The module concept of C is text oriented. The module facilities are implemented by the *C preprocessor*. This is a program that is separate from the C compiler. As far as the C preprocessor is concerned, it can process C programs, but it can process an assembly language program or an HTML script equally well. The entity on which the C preprocessor operates is the *file*. A file that is passed to the C compiler for compilation is first processed by the C preprocessor. All lines that start with a hash-sign # are interpreted as commands. Textual substitutions are made where necessary throughout the entire file, substitution is not restricted to lines beginning with a # sign. Below, we create a module that implements complex arithmetic. After that, the features of the C preprocessor are explored in detail.

8.1.1 Simple modules, `#include`

As an example, we develop a module for arithmetic on complex numbers. A complex number is defined as follows:

$$\begin{array}{rcl} r,s & : & \mathbb{R} \\ c & : & \mathbb{C} \\ c & = & ri+s \end{array}$$

Here r is the imaginary part of the complex number c, $i = \sqrt{-1}$, and s is the real part of c. Complex arithmetic works just like ordinary arithmetic, except that we have to take care that i is properly handled. Examples:

$$\begin{array}{rcll} r,s,t,u & : & \mathbb{R} & \\ c,d & : & \mathbb{C} & \\ c & = & ri+s & \\ d & = & ti+u & \\ c-d & = & (ri+s)-(ti+u) = ri+s-ti-u = (r-t)i+(s-u) & (8.1) \\ cd & = & (ri+s)(ti+u) = riti+riu+sti+su = (ru+st)i+(su-rt) & (8.2) \end{array}$$

Equality (8.2) is true because $i^2 = (\sqrt{-1})^2 = -1$.

The implementation of complex subtraction and multiplication as functions in SML is given below. The `complex_distance` function gives the distance from the origin of the complex plane.

```
structure Complex = struct
  type complex = real * real ;

  (* complex_sub : complex -> complex -> complex *)
  fun complex_sub (r,s) (t,u)
      = (r-t,s-u) : complex ;

  (* complex_multiply : complex -> complex -> complex *)
  fun complex_multiply (r,s) (t,u)
      = (r*u+s*t,s*u-r*t) : complex ;

  (* complex_distance : complex -> real *)
  fun complex_distance (r,s)
      = sqrt (r*r+s*s) ;
end ;
```

The structure Complex is defined containing a type complex, and the functions to operate on complex numbers. These functions can be implemented in C as shown below. The first line imports the mathematics module that provides the sqrt function.

```
#include <math.h>

typedef struct {
  double im , re ;
} complex ;

complex complex_sub( complex c, complex d ) {
  complex r ;
  r.im = c.im - d.im ;
  r.re = c.re - d.re ;
  return r ;
}

complex complex_multiply( complex c, complex d ) {
  complex r ;
  r.im = c.im * d.re + d.im * c.re ;
  r.re = c.re * d.re - d.im * c.im ;
  return r ;
}

double complex_distance( complex c ) {
  return sqrt( c.im * c.im + c.re * c.re ) ;
}
```

To use these functions as a separate module, an *interface* to the module must be defined. The interface lists all functions and types that need to be accessible from outside the module. The types are defined as usual. For functions, the names and types of the arguments and the result are given. In SML, the interface to a module is a *signature*. Here is the signature of the complex number module:

```
signature COMPLEX = sig
  type complex ;
  val complex_sub      : complex -> complex -> complex ;
  val complex_multiply : complex -> complex -> complex ;
  val complex_distance : complex -> real ;
end ;
```

An implementation is associated with this signature as follows:

```
structure Complex : COMPLEX = struct
  type complex = real * real ;

  fun complex_sub (r,s) (t,u)
      = (r-t,s-u) : complex ;

  fun complex_multiply (r,s) (t,u)
      = (r*u+s*t,s*u-r*t) : complex ;

  fun complex_distance (r,s)
      = sqrt (r*r+s*s) ;
end ;
```

The point of separating the interface of a module from its implementation is to be able to maintain and use them separately. SML offers the construct `sig`... `end` to encapsulate an interface and the construct `struct` ... `end` to encapsulate the implementation. The programmer has the option of storing them in separate files or keeping them together. In C, this is not the case: an interface must be stored separately from an implementation.

In C, the interface to a module is stored in a *header file*. The information it contains is similar to the signature of SML. The definition of a function in the header file is known as the *prototype* of the function. It consists of the result type of the function, its name, and the types of its arguments. Here is the interface to the C version of the complex number module:

```
typedef struct {
  double im , re ;
} complex ;

extern complex complex_sub( complex c, complex d ) ;
extern complex complex_multiply( complex c, complex d ) ;
extern double  complex_distance( complex c ) ;
```

The interface gives the type `complex` and the prototypes of the three functions that operate on values of that type. The word `extern` before the function declaration indicates that these three functions can be considered as external functions by any module using the complex module. The interface must be stored in a separate file. By convention, this file has a name that ends with a `.h` suffix (for header). The implementation of the functions is then stored in a file named with a `.c` suffix. Thus we could store the interface above in the file `complex.h`. The implementation of the functions would be found in the file `complex.c`:

```
#include <math.h>
#include "complex.h"

complex complex_sub( complex c, complex d ) {
  complex r ;
  r.im = c.im - d.im ;
  r.re = c.re - d.re ;
  return r ;
}

complex complex_multiply( complex c, complex d ) {
  complex r ;
  r.im = c.im * d.re + d.im * c.re ;
  r.re = c.re * d.re - d.im * c.im ;
  return r ;
}

double complex_distance( complex c ) {
  return sqrt( c.im * c.im + c.re * c.re ) ;
}
```

Note that the type definition has disappeared from the code and has been replaced by the following line:

```
#include "complex.h"
```

The `#include` directive tells the compiler to literally include the file `complex.h` at this place in the program. The type definition and the prototypes are imported by the C preprocessor. The C compiler will verify that the prototype of each function that is exported is consistent with the implementation.

The `#include` directive has been introduced earlier, in Chapter 2, where it was used it to import the module performing input and output. Indeed the statement below imports the interface of the module `stdio`:

```
#include <stdio.h>
```

The name `stdio` is short for 'standard input and output'. We write the filename between the angular brackets < and >, instead of the double quotes shown around `"complex.h"`, because `stdio.h` is a standard library. The names of header files that do not belong to the standard library should be enclosed in double quotes. Similarly, the header file `math.h` has to be included, which allows the program to use the function `sqrt`.

Any C module that uses the complex number module should have a line that imports the complex number interface. As an example, consider a main program to calculate the square of a complex number and then subtract the square from the number itself:

```
#include <stdio.h>
#include "complex.h"

complex complex_square( complex x ) {
  return complex_multiply( x, x ) ;
}

void complex_print( complex x ) {
  printf( "(%f+%fi)", x.re, x.im ) ;
}
```

```
int main( void ) {
  complex x = { 2.0, 1.0 } ;
  complex y = complex_square( x ) ;
  complex z = complex_sub( x, y ) ;
  complex_print( y ) ; printf( "\n" ) ;
  complex_print( z ) ; printf( "\n" ) ;
  return 0 ;
}
```

The main program and the complex number module can be compiled separately. The interface of the complex number module in the file `complex.h` links these two together. To make this link explicit, `complex.h` specifies all types and functions of `complex.c` that are to be used by any other module. In this example, `complex.h` only specifies the type of only a single `struct`-definition, but in other cases it might require the definition of enumerated types, unions, and `#defines`.

8.1.2 Type checking across modules

The module system is the Achilles heel of the C type system; it is important that the C programmer is aware of this weakness so to avoid common mistakes. As described so far, the typing system is safe (that is if certain C features, such as `void *`, type casts and variable argument list are not used). If the prototype of a function specifies three arguments of certain types, the compiler will not accept a call with any other number of arguments or with the wrong types. There are, however, two weaknesses:

1. What if there is no prototype?
2. Can the compiler verify that the prototype is correct?

The first problem is caused by backward compatibility of ISO-C with C. Any function that is called for which no prototype has been declared is assumed to be a function returning an integer number, and no checking of any kind is performed on the argument list. This means that if one forgets to include a header file, the compiler will not complain about using undefined functions, but will simply assume that these functions return an integer. Most of the time the compilation will fail because of some other problem. If, for example, the module `complex.h` is not included, the type `complex` is not defined, which will result in an error. However, a classic error is not to include the file `math.h` when using the mathematical library. When calling the function `sin(x)` or `sqrt(x)`, the compiler will silently assume that these function return an integer. The results are dramatic. Most modern compilers will (on request) inform the programmer about missing prototypes.

The second problem is also interesting. When a module, for example `complex.h`, is included, the compiler must assume that the specification is correct. The correctness of this specification is verified when `complex.c` is compiled. So far so good, but if the programmer forgets to include `complex.h` in the source complex.c, these checks are not performed. Again, the compiler will usually find some other error, for example, because the type complex has not been defined. Modules that do not define types may compile without problem though. Most modern compilers will warn the programmer on request when a function is used that was not prototyped. This warning catches 99% of the errors that arise because of improper use of the module facility.

Both problems are due to the looseness of the module system of C. Languages like Modula-2 and SML do not suffer from these problems, since their module systems were designed to be watertight.

8.1.3 Double imports

In large programs, one often needs the interface to one module whilst building the interface to another module. For example, a module with graphics primitives might rely on a module defining matrices, vectors, and the associated arithmetic. In turn, these modules might depend on other module interfaces.

As in SML, C allows module interfaces to be imported in interfaces. However, the C module system relies on the programmer to do so correctly. As a first example, suppose that the graphics module uses matrices and vectors. Therefore, the header file of the graphics module reads:

```
#include "vector.h"
#include "matrix.h"
/*C graphics function prototypes*/
```

The matrix package in turn needs a definition of vectors, in order to supply operations to multiply a matrix and a vector:

```
#include "vector.h"

typedef struct {
  vector *columns ;
  int coordinates ;
} matrix ;
extern matrix matrix_multiply( matrix x, matrix y ) ;
extern vector matrix_vector( matrix x, vector y ) ;
```

Finally, the vector header file defines some type for a vector:

```
typedef struct {
  double *elements ;
  int coordinates ;
} vector ;
extern double vector_multiply( vector x, vector y ) ;
extern vector vector_add( vector x, vector y ) ;
```

When the interface of the graphics module is used somewhere, the C preprocessor will expand all include directives to import all types and primitives. This will result in the following collection of C declarations:

```
typedef struct {           /*Lines imported by graphics.h*/
```

```
  double *elements ;
  int coordinates ;
} vector ;
extern double vector_multiply( vector x, vector y ) ;
extern vector vector_add( vector x, vector y ) ;

typedef struct {           /*Lines imported via matrix.h*/
  double *elements ;
  int coordinates ;
} vector ;
extern double vector_multiply( vector x, vector y ) ;
extern vector vector_add( vector x, vector y ) ;

typedef struct {           /*Lines imported by graphics.h*/
  vector *columns ;
  int coordinates ;
} matrix ;
extern matrix matrix_multiply( matrix x, matrix y ) ;
extern vector matrix_vector( matrix x, vector y ) ;

/*C graphics function prototypes*/
```

Note that the header file `vector.h` has been included twice. Because the C module mechanism is entirely text-based, the C preprocessor has no objections against including a header file twice. This double inclusion causes the compiler to flag an error, because the types are defined twice. To avoid double inclusion, header files must explicitly be protected. This protection can be implemented by means of an `#ifndef` directive, which conditionally includes text. A proper header file for the vector module would be:

```
#ifndef VECTOR_H
#define VECTOR_H

typedef struct {
  double *elements ;
  int coordinates ;
} vector ;

extern double vector_multiply( vector x, vector y ) ;
extern vector vector_add( vector x, vector y ) ;

#endif /* VECTOR_H */
```

This header file should be read as follows: the part of text enclosed by the `#ifndef VECTOR_H` and `#endif` will only be compiled if the identifier `VECTOR_H` is *not* defined (`#ifndef` stands for ‘if not defined’). Thus, the first time that this header file is included, the program text in it, starting with `#define VECTOR_H` and ending with the prototype of `vector_add`, will be included. This code actually defines VECTOR_H, so the next time that the header file is included, the code will not be included. This is a low level mechanism indeed, but it does the trick.

The C module mechanism really breaks down when names of functions of various modules clash. All functions of all modules essentially share the same name space. This means that no two functions may have the same name. It is good practice to ensure that function names are unique and clearly relate to a module, in order to prevent function names clashing. The complex number module at the beginning of this chapter uses names of the form complex_... to indicate that these belong to the complex number module.

8.1.4 Modules without an implementation

The modules that we have seen so far have an implementation and an interface. Not all modules need an implementation; a module may consist of only an interface. As an example, we can define a module for the booleans as follows:

```
#ifndef BOOL_H
#define BOOL_H

typedef enum { false = 0 , true = 1 } bool ;

#endif /* BOOL_H */
```

This module only exports a type, and it has no corresponding source file. From now on, we will import the module bool.h when booleans are needed.

Like the vector module, the interface of the boolean module has been protected against multiple inclusion, using a #ifndef and #define.

8.1.5 The macro semantics of #define directives

The #define mechanism is used for other purposes than for preventing the multiple inclusion of header files. In Chapter 5, we saw that constants can be defined using a #define. In general, #define associates an identifier with a (possibly empty) sequence of characters:

```
#define monkeys    42 gorillas and 12 gibbons
```

This replaces the identifier monkeys by the text 42 gorillas and 12 gibbons everywhere in the program text after this definition. This mechanism can be abused by defining, for example:

```
#define BECOMES   =
#define INCREMENT ++
```

Now we can write i INCREMENT instead of i++, or i BECOMES 13 instead of i = 13. This will make the code guaranteed unreadable for anyone but the original author. Because replacement of identifiers is textual, the scoping rules of the C grammar are ignored by the C preprocessor. This means that #define statements inside a function have a global effect. Another consequence is that seemingly logical #define directives do not have the desired effect. Consider the following program fragment with three #define directives:

```
#define x    4
#define y    x+x
#define z    y*y
int q = z ;
```

In this example, occurrences of x are replaced by 4, occurrences of y are replaced by 4+4, and occurrences of z are replaced by 4+4*4+4. So x=4, y=8, and z=24, and not 64, as one might have expected. To prevent this kind of error, expressions named using a #define should always be parenthesised:

```
#define x    (4)
#define y    (x+x)
#define z    (y*y)
int q = z ;
```

The definitions above result in the value 64 for z. Note that the parentheses around the 4 are actually not necessary, as 4 will always be interpreted as 4. Parentheses around numbers are often omitted.

The #define can be used with arguments. The arguments are placed in parentheses after the identifier to be defined (spaces between the identifier and the parentheses are illegal). An identifier defined in this way is known as a *macro*. When an occurrence of the macro is replaced by its value, the arguments are substituted textually. As an example, consider the following definition, which uses the ternary conditional expression ... ? ... : ... (defined in Section 2.2.5):

```
#define min(x,y)   ( (x)<(y) ? (x) : (y) )
```

This states that wherever the call min(... , ...) is found, the following text should appear instead:

```
( (x)<(y) ? (x) : (y) )
```

All occurrences of x and y are replaced by the first and second argument of min. Thus, the text min(p , q) will be replaced by:

```
(( p )<( q )?( p ):( q ))
```

Evaluating this expression yields p , which is indeed the smallest of the two characters. The parentheses around x and y are necessary to ensure that argument substitution does not result in unexpected answers. However, despite the parentheses, this macro still exhibits peculiar semantics. Consider the following example:

```
min(  p , getchar() )
```

This should intuitively return the minimum of the character p and the next character on the input stream. Unfortunately, this is not the case, as the textual substitution results in the following expression:

```
( ( p )<( getchar() ) ? ( p ) : ( getchar() ) )
```

The conditional first calls `getchar` and checks if the value is greater than `p` . If this is the case, the value of the expression is `p` . If it is not the case, `getchar` *is called again*, which has a side effect.. This causes the final value to be the *second* character on the input stream (which may or may not be greater than `p`). However, the macro `min` above does have an advantage over the following function `minf`:

```
int minf( int x, int y ) {
  return x < y ? x : y ;
}
```

The *function* `minf` works only if both arguments are of the type `int`. However, the *macro* `min` works on any combination of types. The term that is used to describe this behaviour is that in C macros are *polymorphic*, while functions are *specialised* for certain argument types.

Although polymorphism can be a good reason to use macros, the programmer should be aware that a macro cannot be passed to a higher order function. (Remember that macros are textually substituted). Macros can be quite useful, but they have to be used with care.

8.2 Compiling modules

The C language has been designed so that modules can be compiled separately. That is, the compiler can compile one module without knowledge of the implementation of any of the other modules. The header files provide the necessary information about the interfaces of the other modules. Therefore, if the code of one module is changed, only that module needs to be recompiled. The code from all other modules stays the same. This shortens the development cycle of a C program, as only small parts of programs need to be recompiled.

If one of the header files is changed (for example, because the argument list of a function has been changed), each module that imports the changed header file needs to be recompiled. This is necessary for the compiler to verify that the changed interface is still used correctly. If a module is not recompiled, it might lead to a runtime error. Because it is hard to remember which modules need to be compiled upon a change, most development environments offer a facility to recompile all modules that are dependent on a changed interface.

8.2.1 Separate compilation under UNIX

UNIX C-compilers have a command line option to tell them that a module must be compiled separately. This means that the compiler will not generate an executable, but it will generate an *object*-file. As an example, we can compile the source code of the module `vector` above by:

```
cc -c vector.c
```

The -c option means 'compile this module only'. The C compiler will generate an object-file and store this under the name vector.o. Filenames ending with .o are *object files*. When modules are compiled separately, a final stage is needed which generates an executable, given a number of object-files. This stage is called the *linking* process. In order to link three objects files, say vector.o, matrix.o and graphics.o, we call the C-compiler without the -c option. Because the arguments are object files (recognised by their .o suffixes), the compiler will link the modules together in one binary. In the example below, we have passed the -o option to give the binary a meaningful name:

```
cc vector.o matrix.o graphics.o -o graphics
```

So all the programmer has to do in order to generate an executable is compile all modules that need compilation, and to link the binaries together. The following sequence of commands would achieve this on a UNIX system:

```
cc -c vector.c
cc -c matrix.c
cc -c graphics.c
cc vector.o matrix.o graphics.o -o graphics
```

These statements will compile all modules and link the binaries. Because typing these commands and remembering precisely which modules need to be compiled is an error-prone and tedious process, UNIX supplies a tool which does the work, make. When invoked, make recompiles all modules that need to be recompiled. It needs a file that specifies what to make and how to make it; this file is known as the *make file* and is usually called makefile or Makefile.

Basic make files

The make file specifies how certain *targets* can be made, given certain *dependencies*. The *targets* are the files that are generated by make. The *dependencies* are the files used in the process of making a target. An example target is the file graphics, while example dependencies are the files vector.c and vector.h.

As a first rule for the make file, we are going to state how to generate the target vector.o:

```
vector.o:
	cc -c vector.c
```

These two lines tell make that the target vector.o can be made by executing the command cc -c vector.c. (A low level but important note for the reader: the line with cc should start with a tabulate-character; make will fail if the line starts with 8 spaces instead!) What this fragment of code lacks is the information *when* vector.o needs to be made. There are two possible reasons why vector.c should be recompiled:

1. We have changed something in the program. That is, something in vector.c has been modified.

2. We have changed something in the interface (`vector.h`). Although compilation is not necessary to generate new object-code, it is necessary to verify that the source file and the header file are consistent.

In make terminology, `vector.c` and `vector.h` are *dependencies* of the target `vector.o`. Dependencies can be specified in the makefile by a line `target: dependency`. Therefore, the example `makefile` can be completed by adding the two dependencies:

```
vector.o:
        cc -c vector.c
vector.o: vector.c
vector.o: vector.h
```

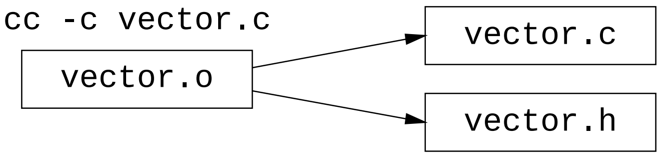
The first of the two dependencies specifies that if `vector.c` has changed (the code of the program was modified), then the module must be recompiled. The second dependency specifies that if `vector.h` has changed (the header file was updated), the module also needs recompilation. Dependencies can be given on separate lines, but one can also state multiple dependencies on a single line. The last two lines can be replaced by one line:

```
vector.o: vector.c vector.h
```

This shortens the make files a bit if there are many dependencies. It is also possible to merge the dependencies with the rule for generating the target:

```
vector.o: vector.c vector.h
        cc -c vector.c
```

The dependencies are conveniently displayed in a picture, known as a dependency graph. The picture below shows three boxes. The two boxes on the right hand side denote the dependencies (`vector.h` and `vector.c`), the box on the left hand side shows the target (`vector.o`), the arrows show the dependencies (`vector.o` depends on both `vector.c` and `vector.h`), and the line in italics shows the command that is used to generate the target:



The module `vector` had only two dependencies, the header file and the code file. The module `matrix` has more dependencies. `matrix` includes the header files `matrix.h` and `vector.h`. Thus `matrix` must be compiled if either `matrix.c` is updated (the code of the matrix module), if `matrix.h` is changed (to verify that the header still matches with the code file), or if `vector.h` is updated (to verify that the use of the vectors complies with the changed definition). The part of the `makefile` that specifies how to compile the module `matrix` is:

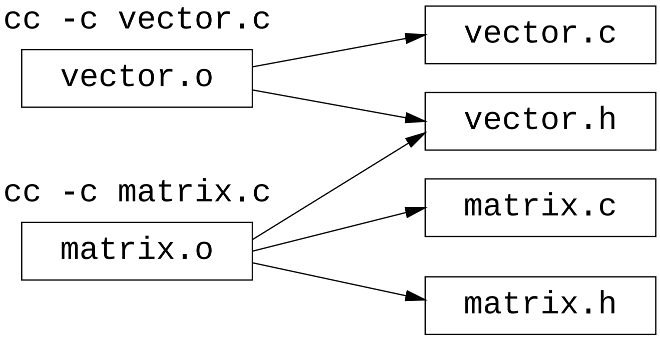
```
matrix.o: matrix.c matrix.h vector.h
        cc -c matrix.c
```

There are now three dependencies, the last one explaining that the target `matrix.o` needs to be recompiled if `vector.h` has been updated.

The two parts of the make file for the modules `vector` and `matrix` can now be concatenated to one `makefile`, specifying how to compile `vector` and `matrix`. This combined `makefile` is shown below. We have placed multiple dependencies on a single line:

```
vector.o: vector.c vector.h
        cc -c vector.c

matrix.o: matrix.c matrix.h vector.h
        cc -c matrix.c
```

The dependency graph for this make file is:



Note that there are two arrows pointing to the file `vector.h`. If `vector.h` is updated then both modules vector and matrix have to be recompiled.

The third module that was used was `graphics`. The module `graphics` includes the header files for `matrix.h`, `vector.h`, and `graphics.h`. Creating the rules for `graphics` is left as an exercise to the reader.

Exercise 8.1 Give the rules for the `makefile` that would describe how to compile the module `graphics`.

Exercise 8.2 Integrate the rules of exercise 8.1 with the previous `makefile` (that specified how to compile the modules `vector` and `matrix`) and draw the dependency graph.

So far, this example has explained how and when modules are to be compiled. What is missing is the linking stage. The rule for *how* to link the modules, assuming that the final binary is called `graphics`, is:

```
graphics:
        cc -o graphics vector.o matrix.o graphics.o
```

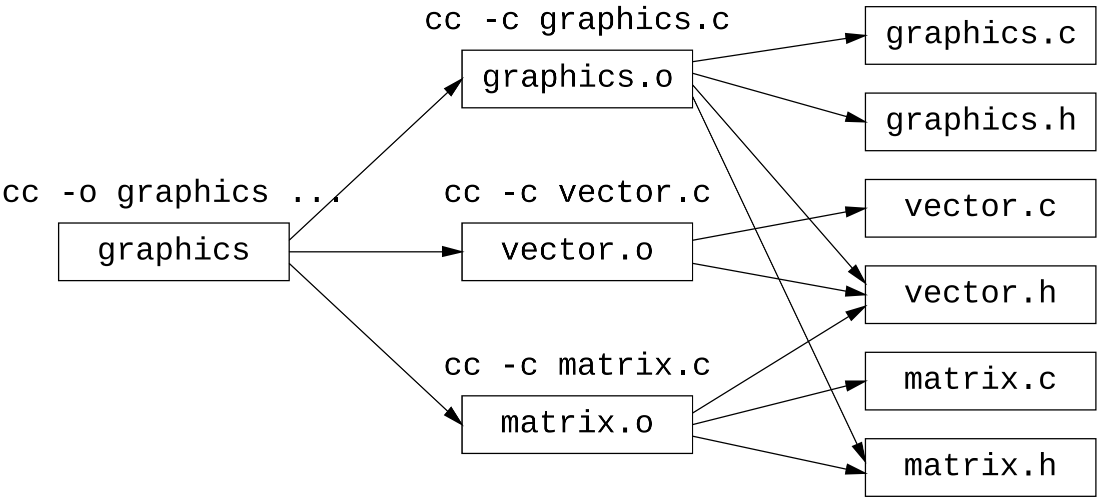
We must also state *when* graphics must be linked. In this case, the target `graphics` must be made when either of the three dependencies `vector.o`, `matrix.o` or `graphics.o` has changed. The dependencies are:

```
graphics: vector.o matrix.o graphics.o
```

Note that each of these three is a actually target of a previous rule! This is shown when we integrate all parts of the make file:

```
graphics: vector.o matrix.o graphics.o
        cc -o graphics vector.o matrix.o graphics.o
vector.o: vector.c vector.h
        cc -c vector.c
matrix.o: matrix.c matrix.h vector.h
        cc -c matrix.c
graphics.o: graphics.c graphics.h matrix.h vector.h
        cc -c graphics.c
```

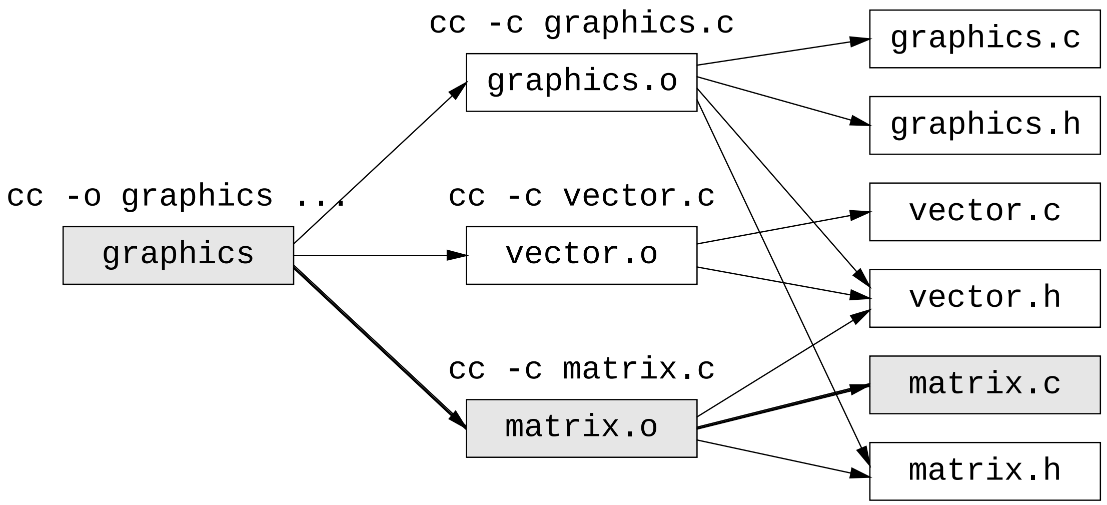
The complete dependency graph can be derived from the makefile:



The fact that vector.o is both a target (of the rule that specified how to compile the module vector) and a dependency (of the rule that specified how to link the program) is shown here as a box that has both inward and outward arrows.

The order of rules in the makefile is irrelevant but for one exception: the rule that is specified at the top specifies the target that will be made by default.

The dependency graph is used by make to find out which modules need to be recompiled. As an example, assume that the file matrix.c was updated recently. By tracing the dependencies backwards, make can deduce that it must remake matrix.o and graphics. What is more, they must be made in that order. This process of following the dependencies is shown below. The targets graphics and matrix.o in the grey boxes need to be remade.



Exercise 8.3 Use the dependency graph to determine which files need to be recompiled when `matrix.h` is changed.

Default rules

It is permitted to omit the rule for certain targets. In this case, the `make` program uses *default rules* to decide how to make the target. In order to find out how to generate a target, `make` will use the file-suffix of the target. Given the file-name suffix, make will search for a default rule that specifies how to make a target.

One of the built-in default rules of make is that any file ending with `.o` can be generated by invoking the C-compiler on a corresponding `.c` file. Other rules state how Fortran or Pascal programs can be compiled. The advantage of this facility is twofold: the make file becomes shorter (now only giving dependencies and the final target rule) and the programmer does not have to worry about the precise compiler options anymore. Our example makefile would be reduced to:

```
graphics: vector.o matrix.o graphics.o
        cc -o graphics vector.o matrix.o graphics.o

vector.o: vector.c vector.h
matrix.o: matrix.c matrix.h vector.h
graphics.o: graphics.c graphics.h matrix.h vector.h
```

Although the rules for compilation can be omitted, the dependencies must still be specified. You can create your own default rules in makefiles, but that subject is beyond the scope of this text. The interested reader is referred to the appropriate UNIX manual pages.

Abbreviations

In the example make file, some text fragments are repeated. As an example, the list of objects that are needed to build the whole program is specified twice:

```
graphics: vector.o matrix.o graphics.o
        cc -o graphics vector.o matrix.o graphics.o
```

The first line lists the objects in the dependencies; the second lists the objects as part of the rule how to make `graphics`. Listing such a set more than once is not only tedious, but it also increases the likelihood of making errors. For this reason, `make` has an abbreviation facility. (The official make terminology is 'macro', but we use abbreviation in this text in order to distinguish them from macros in C.) A text fragment T can be given a name N as follows:

N=T

While before we had to cut and paste the text fragment, we can now write `$(`N`)`. This will be replaced by the text fragment associated with N, T:

```
OBJECTS=vector.o matrix.o graphics.o
graphics: $(OBJECTS)
        cc -o graphics $(OBJECTS)
```

Here `OBJECTS` is used as an abbreviation for the following text:

```
vector.o matrix.o graphics.o
```

The string `$(OBJECTS)` is replaced by the definition every time it is encountered.

There are many standard abbreviations. As an example, `CFLAGS` is used to denote any flags that are to be passed to the C compiler by the built-in rule for compiling a C module. The line below could be inserted in any make file:

```
CFLAGS=-O
```

This causes the `-O` flag to be passed to the C compiler (asking the compiler to optimise the code, investing extra time to make the code faster). Another built-in abbreviation is `CC` which is an abbreviation of the name of the C-compiler. Normally `CC` equals `cc`, but on certain systems where the C-compiler has a different name, the abbreviation will point to a different compiler. A typical use of the `CC` abbreviation might be to use the GNU C compiler from the Free Software Foundation:

```
CC=gcc
```

Making dependencies

The program `make` is convenient to use, provided the make file is correct. If the user makes a mistake in the make file, `make` will not recompile modules when they should be recompiled. This can lead to disastrous results (especially since the linker of C is not fussy at all about inconsistencies). Suppose that the following dependency had been missed out:

```
matrix.o: vector.h
```

In this case, a change in the interface of the vector module will not lead to recompilation of `matrix`.

To prevent these errors, a utility program is available that finds out which dependencies exist between various C modules. This program, called `makedepend`, reads a number of C source modules and generates all the dependencies that exist between these modules. In its simplest form, makedepend is just invoked with all the source modules involved. It will generate all the dependencies and write them in the make file. As an example, consider the following make file:

```
OBJECTS=vector.o matrix.o graphics.o
graphics: $(OBJECTS)
        cc -o graphics $(OBJECTS)
```

The first goal specifies how graphics is to be made; there are no dependencies. We can now type the following command:

```
makedepend graphics.c vector.c matrix.c
```

This will cause makedepend to find all dependencies for these three source files and to write them at the end of the make file. The resulting make file is:

```
OBJECTS=vector.o matrix.o graphics.o
graphics: $(OBJECTS)
        cc -o graphics $(OBJECTS)

# DO NOT DELETE THIS LINE -- make depend depends on it.

graphics.o: graphics.c graphics.h vector.h matrix.h
vector.o: vector.c vector.h
matrix.o: matrix.c matrix.h vector.h
```

The line starting with the #-sign is a comment line in the make file, makedepend inserts it so that on the next run it can throw all old dependencies away. The three lines after the comment line state precisely which modules depend on each other. Every time the user changes something in the way that modules interact, makedepend should be run to update the make file. One can make a target 'depend' in the make file, so that typing make depend makes the dependencies:

```
depend:
        makedepend graphics.c vector.c matrix.c
```

The complete make file for the graphics system is:

```
CFLAGS=-O
OBJECTS=vector.o matrix.o graphics.o

graphics: $(OBJECTS)
        cc -o graphics $(OBJECTS)

depend:
        makedepend graphics.c vector.c matrix.c

# DO NOT DELETE THIS LINE -- make depend depends on it.

graphics.o: graphics.c graphics.h vector.h matrix.h
vector.o: vector.c vector.h
matrix.o: matrix.c matrix.h vector.h
```

As long as there is no file named depend the rule for depend will always be executed when requested. For more advanced information on make and makedepend the reader is referred to the manual pages of these two programs.

8.2.2 Separate compilation on other systems

The reader may at this moment exclaim "This is one big kludge! Each loophole is patched with another loophole". This statement is true to a certain extent. The reason why it seems to be a system of patches on patches is that the UNIX design philosophy (where make, C, and many other tools stem from) states that all tools should be as reusable as possible.

This was a revolutionary idea in the early seventies. The program make can be used to compile any language, not just C. It can compile Fortran programs, make a new UNIX kernel, or be used to decide which chapters of a book need to be typeset. Similarly, UNIX offered one editor that was used for all purposes; furthermore the C preprocessor is, as we have seen before, to a large extent C independent. The advantage is that one has one big toolbox containing many (semi-)general purpose tools.

The disadvantage of this approach is that it is not user friendly at all. Given the set of C modules that constitute a C program, the compiler could itself decide which modules to recompile, without needing a make file or make file-generator. This is more user friendly. Indeed, this is the approach taken by many C development environments running on Macintoshes or modern PCs. Internally, these environments have an editor, a make facility, a tool to work out which dependencies exist, a C preprocessor, a compiler, and a linker, but the subtle differences are hidden from the user. Additionally, an integrated editor allows the editor to check the syntax of the program and can highlight compiler errors in the source code.

Many of these development environments exist; most of these have built-in facilities to recompile only those modules that need recompilation. It is impossible to address all these systems in detail. The reader is referred to the appropriate documentation for the system of interest.

8.3 Global variables

All functions that have been defined so far are functions that operate on values provided via their arguments. Imperative languages have another place where data can be stored known as *global variables*. Global variables can 'remember' data between function calls. When used with care, global variables can be an asset. However, it is often better to use explicit global state, which is not hidden in global variables. As an example, we will develop a pseudo random number generator.

8.3.1 Random number generation, how to use a global variable

Random numbers are often used in simulation programs, where a random choice is to be made. True random numbers cannot be generated by a program, but good pseudo random numbers can be computed. A widely used algorithm is the linear congruential method, which uses the sequence:

$$\begin{array}{rcl} n, m, x_i & : & I\!\!N \\ x_0 & = & 1 \\ x_i & = & n x_{i-1} \bmod m, \text{ if } i > 0 \end{array}$$

If n and m are carefully chosen (m should be a prime number and n should be a primitive root of m), the numbers $x_0, x_1, x_2, \ldots$ will traverse all numbers between 1 and $m-1$ inclusive in a seemingly random order. As an example, let $m = 31$ and $n = 11$, then the sequence of generated numbers is (reading from left to right, top to bottom):

$$\begin{array}{rrrrrrrrrr} 1 & 11 & 28 & 29 & 9 & 6 & 4 & 13 & 19 & 23 \\ 5 & 24 & 16 & 21 & 14 & 30 & 20 & 3 & 2 & 22 \\ 25 & 27 & 18 & 12 & 8 & 26 & 7 & 15 & 10 & 17 \\ 1 & 11 & 28 & \ldots \end{array}$$

The first number of this sequence is called the *seed* of the random number generator. For this type of generator, any seed in the range $1 \ldots m-1$ will work.

Given the series x_0, x_1, x_2, ... which are numbers in the range $1 \ldots m-1$, one can obtain a series of numbers in the range $0 \ldots k-1$ by using the numbers $x_i \bmod k$. Thus for the simulation of a coin that can either be head (1) or tail (0), choose $k = 2$, which gives the pseudo random sequence:

$$\begin{array}{rrrrrrrrrr} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & \ldots \end{array}$$

This series, like the previous one, repeats itself after 30 numbers. This is the period of the random generator, defined by m and n. The range of the numbers is given by k. In a general random number generator k can be chosen by the user, but m and n are chosen by the designer of the module to guarantee a long period.

The SML interface of the random number module would be:

```
signature RANDOM = sig
  type random ;
  val random_init : random ;
  val random_number : random -> int -> random ;
end ;
```

The implementation needs two values `period` and `root` that are not available to the users of the random number generator. Here the interface places a restriction on the SML module.

```
structure Random : RANDOM = struct
  type random = (int * int) ;

  val period = 31 ;
```

```
    val root = 11 ;

    val random_init
        = (1,1) ;

    fun random_number (num,seed) range
        = (seed mod range, root * seed mod period) ;
  end ;
```

The caller of the random number generator receives a tuple containing the state of the random generator and the random number that the caller required. By passing the tuple on to the next call, `random_number` can calculate the next number in the series without difficulties (a standard functional trick to deal with state). The interface of the naive C version of the random number module is:

```
typedef struct {
  int number ;
  int seed ;
} random ;

extern random random_init( void ) ;
extern random random_number( random state, int range ) ;
```

Here is the C implementation of the random number module:

```
#include "random.h"

#define period 31
#define root 11

random random_init( void ) {
  random x = {1,1} ;
  return x ;
}

random random_number( random state, int range ) {
  random result ;
  result.number = state.seed % range ;
  result.seed = root * state.seed % period ;
  return result ;
}
```

The caller of the function `random_number` has to know about the data type `random`, as it must unpack this to take the number out, before passing it on to the next call of random. Using a structure does not give rise to an elegant solution, but with a global variable the problem can be solved satisfactorily.

```
extern void random_init( void ) ;
extern int random_number( int range ) ;
```

```
#include "random.h"
```

```
static int state ;

#define period 31
#define root 11

void random_init( void ) {
  state = 1 ;
}

int random_number( int range ) {
  int result = state % range ;
  state = root * state % period ;
  return result ;
}
```

This new function random_number takes only one argument, the desired range of the random number. The old random number is remembered in the global variable named state. The function random_number thus side effects the global variable state. The variable state is declared *static*:

```
static int state ;
```

The keyword static indicates that the variable state lives as long as the program runs. This in contrast with local variables, which cease to exist when the function in which they have been declared terminates. Because state is declared outside the functions, state is visible to all functions in this module. In this case, the variable state stores the last number of the sequence. Every time random_number is called, the function calculates the random number and updates state.

The new module has a cleaner interface to the outside world, but it does have two serious drawbacks. The first drawback is that random_number is no longer a pure function. Calling it twice with identical arguments results in different numbers. The second disadvantage is that the single state limits the use of the module to a single stream of random numbers. If one program needs two independent streams of random numbers for two different modules, the random number module will not be able to cope with this. Thus, the present version of the random number module is not a good building block. These problems can be alleviated by moving the state out of the module.

8.3.2 Moving state out of modules

Clean C modules use a purely functional interface, as, for example, the module on complex numbers presented in the first section of this chapter. Some functions like the random number generator shown above are naturally formulated using state and state changes. However, they can still be designed so that they have a clear interface. The purely functional interface of the random number module was inconvenient in C, while the version with state was not a good building block. A third solution can be formulated using the following interface:

```
typedef struct {
  int seed ;
} random ;

extern random *random_init( void ) ;
extern int     random_number( random *state, int range ) ;
```

Here the function `random_init` generates a new state for the random number, and the function `random_number` modifies the state pointed to by its argument `state`. This is implemented as follows:

```
#include "random.h"

#define period 31
#define root 11

random *random_init( void ) {
  random *state = malloc( sizeof( random ) ) ;
  if( state == NULL ) {
    printf( "random_init: no space\n" ) ;
    abort( ) ;
  }
  state->seed = 1 ;
  return state ;
}

int random_number( random *state, int range ) {
  int result = state->seed % range ;
  state->seed = root * state->seed % period ;
  return result ;
}
```

The difference between this and the functional version of `random` is minimal. The functional version generated a new state, given the previous state. The imperative version overwrites the old state. In comparison with the previous C version of `random`, the state variable has been lifted from the module that implements `random` and is now passed on to the module using the random generator. The module `random` no longer contains state of its own. It has therefore a better interface, even though this interface is not purely functional. In other words, the new version of `random_number` has a side effect like the previous version. However, the new version side effects one of its arguments, which is visible if we look at only the module interface; the fact that the function `random_number` accepts a pointer as one of its arguments signals that it may (and in this case will) side effect the object pointed at by the pointer. The old version side effects a hidden state, invisible if we look only at the interface.

8.3.3 Scoping and life time of global and local variables

Using the technique of lifting state out of modules, global variables can almost always be avoided. However, in some cases, global variables are the appropriate solution. To explain this, it is important to elaborate on the lifetime and visibility (or scoping) of variables. The variables used in the first 7 chapters are all local variables. Local variables are declared in a block. A local variable is associated with a storage cell just before execution begins of the first statement in the block. The store is deallocated when the last statement of the block has been executed. The block is also the *scope* of the variable, where it can be seen and where it can be used.

Variables declared `static` live for as long as the program executes. They are created when the program starts and cease to exist when the program terminates. These variables are visible to all functions in a module. When a `static` variable is declared inside a function, the *scope* of this variable, the *visibility*, is restricted to the enclosing group of statements. Consider the function `counter` below:

```
int counter( void ) {
  static int count = 10 ;
  return count++ ;
}
```

The declaration `static int count = 10` creates a variable called `count`. The scope of count is restricted to this function, but the lifetime of count is that of a global variable. So `count` is created when the program starts; it is at that moment initialised to `10`, and then `count` is used by the function `counter` in subsequent calls. Note that the initialisation happens only during program startup, not on each function invocation. Thus, the function `counter` will return the values `10`, `11`, `12`, `13`, … on successive calls to `counter`.

C also supports variables that are visible to *any* module of the program. These variables are known as *external variables*. An external variable must be declared outside a function, just like a `static` variable, but without the keyword `static`:

```
int very_global_count = 10 ;
```

To make an external variable visible to other modules, it should also be declared in the header file of the module with the keyword `extern`:

```
extern int very_global_count ;
```

External variables can be read and written by any function in the program. Communication between functions via global variables should be avoided if possible. A good use of global variables is when data need to be broadcasted *once*. As an example, one can have a boolean `verbose` that indicates that every function in the whole program should print what it is doing.

A common mistake with the use of external global variables is to have two global variables with the same name in two separate modules of the program. The C compiler will not complain, but the linker will silently unify these two variables (as they share the same name space). Suppose that there are two modules and that each module has a variable `counter`:

```
/*C module 1 with counter*/
```

```
 int counter ;


/*C module 2 with counter*/
 int counter ;
```

These two counters are intended to be counters local to the module (they should have been declared `static`, but were not). On execution, the program will have only one variable `counter`, and both modules will share this variable. Needless to say, the results will be unexpected.

The scoping and lifetimes of the various types of variables are summarised in the table below:

Class	Declared	Scope	Lifetime
`auto`	Inside functions	Closest scope	Function
`static`	anywhere	Closest scope, but never more than one module	Program
`extern`	anywhere	Closest scope and the other modules	Program

Functions and variables share the same name spaces and the same declaration mechanism. That is, a function declared `static` is only visible to the module where it is declared, not in any other module. Functions that are not declared `static` can be used globally. Functions cannot be declared locally in a function.

8.4 Abstract Data Types

There is one important detail that can be improved on the modules shown earlier. In most cases, the header file exports the full details of the data type on which the module operates. As an example, the header file of the random number module contains a full declaration of the type `random`:

```
 typedef struct {
   int seed ;
 } random ;
```

This is undesirable as it exposes some of the internal workings of the module to its clients, while the clients are not supposed to know it. Each time the internal details of the type `random` change, the interface is changed as well, which will require all clients of the random module to be compiled again. Worse, spelling out the details invites programmers to *use* the internal details.

It is better to omit the details of the type from the interface. C allows this by means of an *incomplete* structure definition. An incomplete structure definition requires the struct to be named, as was necessary to define a recursive type (Chapter 6). This results in the following definition of `random`:

```
 typedef struct random_struct random ;
```

This specifies that the type random refers to a particular structure called random_struct, without revealing its contents. The only operations that can be performed on this type is to declare a pointer to it and to pass it around. The module interface becomes:

```
typedef struct random_struct random ;

extern random *random_init( void ) ;
extern int     random_number( random *state, int range ) ;
```

Now that the internal details of the structure declaration have been removed from the interface of the module, they must be specified in the implementation of the module. The implementation of the module becomes:

```
#include "random.h"

#define period 31
#define root 11

struct random_struct {
  int seed ;
} ;

random *random_init( void ) {
  random *state = malloc( sizeof( random ) ) ;
  if( state == NULL ) {
    printf( "random_init: no space\n" ) ;
    abort( ) ;
  }
  state->seed = 1 ;
  return state ;
}

int random_number( random *state, int range ) {
  int result = state->seed % range ;
  state->seed = root * state->seed % period ;
  return result ;
}
```

The declaration of the struct with the tag random_struct states that it has a single field, named seed. Note that there is no typedef; the type random was already defined in the header file.

One could, for example, change the random number generator so that it uses a better algorithm which maintains a bigger state in two integers by only changing the implementation part of the module:

```
struct random_struct {
  int state1 ;
  int state2 ;
} ;
```

The functions initialising the random number generator and calculating the random numbers have to be changed accordingly. The interface to the rest of the world remains the same.

Exercise 8.4 Give the SML version of the hidden state random number module.

What has been created is called an Abstract Data Type, or ADT. An ADT is a module that defines a type and a set of functions operating on this type. The structure of the type itself is invisible to the outside world. This means that the internal details of how the type is organised are hidden and can be changed when necessary. The only public information about the type are the functions operating on it. Note that no other module can operate on this type, as the internal structure is unknown.

Exercise 8.5 Sections 7.3.1 and 7.3.4, and Exercise 7.13 discussed three implementations of buffered streams: using a sliding queue, a shifting array and a cyclic array. Design an ADT that implements a buffered stream using either of these methods (the third implementation is the most efficient). Your ADT should have the following declarations in the header file:

```
typedef struct buffer Buffer ;

extern Buffer* b_create( FILE *input, int size );
extern bool b_advance( Buffer *b, int n ) ;
extern void b_read(    Buffer *b, char *s, int n ) ;
extern int  b_compare( Buffer *b, char *s, int n ) ;
extern void b_close(   Buffer *b ) ;
```

The first function creates the `Buffer` data structure, and reads the initial part, `b_advance` will advance the buffer `n` positions, when the end of file is encountered it will return `false`; `b_read` will copy `n` characters to the array `s`; `b_compare` will compare `n` characters with the array `s`; and `b_close` will close the buffer and the stream.

The ADT above defines `Buffer` as the structure itself, requiring an explicit `*` in each of the declarations. Alternatively, we could have declared:

```
typedef struct buffer *AltBuffer ;

AltBuffer b_create( FILE *input, int size );
bool      b_advance( AltBuffer b, int n ) ;
...
```

The disadvantage of this is that it is not clear to the user of the module that the buffer passed to `b_advance` is actually modified as a side effect. Explicitly declaring it as a pointer clarifies the intention of the functions. This was also the case with the module that generates random numbers; the function that generated a new number updated the state as a side effect. For an abstract data type that works without side effects, the type can be declared including the pointer.

Exercise 8.6 Design an abstract data type list of strings. Your ADT should have the following declarations in the header file:

```
typedef struct string_list *Slist ;

extern Slist slist_cons( char *s, Slist tail ) ;
extern char* slist_head( Slist x ) ;
extern Slist slist_tail( Slist x ) ;
extern Slist slist_append( Slist x, Slist y ) ;
extern void  slist_destroy( Slist x ) ;
```

The function `slist_append` should not have a side effect, it should create a new list. Decide whether to reuse the list `y` as the tail of the new list, or whether to make a copy of `y`.

Exercise 8.7 Section 5.7 presented a dynamic array. Implement a module which defines an abstract data type for a dynamic array storing integers. The module should at least provide functions for creating an array, changing the bounds, and accessing an element (with a bound check).

Note that ADTs have explicit operations for creating and destroying data structures, for example `b_create` and `b_close` for the buffered stream and `slist_cons`, `slist_append` and `slist_destroy` for the string list module. The explicit destruction clutters the code, but it is essential as the memory would otherwise fill up with unused data structures. Many object oriented languages have complete built-in support for data abstractions, including a garbage collector. This is one of the reasons why many programmers prefer to use an object oriented programming language.

8.5 Polymorphic typing

A module supports a number of important concepts. One of these concepts is that of *reusability*. The random generator shown before returns random numbers in a certain range. The range is an argument of the function. It would not be a good idea to write another random generator for each range. In the same way as functions are permitted to be parametrised over their argument values, it is also useful to allow *types* to be parametrised.

As an example, consider lists: a list of characters was defined in Chapter 6, a list of strings was defined in Exercise 8.6. If the user needs a list of integers, neither of these modules can be reused. The solution to this problem in functional languages is to use polymorphism, implemented through a *type parameter*. The SML definitions of a character list and a general list are:

```
datatype char_list            datatype  a list
  = Nil                         = Nil
  | Cons of (char*char_list);   | Cons of ( a* a list);
```

The a stands for the type parameter ('any type will do'), and all SML functions shown in Chapter 6 will work with this type, a list. The C typing mechanism does not support parametrised types, but it does have a loophole that allows passing arguments of unspecified types. This means that polymorphism can be efficiently supported in C, but without the type security offered by SML. In Chapter 4, where partially applied functions were discussed, a variable of the type void * was used to point to some arbitrary type. In C, polymorphic lists can be constructed using this void pointer:

```
typedef struct list_struct {
  void                 *list_head ;
  struct list_struct   *list_tail ;
} *list ;
```

This type defines a polymorphic list. The first member of the list is a pointer to the data, and the second element is a pointer to the next element of the list. With a little care we can rebuild the full complement of list processing functions on the basis of this new definition. To create a cons cell, the types occurring in the C function cons have to be changed consistently from char to void * as follows:

```
list cons( void *head, list tail ) {
  list l = malloc( sizeof( struct list_struct ) ) ;
  l->list_head = head ;
  l->list_tail = tail ;
  return l ;
}
```

With this function, we can create a list of characters as before. Because the type of the first argument of the new polymorphic cons function is void *, we must pass a pointer to the character, rather than the character itself. To implement a function list_hi as in Chapter 6, we might be tempted to write the following syntactically incorrect C expression:

```
  cons( & H , cons( & i , NULL ) ) /* ILLEGAL C */
```

Expressions such as & H are illegal, as C forbids taking the address of a constant. One might try to be clever and use the following syntactically correct code instead:

```
list list_hi( void ) {
  char H =  H  ;
  char i =  i  ;
  list hi = cons( &H, cons( &i, NULL ) ) ; /* INCORRECT */
  return hi ;
}
```

This version of list_hi is syntactically correct: it is legal to take the address of a variable and use it as a pointer. However, the resulting function is incorrect, for the result list that is created has two dangling pointers; the address of the variables H and i point to the local variables of list_hi, which will disappear as soon as the return statement has been executed. Thus the problem is that the lifetime of the list exceeds that of the variables H and i.

The only correct way to create the two character list is by allocating both the cons-cells and the characters on the heap. Ideally, the allocation of the memory is performed by `cons`, so that one function captures the allocation of both the cell and the area for the data:

```
list cons( void *head, list tail ) {
  list l  = malloc( sizeof( struct list_struct ) ) ;
  void *copy_of_head = malloc( /*C size of the data*/ ) ;
  /*C copy the data from *head to *copy_of_head*/
  l->list_head = copy_of_head ;
  l->list_tail = tail ;
  return l ;
}
```

There are two problems to be resolved here. Firstly how much store needs to be allocated for the data? Secondly, how can we copy the data into the allocated store? The first problem cannot be solved by the function `cons`, since the size of the data pointed to by `head` is unknown to `cons`. Therefore, the size of the data must be passed to `cons` as a third argument. The second problem is resolved by using a standard function of the C library, called `memcpy`. Given a source pointer (`head`), a destination pointer (`copy_of_head`), and the size of the object involved, the following call will copy the data from `head` to `copy_of_head`.

```
memcpy( copy_of_head, head, /*C size of the data*/ ) ;
```

Therefore, the complete code for the polymorphic `cons` reads:

```
list cons( void *head, list tail, int size ) {
  list l  = malloc( sizeof( struct list_struct ) ) ;
  void *copy_of_head = malloc( size ) ;
  memcpy( copy_of_head, head, size ) ;
  l->list_head = copy_of_head ;
  l->list_tail = tail ;
  return l ;
}
```

Here is an example of its use, which builds a list of lists. The C equivalent of the SML list `[ ["H","i"],["H","o"] ]` would be:

```
#define char_cons(c,cs) cons( c, cs, sizeof( char ) )
#define list_cons(c,cs) cons( c, cs, sizeof( list ) )

list list_of_list( void ) {
  char H     =  H  ;
  char i     =  i  ;
  char o     =  o  ;
  list hi    = char_cons( &H, char_cons( &i, NULL ) ) ;
  list ho    = char_cons( &H, char_cons( &o, NULL ) ) ;
  list hi_ho = list_cons( &hi,list_cons( &ho,NULL ) ) ;
  return hi_ho ;
}
```

The access functions head and tail are readily generalised to polymorphic lists:

```
  void * head( list l ) {       list tail( list l ) {
    if( l == NULL ) {             if( l == NULL ) {
      abort() ;                     abort() ;
    }                             }
    return l->list_head ;         return l->list_tail ;
  }                             }
```

Exercise 8.8 Generalise the efficient iterative versions of functions length and nth to work with polymorphic lists.

We are now equipped to try something more demanding. Here is the polymorphic version of extra_filter. The function header contains three occurrences of void *, the first and the last correspond to the extra argument handling mechanism, and the second occurrence corresponds to the element type of the lists that we are manipulating. If we cannot see the difference between these types, then the C compiler cannot see the difference either. This means that any form of type security has vanished by the switch to polymorphic lists.

```
list extra_filter( bool (*pred)( void *, void * ),
                   void * arg, list x_xs, int size ) {
  if ( x_xs == NULL ) {
    return NULL ;
  } else {
    void * x = head( x_xs ) ;
    list xs = tail( x_xs ) ;
    if( pred( arg, x ) ) {
      return cons(x, extra_filter(pred,arg,xs,size), size);
    } else {
      return extra_filter( pred, arg, xs, size ) ;
    }
  }
}
```

Exercise 8.9 Generalise the open list version of append that makes advanced use of pointers (See Section 6.5.3).

To complete this polymorphic list module, the function prototypes and the type list may be specified in a header file as follows:

```
#ifndef LIST_H
#define LIST_H

typedef struct list_struct *list ;
```

```
extern list  cons( void *head, list tail, int size ) ;

extern int   length( list x_xs ) ;

extern void *head( list l ) ;
extern void *nth( list x_xs, int n ) ;
extern list  tail( list l ) ;
extern list  append( list xs, list ys, int size ) ;
extern list  extra_filter( bool (*pred)( void *, void * ),
                           void * arg, list x_xs,
                           int size ) ;
#endif /* LIST_H */
```

The type `list` is a pointer to an incomplete structure, since the structure `list_struct` is not defined in this header file. The type that is supplied to `cons` is `void *`, which is also the type returned by the functions `head` and `nth`. As explained at the end of Chapter 4, the C compiler is not fussy about the use of `void *`. The compiler will neither require a list to contain elements of the same type, nor verify that the type of something that is put in the list matches the type that is coming out of the list. This is in contrast with the security offered by a strongly typed polymorphic language. In C, we can add type security at runtime, but we have to pay the costs of a longer run time. Alternatively, we can implement a set of functions for each type which does offer type safety, but which is undesirable from a software engineering point of view.

8.6 Summary

The following C constructs were introduced:

Header and implementation files In C, the interface of a module is defined in the header (`.h`) file, and the implementation in the source (`.c`) file.

Preprocessor The C preprocessor processes the text of a C program before the actual compilation. The three most important directives are `#define`, `#include` and `#ifndef`. The `#define` directive will cause a macro name to be (literally) replaced with a sequence of characters. The `#include` directive literally includes a file (typically a header file of a module) at that place in the source. The `#ifndef` directive leaves text out if a macro has been defined:

```
#ifndef i

#include "filename"
#define m(x,y ...)  t

#endif /* i */
```

Here, i and m are identifiers, x and y are parameters of the macro being defined, and t is the replacement text for the macro. If there are no parameters, the parentheses should be omitted. The replacement text and parameters are usually enclosed in parentheses otherwise macro substitution would end in chaos.

Global variables Global variables are variables that are declared outside the scope of a function. Every function can read or write them. They may even be visible outside modules, if another module, either deliberately or accidentally, uses a variable with the same name.

Static functions and variables Static variables have a life time as long as global variables, but they are not visible outside a module or function. Static functions are functions that are private to a module. Variables and functions are declared static by prepending their declaration with the keyword `static`.

Separate compilation The module system of C facilitates separate compilation. When the interface of a module changes, all modules using this changed interface need recompilation. The programming environment will almost always supply the necessary tools. The UNIX programming environment provides the program `make` for this purpose.

Incomplete structure definition None of the details of a structure need to be specified in a header file. The header can just specify a type:

```
typedef struct s t ;
```

Provided that the implementation of the module specifies the structure completely:

```
struct s {
  ...
} ;
```

The clients importing the header can only pass values of type t around, only the module where the implementation is defined knows the details of t.

Modules are fundamental for good software engineering:

- Modularisation is essential to structure large programs. In C, modules are supported by the C preprocessor and are not as advanced as modules found in other languages. In particular, modules can be imported twice (leading to compilation errors), and one cannot selectively import parts of a module.

- Only those items that are really needed by the outside world should be exported by defining such items in an interface. Types and internal functions should be confined to the implementation of the module.

- An Abstract Data Type, ADT, specifies a type and a set of operations. The internal workings of an ADT are shielded off by the module. This localises design and maintenance effort, and gives modules that are easily reused.

- A variable should be declared so that the scope is as small as possible, and the lifetime is as short as possible. Avoid global variables, instead store the state in a structure, and pass a pointer explicitly to the functions operating on it.
- Modules using polymorphic data types can be built in C using the `void` pointer. They are only type safe if the programmer invests in runtime checks. Efficient polymorphic data types in C can only be built in a type unsafe way.

8.7 Further exercises

Exercise 8.10 Functions with a small domain are often implemented using a *memo* table. For each value in the domain, one would keep the corresponding function value. Instead of computing the value, it is merely looked up. The factorial function is a good example of a function with a small domain. Here is a table of some of its values:

n	$n!$
1	1
2	2
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
14	87,178,291,200

The function values grow so quickly that, with 32-bit arithmetic, overflow occurs for $13!$. Thus a table of 12 values is sufficient to maintain the entire domain and range of the factorial function for 32-bit arithmetic.

Write a C function that uses an array as a memo table. Your function should compute the appropriate values just once, and it should not compute more values than strictly necessary. When the argument is beyond the domain, your function should abort.

Exercise 8.11 Exercise 8.7 required the implementation of a dynamic array of integers. Define a module that implements a polymorphic dynamic array. Upon creation of the array, the size of the data elements is passed, subsequent functions (for indexing and bound changes) do not need extra parameters.

Exercise 8.12 Write a higher order polymorphic version of the quicksort function from Chapter 7 that takes a comparison function as an argument so that it can sort an array of any data type.

Exercise 8.13 A snoc list is an alternative representation of a list, which stores the head and the tail of a list in reverse order. Here is the SML definition of the snoc list:

```
datatype  a snoc_list = Snoc of ( a snoc_list *  a)
                      | Nil ;
```

Three numbers 1, 2 and 3 could be gathered in a snoc list as follows:

```
Snoc(Snoc(Snoc(Nil,1),2),3) ;
```

An equivalent ordinary list representation would be:

```
Cons(1,Cons(2,Cons(3,Nil))) ;
```

This uses the following definition of a polymorphic list:

```
datatype  a list = Nil
                 | Cons of ( a *  a list) ;
```

Note that in both representations the order of the elements is always 1, 2, 3. Here is the C data structure that we will be using to represent snoc lists:

```
typedef struct snoc_struct {
  void                 * snoc_tail ;
  struct snoc_struct   * snoc_head ;
} * snoc_list ;
```

(a) Define the C functions `snoc`, `head` and `tail` similar to those at the beginning of this chapter 6, but associated with the type `snoc_list`.

(b) Create a higher order C function `sprint` with the following prototype:

```
void sprint(void (*print) ( void * ), snoc_list l ) ;
```

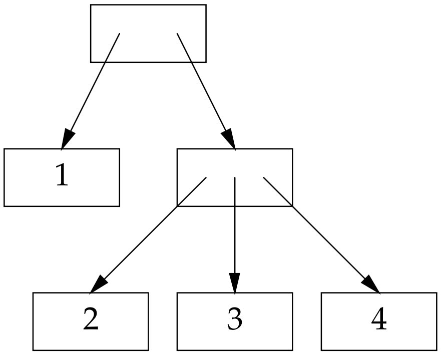
The purpose of this function is to print all elements of the snoc list and in the correct order. Use a comma to separate the elements when printed.

(c) Write a main function to test your snoc list type and the associated functions.

Exercise 8.14 An n-ary tree is a generalisation of a binary tree, which has n branches at each interior node rather than just two. The following SML data structure represents an n-ary tree with integer values at the leafs. Here we have chosen to use the snoc list from the previous exercise.

```
datatype ntree = Br of (ntree snoc_list)
               | Lf of int ;
```

Here is a sample `ntree`:



In an SML program, the sample `ntree` could be created as follows:

```
val sample
  = let
       val l234 = Snoc( Snoc( Snoc( Nil, Lf 2 ),
                                    Lf 3
                                  ),
                              Lf 4
                            )
    in
       Br (Snoc( Snoc( Nil, Lf 1 ),
                 Br  ( l234 )
               )
          )
    end ;
```

(a) Give an equivalent type definition of the SML `ntree` above in C. Use the `snoc_list` as defined in the previous exercise.

(b) Write a C function `nlf` to create a leaf from a given integer key value. Also write a C function `nbr` to construct a branch from a given snoc list of n-ary trees.

(c) Write a function `nprint` with this prototype:

```
void nprint( ntree t ) ;
```

Your function should print an `ntree` as follows:

1. For each leaf print the key value kept in the leaf.
2. Print all elements of a snoc list of branches enclosed in parentheses. Use the function `sprint` from the previous exercise.

The sample `ntree` above should thus be printed as: `(1,(2,3,4))`.

(d) Write a C `main` program to create the sample `ntree` above and print it using `nprint`.

Exercise 8.15 In Exercises 8.13 and 8.14 we have created a data structure, and sets of functions operating on the data structure. Package each data structure with its associated functions into a separate C module.

(a) Create a C header file and a C implementation for the snoc list data type and the functions `snoc`, `head`, `tail` and `sprint`.

(b) Create a C header file and a C implementation for the `ntree` data type and the functions `nlf`, `nbr` and `nprint`.

(c) Write a program to test that your modules are usable.

Chapter 9

Three case studies in graphics

In this chapter we will make three case studies to put in practice the principles and techniques introduced before. The first case study is a small program to draw a fractal using X-windows. The second study is a device independent graphics driver, using X-windows, PostScript, and possibly some other graphics output devices. The third case study is an interpreter for an elementary graphics description language, called `gp`. The `gp` interpreter uses the device independent graphics driver of the second case study.

The three case studies are increasingly more difficult, and the amount of detail left to the reader also increases. The first case study spells out the data structures and algorithms to be used. The second case study gives the data structures and some code. The third case study only suggests the data structures that might be used. The reader is encouraged to work out these case studies.

All programs use X-windows to render the graphics. Being able to use the X-windows library is at the same time a useful skill and a good test of one's programming abilities. We choose the X-windows system because it is widely available; it runs on PC's, workstations and the Macintosh; and it is public domain software. The disadvantage of choosing the X windows system is that it is a complex system, especially for the beginning programmer. For this reason, we use only an essential subset of X. The interested reader is referred to the X-windows programming manuals for complete details [17].

9.1 First case study: drawing a fractal

The *Mandelbrot set* is a fractal, a shape that is irregular, no matter how far it is magnified. It is also a pretty picture. We will write a program to render such a picture on the screen. To perform the actual drawing on the screen, we use the X-windows toolkit.

We divide the problem into two parts. First we will define the Mandelbrot set and define a module that computes whether a point is part of the set or not. Then a module is defined that displays the points on the screen. These two modules constitute a relatively complicated program which will be developed by refining an initial, inefficient version to a final efficient version.

9.1.1 Defining the Mandelbrot set

The Mandelbrot set is defined as follows. Given a point c in the complex plane, the recurrence relation below defines a series z of complex numbers $z_0, z_1, \ldots$:

$$\begin{aligned} z_j, c &: \mathbb{C} \\ z_0 &= 0 \\ z_{j+1} &= z_j^2 - c, \text{ if } j > 0 \end{aligned}$$

For some choices of the point c, the series z diverges. Take the following two examples. If $c = 4$, the series is:

$$z = 0, -4, 12, 140, 19596, \ldots$$

For another choice, $c = 1 - i$, the series is:

$$z = 0, -1+i, -1-i, -1+3i, -9+7i, \ldots$$

An example value of c for which the series does not diverge is $c = 0.1$:

$$z = 0, -0.1, -0.09, -0.0919, -0.09155439, \ldots$$

This series converges to $0.5 - \sqrt{0.35}$. In general, if for a given c, there is a k such that $|z_k| > 1$, the series z diverges. The Mandelbrot set is defined as all points c for which the series z converges. If the set of points is plotted in the complex plane, a pretty picture results.

The program that we are going to develop will calculate an approximation to the Mandelbrot set. To decide whether a point belongs to the set, it would be necessary to compute all points of the series z, which is an infinite task. We are going to approximate this by calculating only a fixed length prefix of each series. If this prefix does not diverge, then we assume that the infinite series does not diverge. The approximation can be improved by increasing the number of steps, but at the expense of extra run time.

The solution to the Mandelbrot problem is first given in SML. We will also need the complex arithmetic module from Chapter 8. The function that decides whether a point belongs to the Mandelbrot set is:

```
(* in_Mandelbrot : Complex.complex -> bool *)
fun in_Mandelbrot c
    = let
          open Complex
          val MAXSTEPS = 31
          fun step i z
              = let
                    val z  = complex_sub (complex_multiply z z) c
                in
                    if complex_distance z > 1.0
                       then false
```

```
                        else if i > MAXSTEPS
                                then true
                                else step (i+1) z
                  end
         in
            step 0 (0.0,0.0)
         end ;
```

The corresponding C implementation needs to import the complex number module, and the boolean module (both from Chapter 8):

```
#include <stdio.h>
#include "mandelbrot.h"
#include "complex.h"
#include "bool.h"

#define MAXSTEPS 31

bool in_Mandelbrot( complex x ) {
  complex z = {0,0} ;
  int i = 0 ;

  while( true ) {
    if( complex_distance( z ) > 1.0 ) {
      return false ;
    } else if( i > MAXSTEPS ) {
      return true ;
    }
    z = complex_multiply( z, z ) ;
    z = complex_sub( z, x ) ;
    i++ ;
  }
}
```

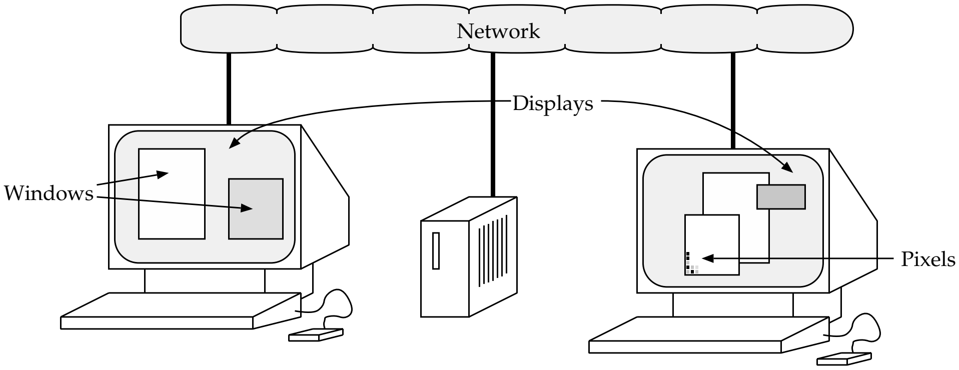
The interface of this module to the external world is:

```
#ifndef MANDELBROT_H
#define MANDELBROT_H

#include "bool.h"
#include "complex.h"
extern bool in_Mandelbrot( complex x ) ;

#endif
```

The function `in_Mandelbrot` is exported, but `MAXSTEPS` is not exported. This hides the details of how accurate the approximation is, so that the approximation can be changed without having to change functions that use `in_Mandelbrot`.

9.1.2 Drawing the fractal on the screen

Before we show which functions the X-windows library provides to draw something on the screen, we give a short overview of the X-windows package. The figure below shows a possible environment where X-windows can be used.



This particular configuration shows three computers, two of which have a *display*. The computers are connected via a network. An X application can run on any computer (whether it has a screen or not) and can use any number of displays on the network. As an example, a multi user game could run on the computer in the middle and use the two displays connected to systems on the left and right. Simple applications, like an editor, will often use only one display and will execute on the machine connected to that display.

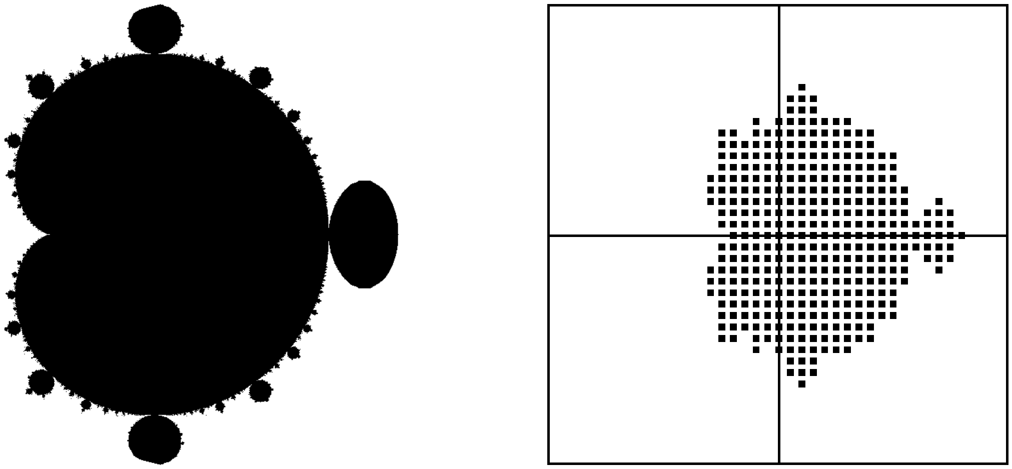
Within a display, an X-application distinguishes one or more *windows* that it controls. Most applications will use a single window, but an application can control an arbitrary number of windows. Within a window, the application program can do whatever it wants; the X-server will not allow an application to write outside its window to avoid destroying other windows, nor will it allow writing in parts of a window that are obscured by other windows.

Each window of an X application is composed of a grid of dots, or *pixels*. Each dot has a specific colour. On a monochrome display, a dot can be 'black' or 'white', but most modern computers have a colour screen, allowing a pixel to have any of a large range of colours. Each pixel on the grid is addressed with a coordinate. The coordinate system is slightly unusual: the origin is in the top left hand corner, the X coordinates run from left to right, and the Y coordinates run *top-down*. So the top left hand pixel of a window is the pixel with coordinates $(0, 0)$, the next one down is pixel $(0, 1)$, and so on. The X library provides functions, for example, to fill a rectangular area of dots with a colour, or to render some characters.

Given that an X application may use multiple displays and windows, the specific window and display to be used must be specified when performing some graphics operation. This information is not stored implicitly in the library, but must be explicitly passed by the program. This might be confusing at first because many applications use only one display and one window, but it makes the library more general. The X designers could have stored a "Current display" and "Current window" in global variables to shorten the argument list, but have (in our opinion rightly) chosen to avoid the global state.

In addition to specifying which display and window to use, all functions using graphics capabilities must indicate how the graphics operations are to be performed. This gives, for example, the colour and font to use when drawing a string of text. This is passed via an object known as a *graphics context*. Again, the designers of the X-windows system have chosen to pass this explicitly, as opposed to maintaining a current font or current colour in a global variable. A graphic context can be dynamically created and destroyed, and there are functions available to modify certain fields in a context.

Now that we know the morphology of a window in the X-windows system, it has to be decided how to draw the Mandelbrot set on the screen. The module in the previous section defines, for any point in the complex plane, whether the point is part of the (approximated) Mandelbrot set. In a window, we can only display part of the complex plane, and we can only show it with a finite resolution, given by the number of pixels. The figure below shows the Mandelbrot set drawn with a high resolution (left) and a window (right) from $-1.25 - 1.25i$ to $1.25 + 1.25i$, with 20×20 pixels on this window.



To display the Mandelbrot set, a mapping must be devised from the (X, Y) integer coordinates of the window to the complex plane. Assuming that the window measures $W \times H$ pixels, with the middle of the window on position $x + yi$, and showing an area of the complex plane with size $w \times hi$, then the complex number $c = r + mi$ is related to X and Y as follows:

$$\begin{array}{rcl}
X, Y, W, H & : & \mathbb{N} \\
r, m, x, y, w, h & : & \mathbb{R} \\
r & = & x + w\dfrac{X}{W} - \dfrac{w}{2} \\
m & = & y + h\dfrac{Y}{H} - \dfrac{h}{2}
\end{array}$$

The C implementation of this specification is shown below. The identifiers have been given slightly more descriptive names (`WIDTH` $= W$, `HEIGHT` $= H$, `width` $= w$, and `height` $= h$):

```
static
```

```
complex window_to_complex( int X, int Y,
                           int WIDTH, int HEIGHT,
                           double x, double y,
                           double width, double height ) {
  complex c ;
  c.re = x + width  * X / WIDTH  - width/2 ;
  c.im = y + height * Y / HEIGHT - height/2 ;
  return c ;
}
```

The expression `width * X / WIDTH` that is used to scale `X` deserves attention as the expression is less innocent than it looks at first sight. The variables `X` and `WIDTH` are integers, while `width` is a floating point number. Suppose the compiler would have interpreted the expression as follows:

```
width * ( X / WIDTH )
```

In this case the division operator would be interpreted as an *integer* division. Because `X < WIDTH`, this division would always return 0, which is not the desired result. Because the `*` and `/` operators have equal precedence and are left associative, the parentheses are inserted properly. The program will first multiply `width` with `X`, and then perform a *floating point* division with `WIDTH` (as the result of `width * X` is a floating point number). It would be better to insert an explicit type coercion to ensure that the `/` is a floating point division, for example:

```
width * X / (double) WIDTH
```

Using the function `window_to_complex`, a point on the screen can be translated to a point on the complex plane, and with the function `in_Mandelbrot`, we can test whether this point belongs to the Mandelbrot set. All that is needed now is the function that draws a pixel on the window. For this purpose, we use the X-windows function `XFillRectangle`, which fills a rectangular area on the screen. Seven arguments must be passed to this function: on which display and which window is the rectangle to appear, which graphics context is to be used to draw the rectangle, and finally the description of the rectangle. The function `XFillRectangle` has the following prototype:

```
void XFillRectangle( Display *d, Window w, GC gc,
                     int x, int y,
                     int width, int height ) ;
```

The rectangle is uniquely determined by the coordinates of its upper left corner (`x` and `y`) and its `width` and `height`, all measured in pixels. The function `XFillRectangle` can be embedded in a function that draws one pixel on a window. This effectively packages the required functionality and provides it with a nicer interface.

```
static
void draw_pixel( Display *theDisplay, Window theWindow,
                 long colour,
                 int x, int y ) {
  GC gc = XCreateGC( theDisplay, theWindow, 0, NULL ) ;
```

```
    XSetForeground( theDisplay,            gc, colour ) ;
    XFillRectangle( theDisplay, theWindow, gc, x, y, 1, 1);
    XFreeGC( theDisplay, gc ) ;           /* Inefficient */
  }
```

The function draw_pixel has five arguments: the display, the window, the colour that we wish to use, and the x- and y-coordinates of the pixel to be drawn. Before we can use a graphics context, it must be created. This is taken care of by the function XCreateGC. The colour of the graphics context is modified by calling XSetForeground. After we have finished with the graphics context, it must be deallocated, which is accomplished by the function XFreeGC. This is an inefficient process; we will come back to this later.

The type of colour is long, which is an abbreviation for a long integer. The type of theDisplay is a pointer, while theWindow, and gc are not pointers (the types are respectively Display *, Window and GC). To make it more confusing, the type GC is actually a pointer type, which explains why the function XSetForeground can modify the contents of the graphics context passed to it. It is an unfortunate fact of life that, when using real world libraries, some functions require explicit pointer types, while other types are pointers implicitly.

Using draw_pixel it is now possible to implement a function that draws the Mandelbrot set. The function draw_Mandelbrot below iterates over all pixels of the window and draws each pixel accordingly:

```
  static
  void draw_Mandelbrot( Display *theDisplay, Window theWindow,
                        long black, long white,
                        int WIDTH, int HEIGHT,
                        double x, double y,
                        double width, double height) {
    int X, Y ;
    complex c ;
    for( X=0 ; X<WIDTH ; X++ ) {
      for( Y=0 ; Y<HEIGHT ; Y++ ) {
        c = window_to_complex( X,Y,WIDTH,HEIGHT,
                               x,y,width,height ) ;
        if( in_Mandelbrot( c ) ) {
          draw_pixel( theDisplay, theWindow, black, X, Y ) ;
        } else {
          draw_pixel( theDisplay, theWindow, white, X, Y ) ;
        }
      }
    }
  }
```

The argument list of the function draw_Mandelbrot has now become rather long. This is not desirable, both because the program code is obfuscated and because the code might execute slower. This issue will be resolved after the program has been completed.

The final part of the program is the `main` function below. It performs three tasks. First the X-windows library is initialised and queried for the display and window. After that, the user is queried which part of the Mandelbrot set should be shown, and finally the Mandelbrot is drawn.

```
#define WIDTH  100
#define HEIGHT 100

int main( int argc , char *argv[] ) {
  XtAppContext context ;
  int          theScreen ;
  Display      *theDisplay ;
  Window       theWindow ;
  long         white, black ;
  double       x, y, width, height ;
  Widget widget = XtVaAppInitialize( &context, "XMandel",
                      NULL, 0, &argc, argv, NULL, NULL ) ;
  XtVaSetValues( widget, XtNheight, HEIGHT,
                         XtNwidth,  WIDTH, NULL ) ;
  XtRealizeWidget( widget ) ;
  theDisplay = XtDisplay( widget ) ;
  theWindow  = XtWindow( widget ) ;
  theScreen  = DefaultScreen( theDisplay ) ;
  white      = WhitePixel( theDisplay, theScreen ) ;
  black      = BlackPixel( theDisplay, theScreen ) ;
  printf("Enter originx, y, width and height: " ) ;
  if( scanf( "%lf%lf%lf%lf", &x,&y,&width,&height ) != 4 ) {
    printf("Sorry, cannot read these numbers\n" ) ;
  } else {
    draw_Mandelbrot( theDisplay, theWindow, black, white,
                     WIDTH, HEIGHT, x, y, width, height ) ;
    XtAppMainLoop( context ) ;
  }
  return 0 ;
}
```

The function `main` does not have a `void` argument list, but instead, the argument count and argument vector described in Chapter 5. The arguments are required by the function that initialises the X-windows library.

All X functions seen so far have had names that are composed of an "X" and the function operation. These functions come from the basic X-windows library. The function `XtVaAppInitialize` is part of the X-toolkit, a set of functions provided on top of basic X-windows that simplifies the use of X-windows. All functions from the X-toolkit have a name that start with '`Xt`'.

The function `XtVaAppInitialize` performs several tasks. The result is that it returns a *widget*, one of the abstractions provided by the X-toolkit. A widget embodies a window and the functions that manage the window. We have to skip over many details here. Example widgets are scroll-bars, buttons, and text editors. A typical X-application consists of a hierarchy of widgets (widgets can be composed of other widgets), but as we hardly use any feature of widgets in this trivial example, we do not discuss it. The interested reader is referred to the X-windows manuals [17].

We apply the function `XtVaSetValues` to set the height and the width of the widget and draw the widget by calling the function `XtRealizeWidget`. This creates a window on the display.

To find out what the identification of the display and window are, the X-toolkit provides the functions `XtDisplay` and `XtWindow`. Given the display, we use a further three functions to determine how the colours `black` and `white` are denoted on this particular display. All arguments are then passed to `draw_Mandelbrot` for drawing the Mandelbrot graphic. Good test values for the program are:

```
0 0 2.5 2.5
```

These values correspond to a square area of the complex plane with lower left coordinate $-1.25 - 1.25i$ and upper right coordinate $1.25 + 1.25i$.

After the Mandelbrot graphic is drawn, the program should not terminate directly, as it would cause the window to disappear. One way of solving this would be to include a non terminating statement at the end, for example:

```
while( true ) { ... }
```

However, this would cause the program to use the processor continuously, which is a waste of resources. The X-toolkit provides a library function that will prevent the application program from terminating, `XtAppMainLoop`. This function does something more important: it also handles all *events* (mouse clicks, key strokes) that are being sent to the application program. The purpose of the main loop is shown later in Section 9.1.4.

To compile the program, the four functions above must be completed with a list of include directives that import the right header files:

```
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include "complex.h"
#include "mandelbrot.h"
```

Furthermore, a `Makefile` is needed, and during the linking stage of the program, we need to tell the compiler to use the X11, X-toolkit, and mathematical libraries:

```
OBJECTS= main.o complex.o mandelbrot.o

mandelbrot: $(OBJECTS)
        $(CC) -o mandelbrot $(OBJECTS) -lXt -lX11 -lm

depend:
        makedepend $(CFLAGS) main.c complex.c mandelbrot.c
```

The exact calling syntax for the compiler and linker is machine dependent. On some machines the user will have to add include and or library path options to the compiler. For example, the compiler might require an option `-I/usr/local/X11/include`, and the final linking stage might need the option `-L/usr/lib/X11`. The local documentation may give more details on how X11 is installed on the machine.

Compiling and executing this program will draw a small Mandelbrot graphic. Stopping the program in its present form is a bit of a problem, as the program will wait indefinitely. Interrupting the program or destroying the window are the only solutions; we will resolve this problem later.

Depending on the speed of your computer system, you can choose to increase the size of the window (`WIDTH` and `HEIGHT`) or the accuracy of the approximation to the Mandelbrot set (by increasing `MAXSTEPS`).

Exercise 9.1 What would happen if the C-preprocessor directives that define `WIDTH` and `HEIGHT` were defined *before* the first function of this module?

9.1.3 Shortening the argument lists

Each of the three functions `window_to_complex`, `draw_pixel`, and `draw_Mandelbrot` has a large number of arguments. This is undesirable, as it does not look pretty, it is too much work to type, and it sometimes obscures the purpose of a function. Here are the prototypes of the relevant functions gathered together:

```
complex window_to_complex( int X, int Y,
                           int WIDTH, int HEIGHT,
                           double x, double y,
                           double width, double height ) ;

void draw_pixel( Display *theDisplay, Window theWindow,
                 long colour,
                 int x, int y ) ;

void draw_Mandelbrot( Display *theDisplay, Window theWindow,
                      long black, long white,
                      int WIDTH, int HEIGHT,
                      double x, double y,
                      double width, double height) ;
```

Some of these arguments are unnecessary, since they are constant. As an example, the variable `theDisplay` is a constant: it is set once (in the `main` program) and used afterwards in many functions. As this application uses only one display, we can choose to make `theDisplay` a `static` variable by declaring it before the first function of the module as follows:

```
static Display *theDisplay ;
```

Similarly, theWindow, black, and white can be defined as global constants (thereby restricting our application to using at most one window). The variables WIDTH and HEIGHT are proper constants, defined before main. If we move these definitions forward, we can use them in window_to_complex. This process results in a list of global variables and constants that need to be declared before the functions:

```
static Display *theDisplay ;
static Window   theWindow ;
static long     black, white ;

#define WIDTH   100
#define HEIGHT  100
```

The function window_to_complex does not need the HEIGHT and WIDTH arguments anymore. Its definition can be simplified to the following:

```
static
complex window_to_complex( int X, int Y,
                           double x, double y,
                           double width, double height ) {
  complex c ;
  c.re = x + width  * X / WIDTH  - width/2 ;
  c.im = y + height * Y / HEIGHT - height/2 ;
  return c ;
}
```

The other functions, draw_pixel and draw_Mandelbrot, can be simplified in a similar fashion. Before showing the new code we will remove an inefficiency in the original code. Each time that the function draw_pixel is called, a graphics context is created and destroyed. It is more efficient to create this graphics context only once in the function draw_Mandelbrot and to reuse it in draw_pixel a number of times. The graphics context must be destroyed in draw_Mandelbrot just prior to leaving the function. This optimisation results in the following code:

```
static
void draw_pixel( GC gc, long colour, int x, int y ) {
  XSetForeground( theDisplay,            gc, colour ) ;
  XFillRectangle( theDisplay, theWindow, gc, x, y, 1, 1);
}


static
void draw_Mandelbrot( double x, double y,
                      double width, double height) {
  int X, Y ;
  GC gc = XCreateGC( theDisplay, theWindow, 0, NULL ) ;
  for( X=0 ; X<WIDTH ; X++ ) {
    for( Y=0 ; Y<HEIGHT ; Y++ ) {
      if( in_Mandelbrot( window_to_complex( X,Y,x,y,
                                     width,height ) ) ) {
        draw_pixel( gc, black, X, Y ) ;
      } else {
        draw_pixel( gc, white, X, Y ) ;
      }
    }
  }
  XFreeGC( theDisplay, gc ) ;
}
```

The function `main` must now be changed, such that instead of creating *local* variables to store, for example, the display, appropriate global variables are initialised.

```
 int main( int argc , char *argv[] ) {
   XtAppContext context ;
   int      theScreen ;
   double   x, y, width, height ;
   Widget widget = XtVaAppInitialize( &context, "XMandel",
                       NULL, 0, &argc, argv, NULL, NULL ) ;
   XtVaSetValues( widget, XtNheight, HEIGHT,
                          XtNwidth,  WIDTH, NULL ) ;
   XtRealizeWidget( widget ) ;
   theDisplay = XtDisplay( widget ) ;
   theWindow  = XtWindow( widget ) ;
   theScreen  = DefaultScreen( theDisplay ) ;
   white      = WhitePixel( theDisplay, theScreen ) ;
   black      = BlackPixel( theDisplay, theScreen ) ;
   printf("Enter originx, y, width and height: " ) ;
   if( scanf( "%lf%lf%lf%lf", &x,&y,&width,&height ) != 4 ) {
     printf("Sorry, cannot read these numbers\n" ) ;
   } else {
     draw_Mandelbrot( x, y, width, height ) ;
     XtAppMainLoop( context ) ;
   }
   return 0 ;
 }
```

The global variables introduced are assigned exactly once. They are used neither to exchange data, nor to implicitly remember state between function calls, but they contain globally constant values. The resulting changes to the module will require the main module to be recompiled. The other modules (`Mandelbrot`, `bool` and `complex`) have not changed.

9.1.4 Handling events

The program as it has been developed so far has two problems. Firstly, the program cannot be stopped in an elegant way: the user has either to abort the computation, or to destroy the window. Secondly, the program does not properly redraw the window if something happens to it, for example if the window becomes obscured by another window, or when the window is closed and reopened. Both problems can be solved if the *events* of the X application are handled properly.

Each time something happens to an application program a so called *event* is posted to the application. The X-library handles these events, and the programmer can instruct the X-library to call specific functions when certain events are received.

As an example, we can define a function that is going to handle the Expose event. An Expose event is posted to the application when (a part of) a window has become visible. To receive the event, the X-library must be informed that we wish to receive information about the event:

```
XtAddEventHandler( widget, ExposureMask, false,
                   draw_Mandelbrot, NULL ) ;
```

Calling this function from the main program will cause the function draw_Mandelbrot to be called each time that part of the window is exposed. This is a good starting point, but what we should realise is that that draw_Mandelbrot requires arguments. We need to pass some arguments through the X-library to draw_Mandelbrot. The X library provides a mechanism for this: the fifth argument of XtAddEventHandler is a pointer of type void *, which is passed on to the function. This is exactly the *extra argument* mechanism that was introduced in Chapter 4.

To use the extra argument mechanism here, we need to define a structure that will hold the arguments and an extra function to unpack the structure and pass the arguments to draw_Mandelbrot. The structure to pass the arguments is:

```
typedef struct {
  double x, y ;
  double width, height ;
} rectangle ;
```

The four elements of this structure each hold one of the arguments. The function that is called by the X-mechanism is shown below. This function has four arguments. The second argument is the pointer to the structure. The type used by X-windows is XtPointer, instead of void *, to guarantee that X programs can also be compiled with pre-ansi C compilers. We ignore the other arguments in this example:

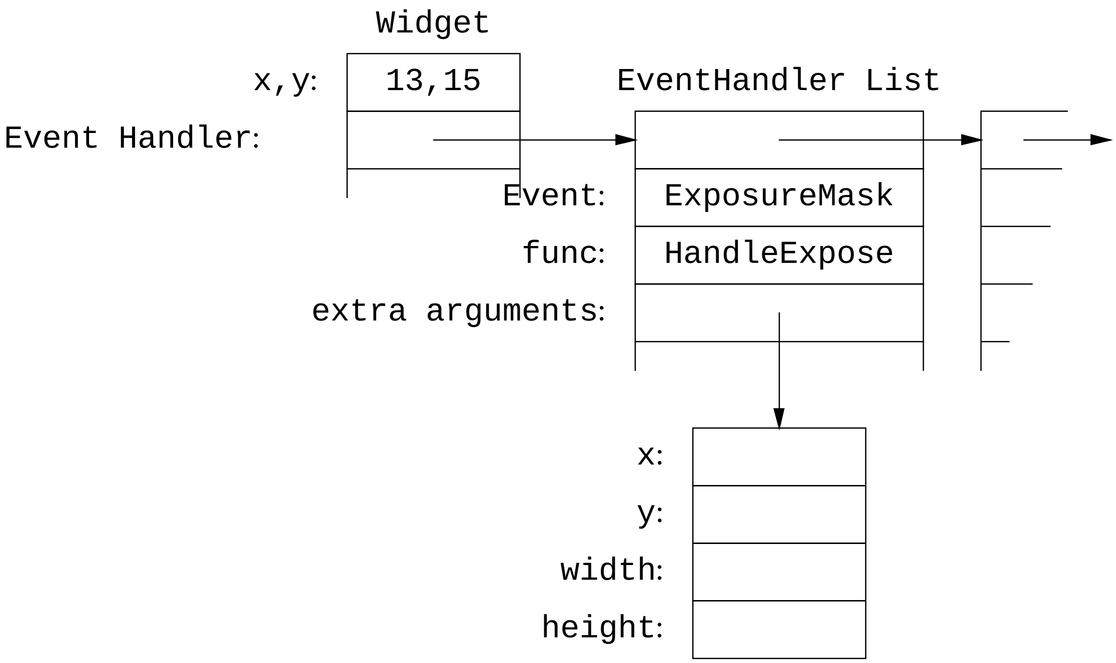
```
void handle_expose( Widget w, XtPointer data,
                    XEvent *e, Boolean *cont ) {
  rectangle *r = data ;
  draw_Mandelbrot( r->x, r->y, r->width, r->height ) ;
}
```

When an expose event comes in, the entire Mandelbrot graphic is drawn. It is not always necessary to draw the whole Mandelbrot graphic, because sometimes only parts of the window need to be redrawn. Inspecting the e argument would allow the program to find out exactly which parts need to be redrawn, but that is beyond the scope of this example.

Now that this event is being handled properly, it is no longer necessary to draw the Mandelbrot graphic in the main program, for as soon as the window has been created, an Expose event is posted automatically by the library. This will cause the Mandelbrot graphic to be drawn. Thus the call to draw_Mandelbrot will be removed from the main program:

```
int main( int argc , char *argv[] ) {
  XtAppContext context ;
  int          theScreen ;
  rectangle    r ;
  Widget widget = XtVaAppInitialize( &context, "XMandel",
                      NULL, 0, &argc, argv, NULL, NULL ) ;
  XtVaSetValues( widget, XtNheight, HEIGHT,
                         XtNwidth,  WIDTH, NULL ) ;
  XtRealizeWidget( widget ) ;
  theDisplay = XtDisplay( widget ) ;
  theWindow  = XtWindow( widget ) ;
  theScreen  = DefaultScreen( theDisplay ) ;
  white      = WhitePixel( theDisplay, theScreen ) ;
  black      = BlackPixel( theDisplay, theScreen ) ;
  printf("Enter originx, y, width and height: " ) ;
  if( scanf( "%lf%lf%lf%lf", &r.x, &r.y, &r.width,
                               &r.height ) != 4 ) {
    printf("Sorry, cannot read these numbers\n" ) ;
  } else {
    XtAddEventHandler( widget, ExposureMask, false,
                       handle_expose, &r ) ;
    XtAppMainLoop( context ) ;
  }
  return 0 ;
}
```

When XtAddEventHandler is called, the function handle_expose is stored in a data structure linked to the widget. The details of the store are well hidden, but one can envisage that the Widget structure maintains a list of functions that handle events, as follows:



The function XtAppMainLoop can find out which functions are to be notified for which events. It does so via the variable context, which maintains a pointer to the top level widget. The function XtAppMainLoop then calls these functions with the appropriate extra arguments when necessary (the precise structure is more complicated than sketched here).

Another event that we might be interested in is when the user presses a button on the mouse. We can use this action to let the user indicate that the application should stop. Reacting to this button requires adding another event handler to the program:

```
XtAddEventHandler( widget, ButtonPressMask, false,
                   handle_button, NULL ) ;
```

The function handle_button must be defined as:

```
void handle_button( Widget w, XtPointer data,
                    XEvent *e, Boolean *cont ) {
  exit( 0 ) ;
}
```

The call to exit gracefully stops the program.

This concludes the presentation of our first case study. It is still a relatively small C program but it performs an interesting computation and renders its results graphically using a windows-library. The program combines a number of good software engineering techniques to create a structure of reusable modules. The program can be extended to enhance its functionality, for example, to use colour instead of black and white rendering, or to zoom in on certain areas of the complex plane.

Exercise 9.2 Modify the Mandelbrot program to draw using different colours. Use the number of steps made to decide which colour to use. The Mandelbrot set itself stays black, the area outside the Mandelbrot set will be coloured. Here are a few functions to use from the X-windows library:

`DefaultColormap( d, s )` returns the colour map of screen `s` on display `d`.

`XAllocColorCells( d, c, 1, NULL, 0, col, n )` When given a display `d`, a colour map `c`, and an array `col` of `n` longs, this function will allocate `n` colours and store them in the array `col`. The function `XAllocColorCells` only allocates colours, it does not fill them with specific colours. This means that each of the longs in the array is now initialised with a value which refers to an as yet undefined colour.

`XStoreColor( d, c, &colour )` This function fills one particular colour. The variable colour is a structure of type `XColor` which has the following members:

- `long pixel` This should be the reference to the colour that you want to set.
- `unsigned short red, green, blue` These are the values for the red, green, and blue components: 0 means no contribution of that colour and, 65535 means maximum; 0,0,0 represents black; 65535, 65535, 65535 represents white.
- `int flags` This field informs which values to set in the colour table, The value `DoRed | DoGreen | DoBlue` sets all colours.

Here is an example of a function that manipulates the colour map:

```
#define NCOLOURS 1
static unsigned long cells[NCOLOURS] ;

void setcolours( Display *d, int screen ) {
  XColor colour ;
  Colormap c = DefaultColormap( d, screen ) ;
  XAllocColorCells( d, c, 1, NULL, 0, cells, NCOLOURS ) ;
  colour.flags = DoRed | DoGreen | DoBlue ;
  colour.red   = 65535 ;
  colour.green = 55255 ;
  colour.blue  = 0     ;
  colour.pixel = cells[0] ;
  XStoreColor( d, c, &colour ) ;
}
```

The function `setcolours` creates a single colour, gold in this case. Whenever `XSetForeground` is called with `cells[0]` as the colour, the pixels will be rendered gold.

9.2 Second case study: device independent graphics

The first case study used the X-windows system as its sole output device. In this second case study we will develop a well engineered *device driver*. The X-windows library offers functions for rendering boxes, circles, and so on, and there are function calls to initialise the X-windows library. All graphics systems have an interface which has such a structure, but the details differ. In this section we will show how to define a general interface that supports the various devices.

We will first develop an interface that has only limited functionality: it should be possible to draw a line from one point to another. As soon as that has been achieved, the functionality will be extended to cater for other primitives, such as boxes, circles or text. The running example will consider two output devices: X-windows and PostScript, but other devices can be added with relative ease. X Windows was introduced in the previous chapter. It is a windowing system that runs on almost any workstation. PostScript is a graphics language that is mainly used to control the output of printers. We will give a brief description of PostScript below.

9.2.1 PostScript

PostScript is a programming language. A PostScript program is specified in ASCII. In order to render the graphics, the program need to be interpreted. PostScript interpreters are found in printers, and in many windowing systems. Note the difference with X: X graphics are drawn by calling a C function, PostScript graphics are drawn by generating a bit of PostScript code. Consider the following PostScript program:

```
%!PS
newpath
  0   0 moveto
  0 100 lineto
100 100 lineto
100   0 lineto
closepath
stroke
showpage
```

The first line `%!PS` declares this as a PostScript program. The second line starts a new line drawing, called a *path* in PostScript. The `moveto` command moves the current position to the point with coordinates (0,0).

PostScript is a *stack based language*. This means that arguments are listed *before* giving the name of the function for which the arguments are intended. PostScript uses the standard Cartesian coordinate system, with the lower left corner of the area that can be used for drawing at (0,0). This is thus different from X-windows, which uses the top left hand corner of the image as point (0,0). Such differences will be hidden by the device independent driver that we are about to develop.

In the PostScript program above, the first `lineto` command defines a line from the current point, (0,0), to the new point (0,100). The next `lineto` defines a line from the current point, (0,100), to the point 100,100. The `closepath` command closes the path (defining a line from the end to the starting point). The `stroke` command actually draws a line through all the points along the path that we have specified. The final `showpage` command causes the output device to start rendering all drawing and text on the current page.

The PostScript unit is a *point*, which is $\frac{1}{72}$ of an inch. The PostScript program above thus draws a square with its south west corner at the origin of the coordinate system. The square has sides slightly larger than 1 inch.

In all our examples we will print PostScript code in an output file. To render such files, it must be viewed using a PostScript viewer (`ghostview` for example), or it must be sent to a PostScript compatible printer. We will not discuss PostScript in any more detail here, we refer the interested reader to the PostScript Reference Manual [4].

9.2.2 Monolithic interface design

To draw a line in X-windows we have to execute the following C function call:

```
XDrawLine( display, window, gc, x1, y1, x2, y2 ) ;
```

Here the display, window and graphics context are identified by `display`, `window` and `gc`. The last four arguments identify the (integer) coordinates of the begin and end point of the line. A PostScript program fragment to draw a line reads:

x_1 y_1 `moveto`
x_2 y_2 `lineto`

Here (x_1, y_1) and (x_2, y_2) are the (real) coordinates of the begin and the end point of the line. The data structures and function that will draw a line in either PostScript or X-windows should take into account the peculiarities of both X-windows and PostScript. Here are the appropriate data structures, where `X11` represents the X-windows library and `PS` the PostScript format:

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <stdio.h>

#define X11HEIGHT 100
#define X11WIDTH  100

typedef enum { X11, PS } Devicetype ;
typedef struct {
  Devicetype tag ;
  union {
    struct {
      Display *d ;
      Window w ;
      GC gc ;
      XtAppContext context ;
      Widget widget ;
    } X11 ;
    struct {
      FILE *out ;
    } PS ;
  } c ;
} Device ;
```

The type Device captures the information about the device. In this case there are only two devices: X11, or PS. Specific information pertaining to each device is needed in the structure. For X-windows the display, window, graphics context, application context and the widget to be used for graphics operation must be stored. For PostScript we store a file descriptor.

The line drawing function itself performs case analysis to decide which output device is appropriate:

```
void draw_line( Device *d,
                int x0, int y0, int x1, int y1 ) {
  switch( d->tag ) {
    case X11:
      XDrawLine( d->c.X11.d, d->c.X11.w, d->c.X11.gc,
                 x0, X11HEIGHT-1-y0, x1, X11HEIGHT-1-y1 ) ;
      break ;
    case PS:
      fprintf( d->c.PS.out, "newpath\n" ) ;
      fprintf( d->c.PS.out, "%d %d moveto\n", x0, y0 ) ;
      fprintf( d->c.PS.out, "%d %d lineto\n", x1, y1 ) ;
      fprintf( d->c.PS.out, "closepath\n" ) ;
      fprintf( d->c.PS.out, "stroke\n" ) ;
      break ;
    default:
      abort() ;
  }
}
```

The function draw_line takes the device parameter, and uses the tag to find out whether to draw a line on the X screen, or whether to output PostScript code. The code also does some transformations: X11 draws upside down (low Y values are at the top of the drawing), while a low Y value in PostScript represents something at the bottom of the drawing. These differences are hidden by the device driver.

The example can be extended with other devices and more functions. For example, the function shown below draws a box. The subtractions made in the arguments to XDrawRectangle show how the device independence hides the differences between the PostScript and X11 views on the coordinate systems:

```
void draw_box( Device *d,
               int x0, int y0, int x1, int y1 ) {
  switch( d->tag ) {
    case X11:
      XDrawRectangle( d->c.X11.d, d->c.X11.w, d->c.X11.gc,
                      x0, X11HEIGHT-1-y0, x1-x0, y1-y0 ) ;
      break ;
    case PS:
      fprintf( d->c.PS.out, "newpath\n" ) ;
      fprintf( d->c.PS.out, "%d %d moveto\n", x0, y0 ) ;
      fprintf( d->c.PS.out, "%d %d lineto\n", x0, y1 ) ;
      fprintf( d->c.PS.out, "%d %d lineto\n", x1, y1 ) ;
      fprintf( d->c.PS.out, "%d %d lineto\n", x1, y0 ) ;
      fprintf( d->c.PS.out, "%d %d lineto\n", x0, y0 ) ;
      fprintf( d->c.PS.out, "closepath\n" ) ;
      fprintf( d->c.PS.out, "stroke\n" ) ;
      break ;
    default:
      abort() ;
  }
}
```

The extension of the device independent driver with a box primitive is not much work. However, each time that another device has to be added, both functions `draw_line` and `draw_box` need to be modified. This is not much work in the case of two functions, but with 15 primitive elements it becomes clear that this solution lacks structure: each function contains information about every device. This means that information about one particular device is scattered over all functions, and implementation of a new device will require the modification of all functions.

9.2.3 Modular interface design

A better way to build a general interface to graphics devices is to insert a new level of abstraction. Looking from an abstract viewpoint, a library to draw pictures on some output device consists of a set of functions for drawing the various primitive elements. For example:

draw box	lower left	(x_0, y_0)	upper right	(x_1, y_1)
draw line	from	(x_0, y_0)	to	(x_1, y_1)
draw circle	centre	(x, y)	radius	r
draw ellipse	centre	(x, y)	radii	r_x and r_y

To manipulate a set of functions such as those listed above we need a data structure that holds them together. In SML we could define the data structure as follows:

```
datatype  a graphics
    = DrawBox of     ( a * int * int * int * int ->  a)
    | DrawLine of    ( a * int * int * int * int ->  a)
    | DrawCircle of  ( a * int * int * int ->  a)
    | DrawEllipse of ( a * int * int * int * int ->  a) ;
```

This data type stores *functions* to draw a box, a line, a circle and an ellipse. It takes something of type `a` (an array of pixels or something else), and it produces a new version of `a`. The designer of a graphics device will have to implement the functions needed to render lines, boxes, and so on, and will create a data structure of the type `graphics` which contains these four functions. The user of the graphics library will simply use one of the available `graphics` structures, and call the functions.

The SML data structure is not complete; an extra `open` function is needed to create an initial value for `a`, and also a `close` function is needed. The full definition of the SML device driver would read:

```
datatype ( a, b, c) graphics
    = Open of        ( b ->  a)
    | DrawBox of     ( a * int * int * int * int ->  a)
    | DrawLine of    ( a * int * int * int * int ->  a)
    | DrawCircle of  ( a * int * int * int ->  a)
    | DrawEllipse of ( a * int * int * int * int ->  a)
    | Close of       ( a ->  c) ;
```

Here `b` is a device dependent data type containing information on how to open the device (for example the size and position of a window, or the filename for a postscript file). Similarly `c` is a device dependent type containing any information that remains after the last object has been drawn.

The `graphics` structure can be translated into C, where we use the state hiding principles of the previous chapter. The function `open` will allocate the state, and return a pointer to it, the other functions will receive this pointer, and modify the state when appropriate. The `close` function in C explicitly deallocates storage, because C has explicit memory management.

```
typedef struct {
  void *(*open)( void *what ) ;
  void  (*draw_line)( void *g, int x0, int y0,
                               int x1, int y1 ) ;
  void  (*draw_box)( void *g,  int x0, int y0,
                               int x1, int y1 ) ;
  /*C Other elements of device driver*/
  void  (*close)( void *g ) ;
} graphics_driver ;
```

Exercise 9.3 Add support for drawing circles and ellipses to the device driver structure.

Exercise 9.4 What is the most important difference between the SML and the C data structure?

For any specific graphics library we need to specify the functions to perform these primitive operations, and a structure containing the pointers to these functions.

So, we can now define the following module for the X11 driver:

```
#include "X11driver.h"
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  Display *d ;
  Window w ;
  GC gc ;
  XtAppContext context ;
  Widget widget ;
} X11Info ;

#define HEIGHT 100
#define WIDTH  100

void *X11_open( void *what ) {
  X11Info *i = malloc( sizeof( X11Info ) ) ;
  X11Open *args = what ;
  long black ;
  XSetWindowAttributes attrib ;
  i->widget = XtVaAppInitialize( &i->context, "XProg", NULL,
                                 0, args->argc, args->argv,
                                 NULL, NULL ) ;
  XtVaSetValues( i->widget, XtNheight, HEIGHT,
                         XtNwidth, WIDTH, NULL ) ;
  XtRealizeWidget( i->widget ) ;
  i->d = XtDisplay( i->widget ) ;
  i->w  = XtWindow( i->widget ) ;
  i->gc = XCreateGC( i->d, i->w, 0, NULL ) ;
  black = BlackPixel( i->d, DefaultScreen( i->d ) ) ;
  XSetForeground( i->d, i->gc, black ) ;
  attrib.backing_store = Always ;
  XChangeWindowAttributes( i->d, i->w, CWBackingStore, &attrib) ;
  return (void *) i ;
}

void X11_draw_line( void *g,
                    int x0, int y0, int x1, int y1 ) {
  X11Info *i = g ;
  XDrawLine( i->d, i->w, i->gc, x0, HEIGHT-1-y0,
                                x1, HEIGHT-1-y1 ) ;
}
```

```
void X11_draw_box( void *g,
                   int x0, int y0, int x1, int y1) {
  X11Info *i = g ;
  XDrawRectangle( i->d, i->w, i->gc, x0, HEIGHT-1-y0,
                                     x1-x0, y1-y0 ) ;
}

void X11_close( void *g ) {
  X11Info *i = g ;
  XFreeGC( i->d, i->gc ) ;
  XtAppMainLoop( i->context ) ;
  free( i ) ;
}

graphics_driver X11Driver = {
  X11_open,
  X11_draw_line,
  X11_draw_box,
  /*C other functions of the driver*/
  X11_close
} ;
```

The interface of the module exports the driver information, that is the structure of the type `graphics_driver`. It also exports the definition of the structure that defines which parameters must be passed to open the device, in this case a pointer to the argument count and the argument vectors:

```
#ifndef X11_DRIVER_H
#define X11_DRIVER_H

extern graphics_driver X11Driver ;

typedef struct {
  int *argc ;
  char **argv ;
} X11Open ;

#endif /* X11_DRIVER_H */
```

Exercise 9.5 Add support for drawing circles and ellipses to the X11 device driver.

Exercise 9.6 Define the appropriate data structure and functions for the Postscript driver.

To use the device independent graphics, we need to pass structures of type `graphics_driver` around. Here is a function that draws a cross, using the draw_line primitives from the driver:

```
void draw_cross( graphics_driver *g, void *openinfo ) {
  void *driver = (*g->open)( openinfo ) ;
  (*g->draw_line)( driver, 0, 0, 100, 100 ) ;
  (*g->draw_line)( driver, 100, 0, 0, 100 ) ;
  (*g->close)( driver ) ;
}

int main( int argc, char *argv[] ) {
  X11Open arguments ;
  arguments.argc = &argc;
  arguments.argv = argv ;
  draw_cross( &PSDriver, "out.ps" ) ;
  draw_cross( &X11Driver, &arguments ) ;
  return 0 ;
}
```

The device independent graphics drivers have been developed using all the techniques that we encountered earlier: extra arguments (Section 4.5), higher order functions (Section 2.5), structured data types (Chapter 4), and handling state (Section 8.4). This has now resulted in a well engineered module of code. Just to list some of the properties of this module:

- Drivers are coded in separate modules. The advantage of this is that maintenance and development of code are decoupled, and that different people can develop drivers independently. The localisation of errors is also easier.

- Instances of different drivers can be used simultaneously. It is possible for a program to use both a PostScript driver and an X11 driver at the same moment in time. A monolithic design does not necessarily have this property: it might use global variables that are shared between the drivers, which will result in run time chaos when the two drivers are used simultaneously.

- If the device allows it, multiple instances of the same driver can be opened simultaneously. For example, output can be generated on three postscript files at the same time. A design that would use global variables in the modules, for example `FILE *out` as a static variable in the module PostScript, would support only one file at a time.

Exercise 9.7 It is possible to write a PostScript fragment that is to be included in another PostScript program (for examples a figure in a book). This so called *encapsulated PostScript* needs a bounding box which specifies how big the picture is. The bounding box is specified in PostScript program in the following way:

%%BoundingBox: x_0 y_0 x_1 y_1

Here (x_0, y_0) is the coordinate of the lower left hand corner and (x_1, y_1) is the coordinate of the upper right hand corner of the figure. Modify the PostScript driver so that the bounding box is maintained (in the PSInfo structure), and printed just before the file is closed. To be properly understood by PostScript interpreters you must print a line

```
%%BoundingBox: (atend)
```

Immediately after the `%!PS` line in the beginning.

Exercise 9.8 The computer system that you have access to may have a native graphics system that is different from X11 or PostScript. Develop a device driver that interfaces to the native graphics system.

Exercise 9.9 Enhance the graphics driver to allow the use of colour.

Exercise 9.10 Rewrite the fractal program of our first case study to use the device independent graphics library. Check the PostScript output of your program.

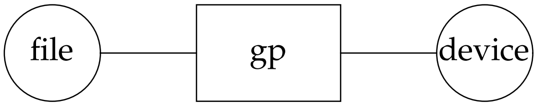
9.3 Third case study: a graphics language

We have now discussed how a device independent library of graphics primitives for boxes, circles and lines can be built. This library can be used directly from a C program that intends to create some graphics. However, the library functions are still quite low level because bookkeeping is required to place and size the lines, circles and boxes appropriately. This can be especially cumbersome when a complicated picture is created out of a large number of primitive graphical objects.

As our third case study we are going to define a graphics programming language for creating line drawings. In addition we will develop a C program, `gp`, that interprets programs written in the graphics language to draw pictures on an output device of our choice. The program will use the device independent graphics library.

We will develop a working skeleton of the language. Not all features are implemented, but the reader is encouraged to add these features. Still, developing this skeleton is not a trivial task: we will have to use some advanced programming techniques which are not covered in this book. We explain these techniques, lexical analysis and parsing, on a need-to-know basis. Books such as the 'red dragon book' [1] on compiler construction, automata, and languages give an in depth coverage of these techniques.

Here is a picture showing the result of using `gp`. The picture has a dual purpose. Firstly it shows the style of line drawings that we will be able to produce. Secondly it symbolises the working of the graphics interpreter `gp` itself:



Examining the picture from left to right we encounter a circle labeled with the text file. This says that we have to create a file containing a graphics program. This file is then read and interpreted by the graphics processor program, as symbolised by the box labeled with the text gp. Finally, the result of the interpretation is shown on an appropriate output device, which is indicated by the circle labeled device.

The graphics language that we will use to create the picture is essentially a simplified and stylised version of the paragraph of text above. The following are the most important elements of the description:

- The English description mentions the graphic *primitives* box and circle.
- The texts "file", "gp" and "device" are used as labels of these three primitives.
- Lines are used to connect the primitives.
- As we are used to reading English text from left to right, we will also assume that pictures in our graphics language are described from left to right.

With these considerations in mind, the following graphics program seems to be a reasonable and succinct description of our picture:

```
.PS
circle "file" ;
line ;
box "gp" ;
line ;
circle "device"
.PE
```

The box and the circle are labeled by writing the text of the label (in double quotes) next to the primitives as *attributes*. The two line primitives take care of the connections. We will call a primitive with its associated attributes an *element*. Then all elements are separated by semi colons and as a finishing touch, the keywords .PS and .PE indicate the start and the end of the graphics program.

The graphics programming language as we have describe here is actually a sub-set of the PIC language designed by Brian Kernighan [6], and our gp program will be a tiny version of the full implementation of the PIC language.

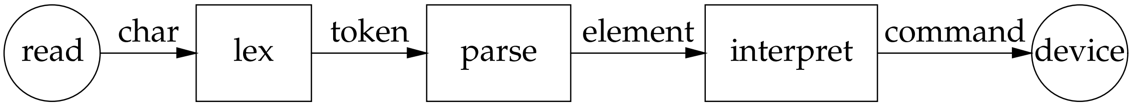
To build an interpreter for a language is a non trivial task because it requires the interpreter (in our case the gp program) to understand the meaning of the words and the sentences of the language. We can identify the following tasks involved in this process of understanding:

lex Recognising the words of the language, such as box, "file", ; and .PS, where a word consists of a sequence of characters. This first task is called the *lexical analysis* or lexing for short.

parse Recognising the sentences of the language, such as circle "file" ;, where the sentences consist of sequences of words. This second task is called *parsing*.

interpret Interpreting the individual sentences in such a way that they form part of a whole: the picture.

Using our graphics language to describe these three tasks and their relationship we arrive at the following structure:



In the following sections we will look at these three tasks and set a number of interesting exercises. When worked out successfully they create a complete `gp` program capable of drawing pictures, of which we have already seen many examples. Several more examples will follow shortly.

9.3.1 Lexical analysis

The graphics language has four different kinds of words, of which we have already seen three categories:

ident (short for identifier). Examples are `circle` and `.PS`.

symbol such as `;`.

text enclosed in double quotes, such as `"file"`.

number It seems a good idea to include also real numbers in our graphics language, as they will be handy for creating primitives of a specific size and movements over a certain distance. An example of a number is `3.1415`.

error We have now essentially decided upon the possible form of the words that we admit to our language. However, a well designed program should always be able to deal with incorrect input. To cater for this we add a further category of words that represent erroneous input. Some examples are: `@#%$` and `3.A`

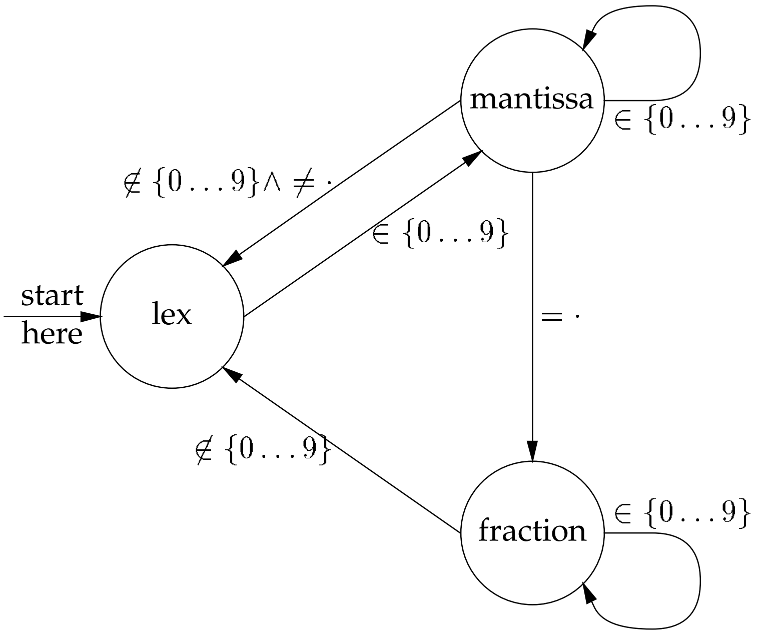
Now we can define a data structure to represent all possible words, which is conventionally called a *token*. Here is the SML version of the data structure:

```
datatype token = Number of real
               | Ident  of string
               | Text   of string
               | Symbol of string
               | Error ;
```

Exercise 9.11 Define a C data structure `token` that is equivalent to the SML `token`, above. Create `malloc` based functions to dynamically allocate `structs` of the appropriate form and contents and create a set of access functions. Package the data structure and its house keeping functions in a module with a clean interface.

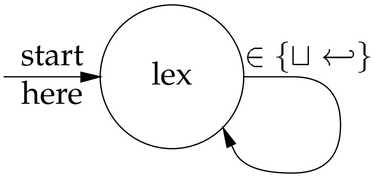
A graphics program, like most other programs, is stored in a file as a sequence of characters. The `token` data structure represents the words of the language, and we have seen that the words are individual sequences of characters. Our next task therefore is to write a function that when given a sequence of characters produces a sequence of tokens: the lexical analysis.

Interestingly, lexical analysis can be described accurately using a picture of the kind that our graphics language is able to create! Here is the process depicting the recognition of real numbers. The circles represent the fact that the recognising process is in a particular *state* and the arrows represent *transitions* from one state to the next.



Recognising a real number is done by interpreting the picture as a road map and using the input characters for directions. If we start in the state labeled lex then we can only move to the state mantissa if a digit in the range $\{0 \ldots 9\}$ is seen in the input. In the state mantissa, further digits can be accepted, each returning into the current state. As indicated by the two remaining arrows, there are two ways out of the state mantissa. When a full stop is seen we move to the state fraction and when some other character is seen, we move back to the initial state lex. In the state fraction we will accept further digits, but when encountering something else than a digit we return to the initial state. The result of a journey using the road map is the collection of characters that have been accepted. This collection represents the current word or token.

A 'road map' such as the one above is called a *state transition diagram*. Similar diagrams can be made for recognising the other tokens of the language; here is another state transition diagram that deals with layout characters, such as spaces (shown as ⊔) and new lines (shown as ↩).



Exercise 9.12 Draw the state transition diagrams for the recognition of `ident`, `text`, `symbol` and `error`.

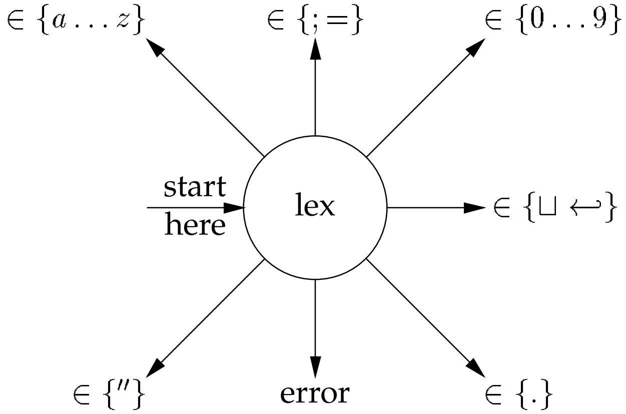
`ident` An identifier begins with a letter or a full stop (`.`), which is then followed by zero or more letters or digits.

`text` A text is an arbitrary sequence of characters enclosed in double quotes (`"`);.

`symbol` A symbol can be either a semi colon (`;`) or an equal sign (`=`).

`error` When anything else is encountered, an error token is returned.

The state diagrams are combined into one by making the lex circle common to all diagrams. If the individual diagrams have been designed properly, there should be a number of arcs out of the lex circle, but no two should carry a label with the same character attached. Furthermore, all possible characters should be dealt with. The error token is used to achieve this:



Exercise 9.13 Write a C function `lex` that implements the combined state transition diagram for the lexical analyser of our graphics language. The function `lex` should take its input from `stdin`.

Gather `lex` and its auxiliary functions in a module, but make sure that only `lex` and the `token` type are exported through the interface.

Exercise 9.14 With the lexical analyser in place it should be possible to write a small test program that reads characters from `stdin` to produce a sequence of tokens. Print the tokens one per line.

9.3.2 Parsing

The lexical analysis delivers a stream of tokens, one for each call to the function `lex`. So now is the time to gather selected tokens into larger units that correspond to the sentences of the language. To make life easier for the interpreter we will define a number of data types that describe the relevant sentences and sub sentences of the graphical language.

Here is the SML data type definition of the graphical primitives:

```
datatype primitive = Box | Circle | Line ;
```

To allow for some flexibility in the graphics language we allow for expressions that consist of either an identifier or a number:

```
datatype expression = Ident  of string
                    | Number of real ;
```

With a primitive we would like to associate not only its text label, but also whether it should have a particular height, width or radius. This gives rise to the following SML data type definition for an attribute of a graphics primitive:

```
datatype attribute = Height of expression
                   | Width  of expression
                   | Radius of expression
                   | Text   of string ;
```

We are now ready to define the data type for the elements (that is, the sentences of the graphics language):

```
datatype element = Up
                 | Down
                 | Right
                 | Left
                 | Assign of string * expression
                 | Prim   of primitive * attribute list ;
```

The `element` data type allows for six different kinds of sentences. The first four are intended to move to a particular position before beginning to draw the next object. The `Assign` constructor associates an identifier (represented here as a string of characters) with an expression. This will enable us to give a name to an expression and refer to that name wherever the value of the expression is required. The last element is the most interesting for it associates a graphics primitive with its list of attributes.

Exercise 9.15 Create a module that defines the data structures in C to represent the `primitive`, `expression`, `attribute` and `element`. These should be heap allocated structures.

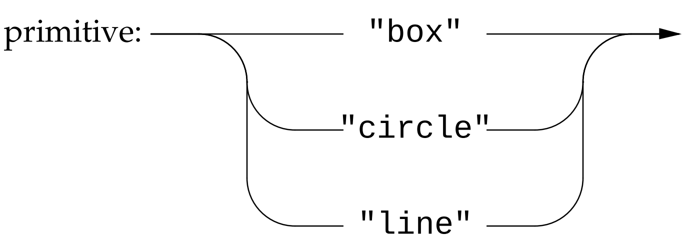
The definition of an `element` would allow us to write (in SML):

```
val box_primitive = Prim(Box,[Text "a box",
                              Height (Number 3.0)]);
```

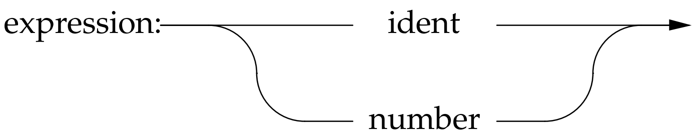
The corresponding list of tokens returned by our lexical analyser would be (also in SML):

```
val box_tokens = [Ident  ".PS",
                  Ident  "box",
                  Text   "\"a box\"",
                  Ident  "height",
                  Number 3.0,
                  Ident  ".PE" ];
```

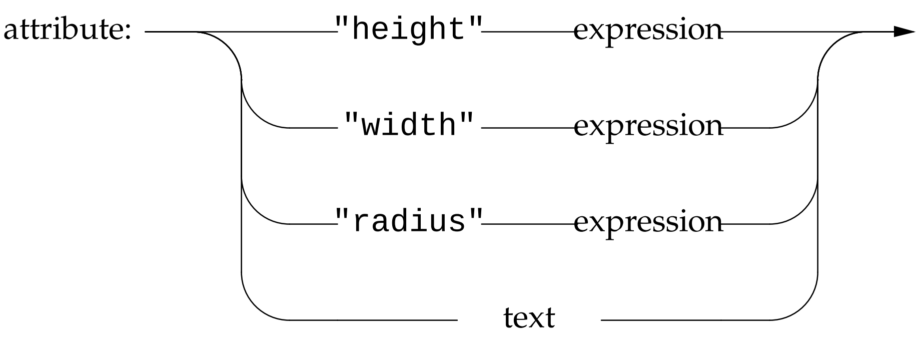
The task of the parser is to gather the tokens in such a way that they fall in the right place in the right (that is, `primitive`, `expression`, `attribute` or `element`) data structure. To achieve this we will use the same technique of state transition based recognition as with the lexical analyser. Firstly, we draw a series of road maps. Here is the state diagram for a primitive. It shows that a primitive can only be one of the three identifiers `"box"`, `"circle"`, or `"line"`:



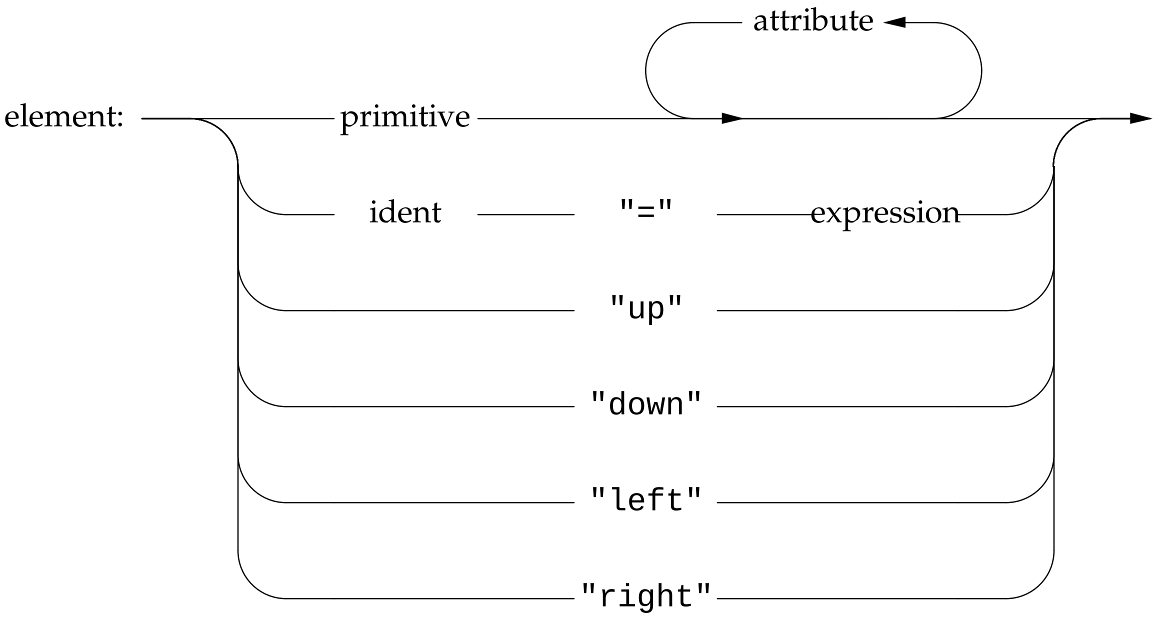
An expression can be an identifier or a number, where any particular identifier or number is acceptable. This is indicated by just mentioning states ident and number. These refer to the tokens of the same name as delivered by the lexical analyser.



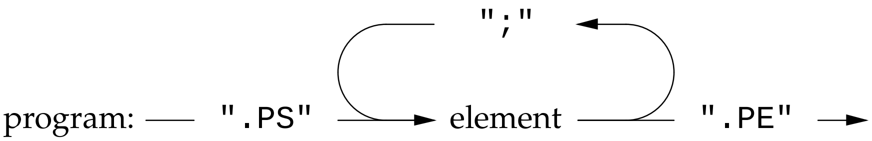
An attribute is more interesting. It consists of a series of alternatives, of which the first three have to consist of two items:



An element is also one of a series of alternatives. A primitive may be followed by zero or more attributes; an assignment must consist of three consecutive items and the moves consist of just the words `up`, `down`, `left` and `right`.



A `gp` program consists of a list of elements, separated by semi colons and surrounded by the words `.PS` and `.PE`:



Exercise 9.16 Write one parse function for each of `primitive`, `attribute`, `expression`, `element`, and `program`. Each parse function should call upon `lex` to retrieve token(s) from the `stdin` stream. Based upon the token received, your parse functions should decide what the next input sentence is, and return a pointer to the `struct` of the appropriate type and contents.

Exercise 9.17 Write a main function that uses `lex` and the parse functions to read a `gp` program from `stdin` and to create a representation of the program in the heap as a list of elements. Write a printing function for each of the data structures involved and use this to print the heap representation of a `gp` program.

9.3.3 Interpretation

We have now at our disposal a library of device independent graphics primitives and the lexing and parsing tools to create a structured, internal representation of a gp program in the heap. To combine these elements we need to write an interpreter of the internal representation of the program, that calls the appropriate library routines. Before we can embark on this two further explanations of how gp works.

Firstly, the program has a notion of a current point and a current direction. By default the current direction is right, but it can be modified by one of the commands up, down, left or right.

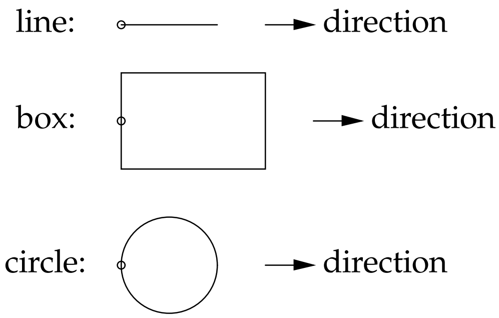
The current point will be aligned with a particular corner or point of the next graphical object to be drawn. Let us assume for now that the current direction is right. The current point is then aligned with:

line The begin point of the line.

box The middle of the left vertical.

circle The left intersect of the circle and a horizontal line through its centre.

The small circles in the picture below give a graphical indication of the alignment points.



The alignment point is thus as far away from the point to which we are moving. This also applies to moves in one of the other three directions. For example, should we be moving upwards, the alignment point is below the object, instead of its left.

Secondly, the gp program maintains a set of variables that control the default dimensions of the primitive objects, as well as the default moving distances. These variables can be set using assignment statements. The variables and their initial settings (in inches) are:

```
.PS
boxht     = 0.5  ;
boxwid    = 0.75 ;
circlerad = 0.25 ;
lineht    = 0.5  ;
linewid   = 0.5  ;
```

```
moveht    = 0.5  ;
movewid   = 0.5  ;
textht    = 0.0  ;
textwid   = 0.0
.PE
```

The default width of a box is 0.75 inches and its default height is 0.5 inches. The default can be overridden either by changing the value of the appropriate variable using an assignment, or by using explicit attributes to state the sizes, so that for example: `box width 0.5` draws a square, and so does this `gp` program:

```
.PS
boxwid = boxht ;
box
.PE
```

Exercise 9.18 Implement the interpreter for the internal representation of a `gp` program.

The `gp` language can be extended with a large variety of useful constructs. The exercises at the end of this chapter provide a number of suggestions.

9.4 Summary

Programming graphics systems is rewarding, but it also a difficult topic that is worthy of study on its own. In this chapter we have merely hinted at some of the possibilities of computer graphics. Three case studies have been made to show how the principles of programming as we have developed them in this book are used in practice. The most important practical points are:

- Use separate modules whenever possible. Modularisation structures the program and allows independent development, testing and maintenance.

- Hide platform and machine dependencies (for example graphics libraries) in separate modules.

- Use higher order functions when appropriate. Writing device drivers using higher order functions simplifies the structure of the program.

- To create your design use a language with polymorphic types. Then translate the design into C, using `void *` if necessary.

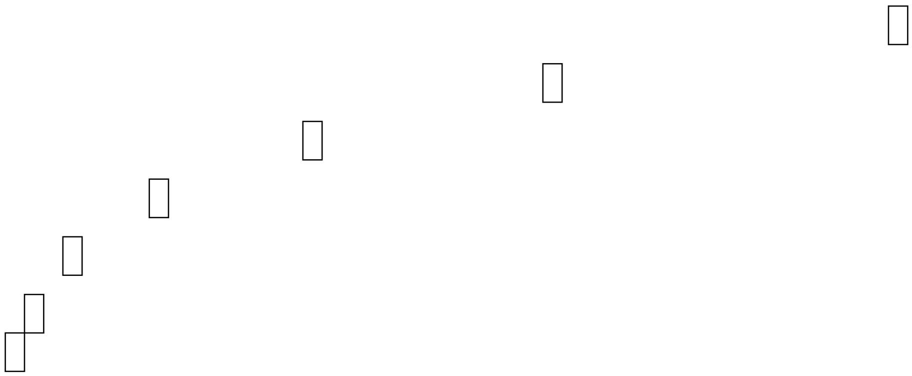
9.5 Further exercises

Exercise 9.19 Extend the expressions of the `gp` language with the usual arithmetic operators, such as `+`, `-`, `*` and `/` and with parentheses `(` and `)`.

Exercise 9.20 Extend the language so that the movements `up`, `down`, `left` and `right` will take attributes, in the same way as a primitive takes attributes.

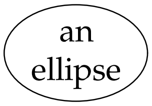
Exercise 9.21 Add to the elements a construct that makes it possible to repeatedly draw a certain object or set of objects:

```
.PS
for i = 1 to 6 do {
  box width 0.1 height 0.2 ;
  right (0.05*i*i) ;
  up 0.1
}
box width 0.1 height 0.2 ;
.PE
```



Exercise 9.22 Add a new primitive: ellipse. Also extend the set of default sizes:

```
.PS
ellipseht  = 0.5  ;
ellipsewid = 0.75 ;
ellipse "an" "ellipse"
.PE
```



Exercise 9.23 Introduce an attribute that allows objects to be drawn in colour.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques, and tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] J. L. Hennessy and D. A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[3] C. A. R. Hoare. Algorithm 64 quicksort. *CACM*, 4(7):321, Jul 1961.

[4] Adobe Systems Inc. *PostScript language reference manual*. Addison Wesley, Reading, Massachusetts, 1985.

[5] R. Jain. *The art of Computer Systems Performance Analysis*. John Wiley, Newyork, 1991.

[6] B. W. Kernighan. PIC — a language for typesetting graphics. *Software—practice and experience*, 12(1):1–21, Jan 1982.

[7] B. W. Kernighan and D. W. Ritchie. *The C programming language - ANSI C*. Prentice Hall, Englewood Cliffs, New Jersey, second edition edition, 1988.

[8] D. E. Knuth. *The art of computer programming, volume 1: Fundamental algorithms*. Addison Wesley, Reading, Massachusetts, second edition, 1973.

[9] L. C. Paulson. *ML for the working programmer*. Cambridge Univ. Press, New York, 1991.

[10] W. H. Press, B. P. Flannery, S. A. Tekolsky, and W. T. Vetterling. *Numerical recipes in C – The art of scientific computing*. Cambridge Univ. Press, Cambridge, England, 1993.

[11] B. Schneier. *Applied cryptography*. John Wiley & Sons, Chichester, England, second edition edition, 1996.

[12] R. Sedgewick. *Algorithms*. Addison Wesley, Reading, Massachusetts, 1983.

[13] E. H. Spafford. The internet worm program: an analysis. *ACM Computer communication review*, 19(1):17–??, Jan 1989.

[14] A. S. Tanenbaum. *Structured computer organisation*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1984.

[15] J. D. Ullman. *Elements of ML programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[16] A. Wikström. *Functional programming using Standard ML*. Prentice Hall, London, England, 1987.

[17] X-Consortium. *X Window Manuals*. O'Reilly & Associates, Inc., New York, 1990.

Appendix A

Answers to exercises

Below are the answers to a selection of the exercises in this book. For almost any exercise, there is more than one correct answer, as there are many algorithms and datastructures that implement a program. The answers that are presented here are the ones that we consider the most appropriate.

Answers to the exercises of Chapter 2

Answer to 2.1: The function `euclid` is called 5 times. A complete trace is:

```
euclid( 558, 198 ) is
euclid( 198, 162 ) is
euclid( 162,  36 ) is
euclid(  36,  18 ) is
euclid(  18,   0 ) is
18
```

Answer to 2.2: The general SML function schema for a cascade of k conditionals is:

```
(*SML general function schema*)
  (* $f$ : $t_1$ -> ... $t_n$ -> $t_r$ *)
  fun $f$ $x_1$ ... $x_n$
      = if $p_1$
           then $g_1$
           else if $p_2$
                then $g_2$
                     ...
                     else if $p_k$
                          then $g_k$
                          else $h$ ;
```

The arguments of f are $x_1 \ldots x_n$, and their types are $t_1 \ldots t_n$ respectively. The expressions $p_1 \ldots p_k$ are predicates over the arguments of f, $g_1 \ldots g_k$ and h are expressions over the arguments of f.

The corresponding general C function schema is:

```
/*C general function schema*/
 $t_r$ $f$( $t_1$ $x_1$, ... $t_n$ $x_n$ ) {
   if( $p_1$ ) {
     return $g_1$ ;
   } else if( $p_2$ ) {
     return $g_2$ ;
     ...
   } else if( $p_k$ ) {
     return $g_k$ ;
   } else {
     return $h$ ;
   }
 }
```

Answer to 2.3: ♣ Here is the table of correspondence for `eval`, using the general function schema of the previous exercise.

schema:	Functional	C
f:	eval	eval
t_1:	int	int
t_2:	char	char
t_3:	int	int
t_4:	char	char
t_5:	int	int
t_r:	int	int
x_1:	x	x
x_2:	o1	o1
x_3:	y	y
x_4:	o2	o2
x_5:	z	z
p_1:	o1 = "+" andalso o2 = "+"	o1 == + && o2 == +
p_2:	o1 = "+" andalso o2 = "*"	o1 == + && o2 == *
p_3:	o1 = "*" andalso o2 = "+"	o1 == * && o2 == +
p_4:	o1 = "*" andalso o2 = "*"	o1 == * && o2 == *
g_1:	(x + y) + z	(x + y) + z
g_2:	x + (y * z)	x + (y * z)
g_3:	(x * y) + z	(x * y) + z
g_4:	(x * y) * z : int	(x * y) * z
h:	raise Match	/*raise Match*/

Answer to 2.5: The program will print the following line:

```
0  = 48,   q  = 113
```

The primes, spaces, equals, and `\n` in the format string are printed literally. The `%c` and `%d` formats are replaced by character and integer representations of the arguments. The first `%c` requires a character argument, `c0`; this is the character `0` . The `%d` format requires an integer, `i0`, which contains the integer representation of `0` , which is 48. The second `%c` requires a character argument again, `cq`; this is the character which is represented by the integer 113, which is `q` . The last `%d` requires an integer, `iq`, which contains the integer 113.

Answer to 2.6: The second definition of r^p requires 8 multiplications: to calculate r^{128}, r^{64} has to be calculated and squared. Therefore it is one multiplication plus the number of multiplications needed to calculate r^{64}. To calculate r^{64}, one multiplication is needed, plus the number of multiplications to calculate r^{32}. Finally, one multiplication is needed to calculate r^1.

In general, the number of multiplications to calculate r^p is bound by $1+2\log_2 p$, as opposed to $p-1$ when one just applies a repeated multiplication.

Answer to 2.7: Proof of the hypothesis $r^p = \prod_{i=1}^{p} r$ by induction over p, where r^p is defined according to (2.4):

Case 1:

$$\begin{array}{lcll}
r^1 & = & r & \{2.4\} \\
 & = & \prod_{i=1}^{1} r & \{\prod\}
\end{array}$$

Case $p+1$, with $p+1$ odd:

$$\begin{array}{lcll}
r^{p+1} & = & r \times r^p & \{2.4\} \\
 & = & r \times \prod_{i=1}^{p} r & \{hypothesis\} \\
 & = & \prod_{i=1}^{p+1} r & \{\prod\}
\end{array}$$

Case $p+1$, with $p+1$ even:

$$\begin{array}{lcll}
r^{p+1} & = & \mathrm{sqr}(r^{(p+1)\,\mathrm{div}\,2}) & \{2.4\} \\
 & = & r^{(p+1)\,\mathrm{div}\,2} \times r^{(p+1)\,\mathrm{div}\,2} & \{\mathrm{sqr}\} \\
 & = & \prod_{i=1}^{(p+1)\,\mathrm{div}\,2} r \times \prod_{i=1}^{(p+1)\,\mathrm{div}\,2} r & \{hypothesis\} \\
 & = & \prod_{i=1}^{p+1} r & \{\prod\}\square
\end{array}$$

Answer to 2.8: ♣ The function `square` is called 4 times, and `power` is called 8 times. A complete trace is:

```
power(1.037155,19) is 1.037155 * power(1.037155,18);
power(1.037155,18) is square( power(1.037155,9) );
power(1.037155,9)  is 1.037155 * power(1.037155,8);
power(1.037155,8)  is square( power(1.037155,4) );
power(1.037155,4)  is square( power(1.037155,2) );
power(1.037155,2)  is square( power(1.037155,1) );
power(1.037155,1)  is 1.037155 * power(1.037155,0);
power(1.037155,0)  is 1.0.
```

Answer to 2.9: ♣ Here is the table of correspondence for the transformation of the SML version of `sum` into the C version.

schema:	Functional	C
f:	`sum`	`sum`
t_1:	`int`	`int`
t_2:	`int`	`int`
t_3:	`int -> real`	`double (*f)( int )`
t_r:	`real`	`double`
x_1:	`i`	`i`
x_2:	`n`	`n`
x_3:	`f`	`f`
p:	`i > n`	`i > n`
g:	`0.0`	`0.0`
h:	`f i + sum (i+1) n f`	`f( i ) + sum( i+1, n, f )`

Answer to 2.10: ♣ Here is an SML function `square` which computes the square of a number by repeatedly summing a progression of odd numbers.

```
(* square : int -> real *)
fun square n
    = let
         fun int2odd i = real (2*i-1)
      in
         sum 1 n int2odd
      end ;
```

Here is the corresponding C version of `square` with its auxiliary function `int2odd`.

```
double int2odd( int i ) {
  return 2*i-1 ;
}

double square( int n ) {
  return sum( 1, n, int2odd ) ;
}
```

Answer to 2.11: ♣ Here is the SML function `nearly_pi`:

```
(* nearly_pi : int -> real *)
fun nearly_pi n
    = let
         fun positive i = 1.0/real (4*i-3)
         fun negative i = 1.0/real (4*i-1)
      in
```

```
          4.0 * ( sum 1 n positive -
                  sum 1 n negative)
        end ;
```

The implementation of this function in C yields two auxiliary functions and the `nearly_pi` function itself.

```
double positive( int i ) {
  return 1.0/(4*i-3) ;
}

double negative( int i ) {
  return 1.0/(4*i-1) ;
}

double nearly_pi( int n ) {
  return 4.0 * ( sum( 1, n, positive) -
                 sum( 1, n, negative) ) ;
}
```

Answer to 2.12: ♣ The C version of `product` and `factorial` are given below. The definition of `int2real` is the same as that defined for `terminal`.

```
double product( int i, int n, double (*f) ( int ) ) {
  if( i > n ) {
    return 1.0 ;
  } else {
    return f( i ) * product( i+1, n, f ) ;
  }
}

double factorial( int n ) {
  return product( 1, n, int2real ) ;
}
```

Answer to 2.13: ♣ The function `nearly_e` calculates an approximation to the number e:

```
(* nearly_e : int -> real *)
fun nearly_e n
    = let
          fun f i = 1.0/factorial i
      in
          1.0 + sum 1 n f
      end ;
```

The translation of this function into C yields an auxiliary function to compute the reciprocal of the factorial and the nearly_e function itself:

```
double recip_factorial( int i ) {
  return 1.0/factorial( i ) ;
}

double nearly_e( int n ) {
  return 1.0 + sum( 1, n, recip_factorial ) ;
}
```

Answer to 2.14: ♣ Here is product redefined in terms of repeat:

```
(* product : int -> int -> (int -> real) -> real *)
fun product i n f
    = let
         fun multiply x y = x * y
      in
         repeat 1.0 multiply i n f
     end ;
```

Answer to 2.15: Here is a monomorphic C version of repeat, which is suitable for use in sum and product.

```
double repeat( double base,
               double (*combine) (double, double),
               int i, int n, double (*f) (int) ) {
  if( i > n ) {
    return base ;
  } else {
    return combine( f( i ),
                    repeat( base, combine, i+1, n, f) ) ;
  }
}
```

Here is the redefined sum in terms of repeat. It requires an auxiliary function double_add:

```
double double_add( double x, double y ) {
  return x+y ;
}

double sum( int i, int n, double (*f) (int) ) {
  return repeat( 0.0, double_add, i, n, f ) ;
}
```

Answer to 2.16: Here is the SML function nearly_phi to calculate an approximation of the golden ratio:

```
  (* nearly_phi : int -> real *)
  fun nearly_phi n
      = let
          fun constant_one i = 1.0
          fun divide x y = x / (1.0 + y)
        in
          1.0 + repeat 1.0 divide 1 n constant_one
       end ;
```

The C version of nearly_phi requires two auxiliary functions:

```
  double constant_one( int i ) {
    return 1.0 ;
  }

  double divide( double x, double y ) {
    return x / (1.0 + y) ;
  }

  double nearly_phi( int n ) {
    return 1.0 + repeat( 1.0, divide, 1, n, constant_one ) ;
  }
```

Answer to 2.17: The general SML function schema for a cascade of k conditionals with j local definitions is:

```
(*SML general function schema with locals*)
  (* $f$ : $t_1$ -> ... $t_n$ -> $t_r$ *)
  fun $f$ $x_1$ ... $x_n$
      = let
          val $y_1$ = $z_1$ (* $y_1$ : $t_{y_1}$ *)
          ...
          val $y_j$ = $z_j$ (* $y_j$ : $t_{y_j}$ *)
       in
          if $p_1$
             then $g_1$
             else if $p_2$
                     then $g_2$
                         ...
                         else if $p_k$
                                 then $g_k$
                                 else $h$
       end ;
```

The $x_1 \ldots x_n$ are the arguments of f, and their types are $t_1 \ldots t_n$ respectively. The type of the function result is t_r. The local variables of f are $y_1 \ldots y_j$; their values are the expressions $z_1 \ldots z_j$ and their types are $t_{y_1} \ldots\ t_{y_j}$ respectively. The expressions $p_1 \ldots p_k$ are predicates over the arguments and the local variables. The $g_1 \ldots g_k$ and h are expressions over the local variables and arguments. The corresponding general C function schema is:

```
/*C general function schema with locals*/
  $t_r$ $f$( $t_1$ $x_1$, ... $t_n$ $x_n$ ) {
    const $t_{y_1}$ $y_1$ = $z_1$ ;
    ...
    const $t_{y_j}$ $y_j$ = $z_j$ ;
    if ( $p_1$ ) {
      return $g_1$ ;
    } else if ( $p_2$ ) {
      return $g_2$ ;
      ...
    } else if ( $p_k$ ) {
      return $g_k$ ;
    } else {
      return $h$ ;
    }
  }
```

Answer to 2.20: ♣ The relation between a temperature in Centigrade and Fahrenheit is:

$$\begin{aligned} c &: I\!R \\ \text{fahrenheit} &: I\!R \to I\!R \\ \text{fahrenheit}(c) &= \frac{c \times 9}{5} + 32 \end{aligned}$$

The SML function reads:

```
(* fahrenheit : real -> real *)
fun fahrenheit c = c * 9.0 / 5.0 + 32.0 ;
```

The corresponding C program is:

```
#include <stdio.h>

double fahrenheit( double c ) {
  return c * 9.0 / 5.0 + 32.0 ;
}

int main( void ) {
  printf( "Fahrenheit\n") ;
  printf( "%f\n", fahrenheit( 0 ) ) ;
  printf( "%f\n", fahrenheit( 28 ) ) ;
  printf( "%f\n", fahrenheit( 37 ) ) ;
  printf( "%f\n", fahrenheit( 100 ) ) ;
  return 0 ;
}
```

This will print the following output:

```
32.000000
82.400000
98.600000
212.000000
```

Answer to 2.21: ♣

(a) The specification of the pop_count function is:

$$\texttt{pop\_count}(b_n b_{n-1} \ldots b_1 b_0) \;=\; \sum_{i=0}^{n} b_i$$

(b) Here is an SML function that implements the population count instruction:

```
(* pop_count : int -> int *)
fun pop_count n
    = if n = 0
         then 0 : int
         else n mod 2 + pop_count (n div 2) ;
```

(c) Here is the C version of pop_count:

```
int pop_count( int n ) {
  if( n == 0 ) {
    return 0 ;
  } else {
    return n % 2 + pop_count (n / 2) ;
  }
}
```

(d) The table of correspondence for the function schema is:

Schema	Functional	C
f	pop_count	pop_count
t_1	int	int
t_r	int	int
x_1	n	n
p	n = 0	n == 0
f	0	0
g	n mod 2 + pop_count (n div 2)	n % 2 + pop_count (n / 2)

(e) Here is a main program that calls pop_count:

```
#include <stdio.h>

/*C population count*/

int main( void ) {
  printf( "population count\n") ;
  printf( "of     0 is %d\n", pop_count( 0 ) ) ;
  printf( "of     9 is %d\n", pop_count( 9 ) ) ;
  printf( "of 65535 is %d\n", pop_count( 65535 ) ) ;
  return 0 ;
}
```

The solution to the population count problem could be formulated better using the bit-operators of C. The >> operator shifts an integer to the right, and the & operator performs an and-operation on the integer. Thus pop_count could be written as:

```
int pop_count( int n ) {
  if( n == 0 ) {
    return 0 ;
  } else {
    return (n & 1) + pop_count( n >> 1 ) ;
  }
}
```

Note that because the & operator has a lower priority than the + operator, the expression n&1 needs to be in parentheses. Without the parentheses, the return value would be n & (1 + pop_count(n>>1)), which is different.

Answer to 2.22: ♣

(a) The specification of the checksum function is:

$$\text{checksum}(n_k n_{k-1} \ldots n_1 n_0) \quad = \quad \sum_{i=0}^{k} n_i$$

(b) Here is an SML function that implements the checksum function:

```
(* checksum : int -> int *)
fun checksum n
    = if n = 0
           then 0 : int
           else n mod 16 + checksum (n div 16) ;
```

Here are some test cases for checksum:

```
checksum 0 ;
checksum 17 ;
checksum 18 ;
checksum 65535 ;
```

(c) Here is the C version of `checksum`:

```
int checksum( int n ) {
  if( n == 0 ) {
    return 0 ;
  } else {
    return n % 16 + checksum(n / 16) ;
  }
}
```

(d) The table of correspondence for the function schema is:

Schema	Functional	C
f	`checksum`	`checksum`
t_1	`int`	`int`
t_r	`int`	`int`
x_1	`n`	`n`
p	`n = 0`	`n == 0`
f	`0`	`0`

g	`n mod 16 +`
	`checksum(n div 16)`
		`n % 16 +`
		`checksum(n / 16)`

(e) Here is a main program that calls `checksum`:

```
int main( void ) {
  printf( "nibble checksum\n") ;
  printf( "of     0 is %d\n", checksum( 0 ) ) ;
  printf( "of    17 is %d\n", checksum( 17 ) ) ;
  printf( "of    18 is %d\n", checksum( 18 ) ) ;
  printf( "of 65535 is %d\n", checksum( 65535 ) ) ;
  return 0 ;
}
```

Answer to 2.23: ♣ Here is an SML function that computes the n-th Fibonacci number:

```
(* fib : int -> int *)
fun fib n
    = if n = 0
         then 0 : int
         else if n = 1
                 then 1
                 else fib (n-1) + fib (n-2) ;
```

Here is the C version of `fib` embedded in a main program:

```
#include <stdio.h>

int fib ( int n ) {
  if( n == 0 ) {
    return 0 ;
  } else if( n == 1 ) {
    return 1 ;
  } else {
    return fib( n-1 ) + fib( n-2 ) ;
  }
}

int main( void ) {
  printf( "Fibonacci\n") ;
  printf( "0  %d\n", fib( 0 ) ) ;
  printf( "1  %d\n", fib( 1 ) ) ;
  printf( "7  %d\n", fib( 7 ) ) ;
  return 0 ;
}
```

Answer to 2.24: ♣

(a) The differences between the Fibonacci series and the nFib series are:

- The Fibonacci series starts with 0, the nFib series with 1;
- The nFib series adds an extra 1 for each application of the inductive case.

(b) Here is an SML function that computes the n-th nFib number:

```
(* nfib : int -> int *)
fun nfib n = if n = 0 orelse n = 1
                then 1 : int
                else 1 + nfib(n-1) + nfib(n-2) ;
```

Here are a few test cases for `nfib`:

```
nfib 0 ;
nfib 1 ;
nfib 7 ;
```

(c) Here is the C version of `nfib`:

```
int nfib( int n ) {
  if( n == 0 || n == 1 ) {
    return 1 ;
  } else {
    return 1 + nfib( n-1 ) + nfib( n-2 ) ;
  }
}
```

(d) Here is a main program for testing the C version of `nfib`:

```
int main( void ) {
  printf( "nFib\n") ;
```

```
    printf( "of 0 is %d\n", nfib( 0 ) ) ;
    printf( "of 1 is %d\n", nfib( 1 ) ) ;
    printf( "of 7 is %d\n", nfib( 7 ) ) ;
    return 0 ;
  }
```

Answer to 2.25: ♣ Here is an SML function that computes powers of powers:

```
(* power_of_power : int -> int -> int *)
fun power_of_power m n
    = if n = 0
         then 1
         else power m (power_of_power m (n-1)) ;
```

It uses a straightforward integer `power` function:

```
(* power : int -> int -> int *)
fun power r p = if p = 0
                   then 1 : int
                   else r * power r (p-1) ;
```

Here is the C version of `power_of_power` embedded in a main program:

```
#include <stdio.h>

int power( int r, int p ) {
  if( p == 0 ) {
    return 1 ;
  } else {
    return r * power( r, p-1 ) ;
  }
}

int power_of_power( int m, int n ) {
  if( n == 0 ) {
    return 1 ;
  } else {
    return power( m, power_of_power( m, n-1 ) ) ;
  }
}

int main( void ) {
  printf( "power of power\n") ;
  printf( "0 17: %d\n", power_of_power( 0, 17 ) ) ;
  printf( "1 17: %d\n", power_of_power( 1, 17 ) ) ;
  printf( "2  0: %d\n", power_of_power( 2,  0 ) ) ;
  printf( "2  1: %d\n", power_of_power( 2,  1 ) ) ;
  printf( "2  2: %d\n", power_of_power( 2,  2 ) ) ;
  printf( "2  3: %d\n", power_of_power( 2,  3 ) ) ;
  printf( "2  4: %d\n", power_of_power( 2,  4 ) ) ;
  return 0 ;
}
```

Answer to 2.26:

(a) Given the function $f(x)$, its derivative $f'(x)$, and an initial approximation x_0, here is how the Newton-Raphson method calculates an approximation to the root:

$$\begin{array}{rcl} \text{newton\_raphson}, f, f' & : & I\!R \rightarrow I\!R \\ \text{newton\_raphson}(x) & = & \left\{ \begin{array}{rl} x, & \text{if } |f(x)| < \epsilon \\ \text{newton\_raphson}(x - \dfrac{f(x)}{f'(x)}), & \text{otherwise} \end{array} \right. \end{array}$$

If $f(x)$ is further from 0 than the (small) distance ϵ allows, a subsequent approximation is made. By choosing a small enough value for ϵ, the root will be determined with a high precision, but for a value of ϵ which is too small the algorithm might not find a root.

(b) For convenience, the SML version of the Newton-Raphson method is parametrised over ϵ, f, and f' so that the distance ϵ and the functions f and f' are passed to the function calculating the root. This gives the following function:

```
(* newton_raphson : (real->real) -> (real->real) ->
                     real -> real *)
fun newton_raphson f f  eps x
    = let
         val fx = f(x) (* fx : real *)
      in
         if absolute(fx) < eps
            then x
            else newton_raphson f f  eps (x-fx/f (x))
      end ;
```

(c) Here is the definition of `parabola` and its derivative `parabola` :

```
(* parabola : real -> real *)
fun parabola  x = x * x - 2.0 ;

(* parabola  : real -> real *)
fun parabola  x = 2.0 * x ;
```

The following table gives the results of using the Newton-Raphson method to find the root of the `parabola` function for a number of values for the initial estimate and the accuracy.

```
newton_raphson parabola parabola  0.001 1.5 = 1.41421;
newton_raphson parabola parabola  0.1 200.0 = 1.41624;
```

As expected the second answer is not as precise as the first ($\sqrt{2} = 1.4142356\ldots$).

(d) The C implementation of the Newton Raphson program is shown below.

```
#include <stdio.h>

double absolute( double x ) {
  if( x >= 0 ) {
    return x ;
  } else {
    return -x ;
  }
}

double newton_raphson( double (*f)( double ),
                       double (*f_)( double ),
                       double eps, double x ) {
  const double fx = f( x ) ;
  if( absolute( fx ) < eps ) {
    return x ;
  } else {
    return newton_raphson( f, f_, eps, x - fx/f_(x) ) ;
  }
}

double parabola( double x ) {
  return x * x - 2.0 ;
}

double parabola_( double x ) {
  return 2 * x ;
}

int main( void ) {
  printf( "%f\n", newton_raphson( parabola, parabola_,
                                  0.001, 1.5 ) ) ;
  printf( "%f\n", newton_raphson( parabola, parabola_,
                                  0.1, 200.0 ) ) ;
  return 0 ;
}
```

The second argument to `newton_raphson` represents the derivative of the function `f`. C does not permit the use of an apostrophe ( ) in an identifier, so we had to choose another character (_) instead.

(e) The execution of the C program will as always start by executing the first statement of `main`, a call to `printf`, which requires the value of `newton_raphson(...)` to be calculated. This function is invoked with four arguments; the arguments `eps` and `x` have the values `0.001` and `1.5`, while the arguments `f` and `f_` have the values `parabola` and `parabola_`. The first statement of `newton_raphson` is

```
const double fx = f( x ) ;
```

Because the argument `f` in this case is the function `parabola`, and `x` has the value `1.5`, this is effectively the same as:

```
const double fx = parabola( 1.5 ) ;
```

This results in the value `0.25`. Hence, the double `fx` has the value `0.25`. Because the absolute value of `0.25` is greater than `eps`, the function `newton_raphson` is recursively called with the same arguments as before, except `x`, which is now:

`x-fx/f_(x)` = `1.5 - 0.25/parabola_( 1.5 )`

This evaluates to about `1.41667`. Then the next iteration starts by computing `fx = f( x )`, which is about `0.006944444`. This is just larger than `eps`, so a third call to `newton_raphson` is performed with `x` equal to `1.4142156862`... which is correct to the first 5 digits.

(f) The Newton Raphson method does not terminate for $f(x) = \frac{1}{x} - 1$ when starting with $x_0 = 3$. The first new point is $x_1 = 3 - \frac{f(3)}{f'(3)} = -3$, the next point is $x_2 = -3 - \frac{f(-3)}{f'(-3)} = -15$. Each next point is further away from the root of the function.

Answers to the exercises of Chapter 3

Answer to 3.2: The SML and C versions of `leap` that take the extended range of years into account are:

```
(* leap : int -> int *)
fun leap y = if y mod 4 <> 0 orelse
                y mod 100 = 0 andalso y mod 400 <> 0
                then leap (y+1)
                else y ;
```

```
int leap( int y ) {
  if( (y % 4 != 0) ||
      (y % 100 == 0 && y % 400 != 0) ) {
    return leap( y+1 ) ;
  } else {
    return y ;
  }
}
```

Answer to 3.3: ♣ The table below shows the correspondence between the two versions of `leap`. The first column refers to the basic while-schema.

schema:	Functional	C
f:	`leap`	`leap`
t:	`int`	`int`
t_r:	`int`	`int`
x:	`y`	`y`
p:	`y mod 4 <> 0`	`y % 4 != 0`
g:	`y+1`	`y+1`
h:	`y`	`y`

Answer to 3.5: ♣ The correspondence between elements of the multiple argument while-schema and the version of `euclid` above is as follows:

schema:	Functional	C
f:	euclid	euclid
n:	2	2
(t_1*t_2):	(int*int)	(int,int)
t_r:	int	int
(x_1,x_2):	(m,n)	(m,n)
p:	n > 0	n > 0
g:	(n,m mod n)	(n,m % n)
h:	m	m

Answer to 3.6:

Just after executing	values of		
the code	m	n	old_n
int euclid(int m, int n) {	9	6	
while(n > 0) {	9	6	
const int old_n = n ;	9	6	6
n = m % old_n ;	9	3	6
m = old_n ;	6	3	6
while(n > 0) {	6	3	
const int old_n = n ;	6	3	3
n = m % old_n ;	6	0	3
m = old_n ;	3	0	3
while(n > 0) {	3	0	
return m ;	3	0	

The function returns the value 3.

Answer to 3.7: ♣ Proof of the hypothesis $\texttt{n}! = \texttt{fac n}$ by induction over n:

Case 1:

$$\begin{aligned} \texttt{1}! &= \prod_{i=1}^{1} \texttt{i} && \{!\} \\ &= \texttt{1} && \{\prod\} \\ &= \texttt{fac 1} && \{\texttt{fac.1}\} \end{aligned}$$

Case n+1:

$$\begin{aligned} \texttt{(n+1)}! &= \prod_{i=1}^{n+1} \texttt{i} && \{!\} \\ &= \texttt{(n+1) * } \prod_{i=1}^{n} \texttt{i} && \{\prod\} \\ &= \texttt{(n+1) * n}! && \{!\} \\ &= \texttt{(n+1) * fac n} && \{\text{hypothesis}\} \\ &= \texttt{fac (n+1)} && \{\texttt{fac.1}\}\square \end{aligned}$$

Answer to 3.8: Proof of the hypothesis `b * n!` = `fac_accu n b` by induction over `n`:

Case `1`:

```
b * 1!       = b * 1                      {!}
             = b                          {arithmetic}
             = fac_accu 1 b               {fac_accu}
```

Case `n+1`:

```
b * (n+1)!   = b * (n+1) * n!             {!}
             = (b * (n+1)) * n!           {associativity*}
             = fac_accu n (b * (n+1))     {hypothesis}
             = fac_accu n ((n+1) * b)     {commutativity*}
             = fac_accu (n+1) b           {fac_accu}□
```

Answer to 3.9: ♣ The correspondence between the elements of the function `fac_accu` and the schema (after tupling):

schema:	Functional	C
f:	`fac_accu`	`fac_accu`
n:	`2`	`2`
(t_1*t_2):	`(int*int)`	`(int,int)`
(x_1*x_2):	`(n,b)`	`(n,b)`
p:	`n > 1`	`n > 1`
g:	`(n-1,n*b)`	`(n-1,n*b)`
h:	`b`	`b`

Answer to 3.10: An efficient body of the while statement would be as shown below. It is not elegant, as the statements that compute the values of `m` and `f_m` appear in two places. This makes the code difficult to maintain, since it means that a change of the code has to be made twice.

```
if( f_m < 0.0 ) {
  l = m ;
} else {
  h = m ;
}
m = (l+h)/2.0 ;
f_m = f(m) ;
```

Answer to 3.11: The problem is that, before terminating, `main` wants to perform an operation on the value returned by bisection. A naive inlining operation would duplicate this operation. The solution is to break out of the while loop using a `break`, as shown below:

```
int main( void ) {
  double x ;
  double l = 0.0 ;
  double h = 2.0 ;
  double m ;
  double f_m ;
  while( true ) {
    m = (l+h)/2.0 ;
    f_m = parabola(m) ;
    if( absolute( f_m ) < eps ) {
      x = m ;
      break ;
    } else if( absolute( h-l ) < delta ) {
      x = m ;
      break ;
    } else if( f_m < 0.0 ) {
      l = m ;
    } else {
      h = m ;
    }
  }
  printf("The answer is %f\n", x ) ;
  return 0 ;
}
```

Note that this program can be optimised: `x` merely serves to bring the value of `m` out of the while-loop. If every instance of `m` is replaced by `x`, the program will be a bit shorter.

Answer to 3.12: The relationship between `repeat` on the one hand and the functions from this chapter on the other hand is:

$$\texttt{repeat}\ b\ c\ i\ n\ f = \texttt{foldr}\ c\ b\ \texttt{(map}\ f\ \texttt{(}i\texttt{--}n\texttt{))}$$

This shows that `repeat` combines the generation of a sequence of values with the accumulation of the result. In the approach that we have taken in this chapter, these two activities are separated. This increases the flexibility and the usefulness of the individual building blocks `foldr`, `map` and `--`.

Answer to 3.13: ♣ Proof of (3.4) by induction on the length of the list xs.

Case `[]`:

$$
\begin{array}{lll}
\multicolumn{2}{l}{\texttt{foldl}\ g\ b\ \texttt{(map}\ f\ \texttt{[])}} & \\
\quad = & \texttt{foldl}\ g\ b\ \texttt{[]} & \{\texttt{map.1}\} \\
\quad = & b & \{\texttt{foldl.1}\} \\
\quad = & \texttt{foldl}\ h\ b\ \texttt{[]} & \{\texttt{foldl.1}\}
\end{array}
$$

Case `(`x`::`xs`)`:

$$
\begin{array}{lll}
\multicolumn{2}{l}{\texttt{foldl}\ g\ b\ \texttt{(map}\ f\ \texttt{(}x\texttt{::}xs\texttt{))}} & \\
\quad = & \texttt{foldl}\ g\ b\ \texttt{(}f\ x\texttt{::map}\ f\ x\texttt{)} & \{\texttt{map.2}\} \\
\quad = & \texttt{foldl}\ g\ \texttt{(}g\ b\ \texttt{(}f\ x\texttt{))}\ \texttt{(map}\ f\ xs\texttt{)} & \{\texttt{foldl.2}\} \\
\quad = & \texttt{foldl}\ h\ \texttt{(}g\ b\ \texttt{(}f\ x\texttt{))}\ xs & \{\text{hypothesis}\} \\
\quad = & \texttt{foldl}\ h\ \texttt{(}h\ b\ x\texttt{)}\ xs & \{h\texttt{.1}\} \\
\quad = & \texttt{foldl}\ h\ b\ \texttt{(}x\texttt{:}xs\texttt{)} & \{\texttt{foldl.2}\}\square
\end{array}
$$

Answer to 3.14: Proof of (3.6) by induction on the length of the list xs.

```
Case []:
foldl g b (filter p [])
   = foldl g b []                               {filter.1}
   = b                                          {foldl.1}
   = foldl h b []                               {foldl.1}

Case (x:xs) and p x = false:
foldl g b (filter p (x::xs))
   = foldl g b (if p x
                  then x::filter p xs
                  else filter p xs)             {filter.2}
   = foldl g b (filter p xs)                    {p x = false}
   = foldl h b xs                               {hypothesis}
   = foldl h (if p x
                then g b x
                else b) xs                      {p x = false}
   = foldl h (h b x) xs                         {h.1}
   = foldl h b (x:xs)                           {foldl.2}

Case (x:xs) and p x = true:
foldl g b (filter p (x::xs))
   = foldl g b (if p x
                  then x::filter p xs
                  else filter p xs)             {filter.2}
   = foldl g b (x::filter p x)                  {p x = true}
   = foldl g (g b x) (filter p xs)              {foldl.2}
   = foldl h (g b x) xs                         {hypothesis}
   = foldl h (if p x
                then g b x
                else b) xs                      {p x = true}
   = foldl h (h b x) xs                         {h.1}
   = foldl h b (x:xs)                           {foldl.2}□
```

Answer to 3.16: The looping C version (using bit operation) reads:

```
int pop_count( int n ) {
  int accu = 0 ;
  while( n != 0 ) {
    accu = accu + (n&1) ;
    n = n >> 1 ;
  }
  return accu ;
}
```

```
int main( void ) {
  printf( "population count\n");
  printf( "of     0 is %d\n", pop_count( 0 ) ) ;
  printf( "of     9 is %d\n", pop_count( 9 ) ) ;
  printf( "of 65535 is %d\n", pop_count( 65535 ) ) ;
  return 0 ;
}
```

Answer to 3.17: Here is the C version of `power_of_power` embedded in a main program:

```
#include <stdio.h>

int power( int r, int p ) {
  int accu = 1 ;
  while( p != 0 ) {
    accu = accu * r ;
    p = p - 1 ;
  }
  return accu ;
}

int power_of_power( int m, int n ) {
  int accu = 1 ;
  while( n != 0 ) {
    accu = power( m, accu ) ;
    n = n - 1 ;
  }
  return accu ;
}

int main( void ) {
  printf( "power of power\n");
  printf( "0 17: %d\n", power_of_power( 0, 17 ) ) ;
  printf( "1 17: %d\n", power_of_power( 1, 17 ) ) ;
  printf( "2  0: %d\n", power_of_power( 2,  0 ) ) ;
  printf( "2  1: %d\n", power_of_power( 2,  1 ) ) ;
  printf( "2  2: %d\n", power_of_power( 2,  2 ) ) ;
  printf( "2  3: %d\n", power_of_power( 2,  3 ) ) ;
  printf( "2  4: %d\n", power_of_power( 2,  4 ) ) ;
  return 0 ;
}
```

Inlining `power` into `power_of_power` is not really desirable in this case. The function becomes messy, and a function `power` is a useful building block anyway. A first result would be:

```
int power_of_power( int m, int n ) {
  int accu = 1 ;
  while( n != 0 ) {
    int p = accu ;
    int power = 1 ;
    while( p != 0 ) {
      power = power * m ;
      p = p - 1 ;
    }
    accu = power ;
    n = n - 1 ;
  }
  return accu ;
}
```

Note that one of the variables `p` or `power` is, strictly speaking, superfluous: replacing all occurrences of `p` with `accu` would not modify the functionality of the program. Replacing all occurrences of `power` with `accu` would also be legal. However, you cannot remove both `p` and `power`.

Answer to 3.18: ♣ Here is the C function `chess_board` and three auxiliary functions:

```
bool odd( int i ) {
  return i % 2 == 1 ;
}

void dash( int width ) {
  int col ;
  for( col = 1; col <= width; col++ ) {
    printf( "--" ) ;
  }
  printf( "-\n" ) ;
}

void black_or_white( int row, int width ) {
  int col ;
  for( col = 1; col <= width; col++ ) {
    if( odd( row+col ) ) {
      printf( "|X" ) ;
    } else {
      printf( "| " ) ;
    }
  }
  printf( "|\n" ) ;
}
```

```
void chess_board( int width, int height ) {
  int row ;
  for( row = 1; row <= height; row ++ ) {
    dash( width ) ;
    black_or_white( row, width ) ;
  }
  dash( width ) ;
}
```

Answer to 3.19: ♣ The SML version with tupled arguments reads:

```
(* newton_raphson :
   (real->real)*(real->real)*real*real -> real *)
fun newton_raphson(f,f ,eps,b)
    = let
          val fb = f(b)
      in
          if absolute fb < eps
             then b
             else newton_raphson(f,f ,eps,b-fb/f (b))
      end ;
```

This neatly matches with the general tail recursion/while-schema. From the correspondence for the schema, it follows that the multiple assignment statement for the `newton_raphson` function should be:

```
(f,f_,eps,b) = (f,f_,eps,b - f(b)/f_(b)) ;
```

Only the value of `b` changes from one iteration of the while-loop to the next; the other three arguments `f`, `f_`, and `eps` remain unchanged. Therefore, the multiple assignment reduces to a single assignment-statement: `b = b - f(b)/f_(b);`. The C implementation of the `newton_raphson` function is therefore:

```
double newton_raphson( double (*f)( double ),
                       double (*f_)( double ),
                       double eps, double b ) {
  double fb ;
  while( true ) {
    fb = f(b) ;
    if( absolute( f(b) ) < eps ) {
      return b ;
    } else {
      b = b - fb / f_(b) ;
    }
  }
}
```

There is another C program with the same solution, using an assignment within an expression. This solution is often quoted as truly idiomatic C:

```
double newton_raphson( double (*f)( double ),
                       double (*f_)( double ),
                       double eps, double b ) {
  double fb ;
  while( absolute( fb = f(b) ) >= eps ) {
    b = b - fb/f_(b) ;
  }
  return b ;
}
```

The declaration `double fb;` allocates a cell in the store as usual. The value associated with `fb` should be the result of computing `f(b)`. The assignment statement `fb=f(b);` is performed in the middle of the condition of the while-statement. This is another use of assignments as expressions. We prefer the previous solution as assignments in conditional expressions do not lead to maintainable programs.

Answers to the exercises of Chapter 4

Answer to 4.1: In the drawings below we have used $\mathtt{d}_0$ and $\mathtt{d}_1$ to denote the two parts of `d`. Each of these parts occupies one storage cell. The structure `structx` will occupy 5 cells, and the union `unionx` requires two cells:

structx:

i
$\mathtt{d}_0$
$\mathtt{d}_1$
p.x
p.y

unionx:

i, $\mathtt{d}_0$, or p.x
$\perp$, $\mathtt{d}_1$, or p.y

Answer to 4.2: The SML datatype for a point in 2-dimensional space is defined in the following way:

```
datatype coordinate = Polar     of  real * real |
                      Cartesian of  real * real ;
```

The equivalent C type-declaration is:

```
typedef enum { Polar, Cartesian } coordinatetype ;

typedef struct {
  coordinatetype tag ;
  union {
    struct {
      double x ;
      double y ;
    } cartesian ;
    struct {
      double r ;
      double theta ;
    } polar ;
  } contents ;
} coordinate ;
```

For an argument, say `p`, of the type `coordinate`, the type of the coordinate is stored in `p.tag`. If the type of the coordinate is `Polar`, then the radius is stored in `p.contents.polar.r`, and the angle is stored in `p.contents.polar.theta`. If it is a `Cartesian` coordinate, the X and Y-coordinates are stored in `p.contents.cartesian.x` and `p.contents.cartesian.y`.

Answer to 4.4: The type pointer to pointer to double can be defined as follows:

```
typedef double ** pointer_to_pointer_to_double ;
```

Answer to 4.5: The type `type0` is a structure containing two elements: one is a pointer to a floating point value `x`, the other is an integer `y`. The type `type1` is a pointer to a `type0` structure.

Answer to 4.6: This question is not as easy as it might look:

- **(a)** Yes, `*&i` has the same value as `i`: `&i` is a pointer to `i`, dereferencing (following) this pointer results in the value of `i` (`123`).
- **(b)** The expression `*&`x is only legal if x is an object with an address (because of the definition of the prefix-& operator). For all x for which &x is legal, `*&`x and x denote the same value. In the example functions, the legal values for x are `i`, `p`, `q`, and `r`.
- **(c)** The expressions `&*p` and `p` denote the same value in almost all cases. In our example, `p` is a pointer to `i`, so the object `*p` has an address, namely the address of `i`. This is the same value as `p`. In cases where `p` contains an illegal address (they will be explained later), the expression `*p` is illegal, hence `&*p` does not have the same value.
- **(d)** The expressions `&*`z and z are only equal if z is a legal pointer-value. In any other case, for example, if z is the integer `12` or some illegal pointer value, the expressions are not equal. In the example functions, the equality is true for z equals `p`, `q`, or `r`.

Answer to 4.7: The full code could read:

```
#include <stdio.h>

typedef struct {
  double x ;
  double y ;
} point ;

void print_point( point *p ) {
  printf( "(%f,%f)\n", p->x, p->y ) ;
}

void rotate( point *p ) {
  double ty = p->y ;
  p->y = -p->x ;
  p->x = ty ;
}

int main( void ) {
  point r0 = {   1, 0 } ;
  point r1 = {  -0.1, -0.3 } ;
  point r2 = { 0.7, -0.7 } ;
  rotate( &r0 ) ;
  print_point( &r0 ) ;
  rotate( &r1 ) ;
  print_point( &r1 ) ;
  rotate( &r2 ) ;
  print_point( &r2 ) ;
  return 0 ;
}
```

Pay special attention to the implementation of `rotate`. Because both members of the input are necessary in the computation of both output members, one needs to introduce a temporary variable, `ty`. It can be implemented in a language with a multiple assignment as:

```
(p->y,p->x) = (-p->x,p->y) ;
```

Answer to 4.10: ♣

(a) The * operator expects a pointer and dereferences the pointer. This yields the value pointed at. The & operator expects a value that it can work out the address of and returns that address as a pointer. The * operator could be be viewed as undoing the effect of the & operator.

(b) The answer printed by the `main` function is two times the number 3, then the number 1 (for true), the number 3, two times the number 1 (for true), and again the number 3.

(c) The program would print 4 as a result of the following function calls. Firstly:

```
twice( add, 2 ) == add( add( 2 ) )
```

Secondly:

```
add( add( 2 ) ) == add( 3 )
```

And finally:

```
add( 3 ) == 4
```

(d) The differences are the following:

- Program (c) prints 4 and Program (d) prints 14.
- Program (c) can handle only functional arguments with one argument; Program (d) can handle only functional arguments with two arguments.

(e) The differences are the following:

- Program (e) can handle functional arguments with any number of arguments and is thus more general than either Program (c) or Program (d).
- Program (e) prints the same answer as Program (d).

Answer to 4.13: ♣

(a) Below we define a point, and a bounding box in terms of two points:

```
typedef struct { int x, y ; } point ;
typedef struct { point ll, ur ; } bbox ;
```

(b) The function to normalise makes a bounding box, and ensures that the `ll` field contains the minimum x and y values, while the `ur` contains the maximum values:

```
bbox normalise( bbox b ) {
  bbox r ;
  if( b.ll.x < b.ur.x ) {
    r.ll.x = b.ll.x ;
    r.ur.x = b.ur.x ;
  } else {
    r.ur.x = b.ll.x ;
    r.ll.x = b.ur.x ;
  }
  if( b.ll.y < b.ur.y ) {
    r.ll.y = b.ll.y ;
    r.ur.y = b.ur.y ;
  } else {
    r.ur.y = b.ll.y ;
    r.ll.y = b.ur.y ;
  }
  return r ;
}
```

(c) The combining function returns the minimum `x` and `y` values from the two `ll` points of the input boxes in the new `ll` point, and the maximum in the `ur` point:

```
bbox combine( bbox b, bbox c ) {
  bbox r ;
  r.ll.x = b.ll.x < c.ll.x ? b.ll.x : c.ll.x ;
  r.ll.y = b.ll.y < c.ll.y ? b.ll.y : c.ll.y ;
  r.ur.x = b.ur.x > c.ur.x ? b.ur.x : c.ur.x ;
  r.ur.y = b.ur.y > c.ur.y ? b.ur.y : c.ur.y ;
  return r ;
}
```

Note the use of the conditional operator …`?`…`:`… to return the minimum/maximum value.

(d-g) Below are all the graphic elements. Note that we have defined separate types for `line`, `rect` and `bbox`, although they all have the same members. These data types serve different purposes and therefore it is better to make them separate types.

```
typedef struct { point from, to ; } line ;
typedef struct { point centre ; int radius  ; } circ ;
typedef struct { point c1, c2 ; } rect ;

typedef enum  { Line, Circ, Rect } elementtype ;
typedef struct {
  elementtype tag ;
  union {
    line l ;
    circ c ;
    rect r ;
  } el ;
} element ;
```

(h) The function to calculate the bounding boxes must perform some normalisations. This is neccessary even when handling a circle, as the radius may be negative.

```
bbox bbox_of_element( element e ) {
  bbox r ;
  switch( e.tag ) {
    case Rect :
      r.ll = e.el.r.c1 ;
      r.ur = e.el.r.c2 ;
      return normalise( r ) ;
    case Line :
      r.ll = e.el.l.from ;
      r.ur = e.el.l.to ;
      return normalise( r ) ;
    case Circ :
      r.ll.x = e.el.c.centre.x - e.el.c.radius ;
      r.ll.y = e.el.c.centre.y - e.el.c.radius ;
      r.ur.x = e.el.c.centre.x + e.el.c.radius ;
      r.ur.y = e.el.c.centre.y + e.el.c.radius ;
      return normalise( r ) ;
    default :
      abort() ;
  }
}
```

(i) The main program finally creates three objects and calls `bbox_of_element` and `combine` to calculate a final bounding box:

```
int main( void ) {
  rect r = { {3,5}, {5,4} } ;
  circ c = { {2,2}, 1 } ;
  line l = { {2,2}, {6,1} } ;
  element e ;
  bbox b;
  e.tag = Rect ; e.el.r = r ;
  b = bbox_of_element( e ) ;
  e.tag = Line ; e.el.l = l ;
  b = combine( b, bbox_of_element( e ) ) ;
  e.tag = Circ ; e.el.c = c ;
  b = combine( b, bbox_of_element( e ) ) ;
  printf("Bounding box: (%d,%d) (%d,%d)\n",
         b.ll.x, b.ll.y, b.ur.x, b.ur.y ) ;
  return 0 ;
}
```

Answers to the exercises of Chapter 5

Answer to 5.1: ♣ An SML function to concatenate two arrays `s` and `t` is:

```
(* concatenate :  a array *  a array ->  a array *)
fun concatenate(s,t)
    = let
          val n_s = length(s)
          val n_t = length(t)
          fun f i = if i < n_s
                       then sub(s,i)
                       else sub(t,i-n_s)
      in
          tabulate(n_s+n_t,f)
      end ;
```

Answer to 5.2: ♣ An SML function to return the data of the array `s` from index `l` to index `u` is:

```
(* slice :  a array * int * int ->  a array *)
fun slice(s,l,u) = let
                       fun f i = sub(s,i+l)
                    in
                       tabulate(u-l+1,f)
                    end ;
```

Answer to 5.4: Here is a complete C program that simulates the card game. Please note that the function `player_A` does not read `secret`, but *only* passes it on to `player_B`.

```
#include <stdlib.h>

typedef enum { false=0, true=1 } bool ;

#define n_number 4
#define n_card 3
typedef int deck[n_card][n_number] ;

const deck card = {{1,3,5,7},{2,3,6,7},{4,5,6,7}} ;

bool player_B( int c, int n ) {
  int i ;
```

```
    for( i = 0; i < n_number; i++ ) {
      if( card[c][i] == n ) {
        return true ;
      }
    }
    return false ;
  }

  int player_A( int secret ) {
    int guess = 0 ;
    int power = 1 ;
    int c ;
    for( c = 0; c < n_card; c++ ) {
      if( player_B( c, secret ) ) {
        guess += power ;
      }
      power *= 2 ;
    }
    return guess ;
  }

  int main( int argc, char * argv [] ) {
    if( argc != 2 ) {
      printf( "usage: %s [0..7]\n", argv[0] ) ;
    } else {
      int secret = argv[1][0] -  0  ;
      printf( "the secret number is: %d\n", player_A( secret ) ) ;
    }
    return 0 ;
  }
```

What would happen if the program was tried with 8 or 9 as the secret number?

Answer to 5.6: In all other cases, the programmer can argue that the index is within range. During the initialisation of `hist` the integer `i` runs from `0` to `u-l`, hence it is within the bounds. The index `i` in `input[ i ]` is running from `0` to some number, but as long as the NULL character has not been seen it is within the bounds. Similarly, the variable `i` is in the range `0` … `u-l-1` when printing the histogram.

Note that if it was known a priori that all characters in the input array are always in the range `a` … `z` , then the bound check before updating `input[ i ]` could be removed.

Answer to 5.7: ♣ An SML version of `histogram` using a dynamic array is:

```
  (* inc : dynamic_array -> int -> dynamic_array *)
  fun inc (a,l,u) i = (upd(a,i,sub(a,i)+1),l,u) ;
```

```
(* histogram : char array -> int -> int -> dynamic_array *)
fun histogram input l u
    = let
         val n = length input
         val lb = ord(sub(input,0))
         val empty = array_alloc lb lb
         fun tally (hist as (_,hist_lb,hist_ub)) i
             = let
                  val c = ord(sub(input,i))
               in
                  if c < hist_lb
                     then inc (extend hist c hist_ub) (c-l)
                     else if c > hist_ub
                             then inc (extend hist hist_lb c) (c-l)
                             else inc hist (c-l)
               end
      in
         foldl tally empty (0 -- n-1)
      end ;
```

Answer to 5.10: ♣

(a) The function `print_three` prints the first three characters of its (string) argument. Strings are conventionally terminated by a null character. If the string contains fewer than three characters this can be detected.

```
void print_three( char s [] ) {
  int i ;
  for( i = 0; i < 3; i++ ) {
    if( s[i] ==  \0  ) {
      return ;
    } else {
      printf( "%c", s[i] ) ;
    }
  }
}
```

(b) The data structure `month` below has two fields: one to store the number of days in a month and the other to store the name of the month. Information from the application domain is used to limit the size of the strings used to one more than the maximum number of characters in a month (1+9).

```
typedef struct {
          int  days ;
          char name[10] ;
```

```
        } month ;
```

The array `leap` as declared below has 12 entries because there are 12 months in a year:

```
month leap [12] = { {31, "January"},
                    {29, "February"},
                    {31, "March"},
                    {30, "April"},
                    {31, "May"},
                    {30, "June"},
                    {31, "July"},
                    {31, "August"},
                    {30, "September"},
                    {31, "October"},
                    {30, "November"},
                    {31, "December"} } ;
```

(c) The function `print_year` below prints a table of the name and the number of days for each month of a year.

```
void print_year( month a_year [] ) {
  int i ;
  int total_days = 0 ;
  for( i = 0; i < 12; i++) {
    total_days += a_year[i].days ;
    print_three( a_year[i].name ) ;
    printf( ". has %d days\n", a_year[i].days ) ;
  }
  printf( "This year has %d days\n", total_days ) ;
}
```

Answer to 5.11: Here is a complete C program that includes both a discrete version of `extra_bisection` and a main program to test it.

```
#include <stdlib.h>
#include <string.h>

typedef enum { false=0, true=1 } bool ;

int extra_bisection( int (*f)( void *, int ),
                     void *x, int l, int h  ) {
  int m ;
  int f_m ;
  while( true ) {
    m = (l + h) / 2 ;
    f_m = f( x, m ) ;
    if(  f_m == 0 ) {
      return m ;
    } else if( h-l <= 1 ) {
      return m ;
    } else if( f_m < 0 ) {
      l = m ;
    } else {
      h = m ;
    }
  }
}

int direction( void *p, int i ) {
  char ** data = p ;
  return strcmp( data[i] , data[0] ) ;
}

int main( int argc, char *argv[] ) {
  int i = extra_bisection( direction, argv, 1, argc ) ;
  printf( "%d\n", i ) ;
  return 0 ;
}
```

Answer to 5.12:

```
int findbreak( void *p, int i ) {
  if( brk( i ) == 0 ) {
    return -1 ;
  }
  return 1 ;
}

int main( void ) {
  int b = extra_bisection( findbreak, NULL, 0, 0x7FFFFFFF ) ;
  printf( "Break: %x\n", b ) ;
  return 0 ;
}
```

Answer to 5.13: ♣

(a) Here are two appropriate constants and a `typedef` suitable to store the exam results.

```
#define max_student 130
#define max_module  12
```

```
typedef int exam [max_student] [max_module] ;
```

(b) Here is a function `total` that computes the sum of all the scores for a particular student:

```
double total( exam table, int student ) {
  int m ;
  int t = 0 ;
  for( m = 0; m < max_module; m++ ) {
    t += table[student][m] ;
  }
  return t ;
}
```

(c) Here is a function `print` to display the exam results nicely formatted:

```
void print( exam table ) {
  int s, m ;
  bool ok = true ;
  for( s = 0; s < max_student; s++ ) {
    int n = non_zero( table, s ) ;
    if( n > 0 ) {
      int t = total( table, s ) ;
      printf( "%4d:", s ) ;
      for( m = 0; m < max_module; m++ ) {
        printf( "%3d", table[s][m] ) ;
        if( table[s][m] < 0 || table[s][m] > 100 ) {
          printf( "?" ) ;
          ok = false ;
        } else {
          printf( " " ) ;
        }
      }
      printf( "%4d %4d %6.2f\n", t, n, (double) t /
                                        (double) n ) ;
    }
  }
  printf( "exam results " ) ;
  if( ok ) {
    printf( "ok\n" ) ;
  } else {
    printf( "not ok\n" ) ;
  }
}
```

The `print` function uses an auxiliary function:

```
int non_zero( exam table, int student ) {
  int m ;
```

```
    int n = 0 ;
    for( m = 0; m < max_module; m++ ) {
      if( table[student][m] != 0 ) {
        n ++ ;
      }
    }
    return n ;
  }
```

(d) Here is a main program that initialises the table to some arbitrary values and then prints it.

```
int main( void ) {
  { exam table = {{1,2,3},{4,5,6,7},{0},{8},{-3}} ;
    print( table ) ;
  }
  { exam table = {{1,2,3},{4,5,6,7},{0},{8},{3}} ;
    print( table ) ;
  }
  return 0 ;
}
```

(e) Code has been included to check that when printing the table, each score is within the permitted range. The program prints ″exam results ok″ if all scores are within range and it prints ″exam results not ok″ otherwise. Each offending score is also accompanied by a question mark.

Answer to 5.14: ♣ Here are the C data structures required to support the spread sheet.

(a) The maximum number of work sheets is `max_sheet`:

```
#define max_sheet 5
typedef sheet work[max_sheet] ;
```

(b) The name, date and time identify a work sheet:

```
#define max_name 10
typedef char name[max_name] ;

typedef struct {
  int the_day ;
  int the_month ;
  int the_year ;
} date ;

typedef struct {
  int the_hour ;
  int the_minute ;
```

```
    } time ;
```

(c) A work sheet is a collection of cells with their identification:

```
    #define max_row 10
    #define max_column 10
    typedef struct {
      name the_name ;
      date the_date ;
      time the_time ;
      cell the_cell[max_row][max_column] ;
    } sheet ;
```

(d) Here is the type of all possible cell values:

```
    #define max_formula 80
    typedef char formula[max_formula] ;

    typedef union {
      formula the_formula ;
      int     the_int ;
      float   the_real ;
      bool    the_bool ;
    } value ;
```

(e) Here are the four different cell kinds:

```
    typedef enum { Formula, Int, Real, Bool } kind ;
```

(f) A cell has a flag to tell whether it is in use:

```
    typedef struct {
      bool  in_use ;
      kind  the_kind ;
      value the_value ;
    } cell ;
```

Answer to 5.15: ♣ Here is a C program to print magic squares of order $1, 3, 5 \ldots 17$.

```
#include <stdio.h>

#define maxmax 17
typedef int magic[maxmax][maxmax] ;

void zero( magic square, int max ) {
  int row, col ;
  for( row = 0; row < max; row ++ ) {
    for( col = 0; col < max; col++ ) {
      square[row][col] = 0 ;
    }
  }
}
```

```
void fill( magic square, int max ) {
  int row = 0 ;
  int col = max / 2 ;
  int cnt = 0 ;
  while( cnt < max*max ) {
    if( square[row][col] == 0 ) {
      cnt++ ;
      square[row][col] = cnt ;
      col = (col + max - 1) % max ;
      row = (row + max - 1) % max ;
    } else {
      col = (col + 1) % max ;
      row = (row + 2) % max ;
    }
  }
}

void print( magic square, int max ) {
  int row, col ;
  for( row = 0; row < max; row ++ ) {
    for( col = 0; col < max; col++ ) {
      printf( "%4d", square[row][col] ) ;
    }
    printf( "\n" ) ;
  }
}

int main( void ) {
  int max ;
  magic square ;
  for( max = 1; max <= maxmax; max += 2 ) {
    zero( square, max ) ;
    fill( square, max ) ;
    print( square, max ) ;
    printf( "\n" ) ;
  }
  return 0 ;
}
```

Answers to the exercises of Chapter 6

Answer to 6.1: ♣ The following C procedure prints a character list using a while-statement.

```
void print_list( char_list x_xs ) {
  while( x_xs != NULL ) {
    printf( "%c", head( x_xs ) ) ;
    x_xs = tail( x_xs ) ;
  }
}
```

Answer to 6.2:

```
typedef enum { Cons, Nil } char_list_tags ;

typedef struct char_list {
  char_list_tags tag ;
  union {
    struct {
      char                list_head ;
      struct char_list  * list_tail ;
    } cons_cell ;
  } char_list_union ;
} *char_list ;
```

The alternative, corresponding to the tag Nil, does not hold any information. In such a case, where there is only one alternative that holds information, the union is not necessary, and the type can be simplified to:

```
typedef enum { Cons, Nil } char_list_tags  ;
typedef struct char_list {
  char_list_tags      tag ;
  char                list_head ;
  struct char_list  * list_tail ;
} *char_list ;
```

Each list must now be terminated with an extra element with a tag-value Nil. The inefficiency of managing these cells that terminate lists is the reason that C uses a special pointer value NULL to terminate a list.

Answer to 6.3: ♣ A tail recursive version of length is:

```
(* length : char_list -> int *)
fun length x_xs       = length  0 x_xs
```

```
(* length  : int -> char_list -> int *)
and length  accu x_xs = if x_xs = Nil
                           then accu
                           else length  (accu+1) (tail x_xs) ;
```

The while-schema of Chapter 3 and suitable simplification yields the following efficient C implementation.

```
int length( char_list x_xs ) {
  int accu = 0 ;
  while( x_xs != NULL ) {
    accu++ ;
    x_xs = tail( x_xs ) ;
  }
  return accu ;
}
```

Answer to 6.4: The C implementation of `nth` shown below aborts with an error message when a non-existent list element is accessed. It is the result of using the general while-schema of Chapter 3.

```
char nth( char_list x_xs, int n ) {
  while( true ) {
    if( x_xs == NULL ) {
      printf( "nth\n" ) ;
      abort() ;
    }
    if ( n == 0 ) {
      return head( x_xs ) ;
    }
    x_xs = tail( x_xs ) ;
    n-- ;
  }
}
```

Answer to 6.5: ♣ A recursive definition of `append` without pattern matching is:

```
(* append : char_list -> char_list -> char_list *)
fun append x_xs ys
    = if x_xs = Nil
         then ys
         else Cons(head x_xs,append (tail x_xs) ys) ;
```

Answer to 6.6: Filtering a sequence s to select only the elements that satisfy a certain predicate p can be specified as follows:

$$\begin{aligned} \text{filter} &: (\alpha \to \mathbb{B} \times \mathbb{N} \to \alpha) \to (\mathbb{N} \to \alpha) \\ \text{filter}(p, s) &= \text{squash}(\{i \mapsto s(i) \mid i \in \text{domain}(s) \wedge p(s(i))\}) \end{aligned}$$

The set comprehension above does not produce a sequence, but a mere set of maplets. It is not a sequence because there may be 'holes' in the domain. The auxiliary function 'squashes' the domain so that all the holes are removed. Here is the definition of squash:

$$\begin{aligned} \text{squash} &: (\mathbb{N} \to \alpha) \to (\mathbb{N} \to \alpha) \\ \text{squash}(s) &= \{(i-1) \mapsto s(k) \mid k \in \text{domain}(s) \wedge i = \#(\{0 \ldots k\} \cap \text{domain}(s))\} \end{aligned}$$

Answer to 6.7: ♣ Mapping a function f over all elements of a sequence s delivers the following sequence:

$$\begin{aligned} \text{map} &: (\alpha \to \beta \times \mathbb{N} \to \alpha) \to (\mathbb{N} \to \beta) \\ \text{map}(f, s) &= \{i \mapsto f(s(i)) \mid i \in \text{domain}(s)\} \end{aligned}$$

Answer to 6.9: The general version of `map` is `extra_map`:

```
char_list extra_map( char (*f)( void *, char ),
                     void * arg, char_list x_xs ) {
  if( x_xs == NULL ) {
    return NULL ;
  } else {
    char x = head( x_xs ) ;
    char_list xs = tail( x_xs ) ;
    return cons( f( arg, x ), extra_map( f, arg, xs ) ) ;
  }
}
```

Answer to 6.10: ♣ A version of `copy` that traverses the input list from the right, using `foldr`, is:

```
(* copy : char_list -> char_list *)
fun copy xs = let
                 val n = length xs
                 fun copy  i accu = Cons(nth xs i,accu)
              in
                 foldr copy  Nil (0 -- n-1)
              end ;
```

The for-schema can be used to translate this function into C. After simplification this yields:

```
char_list copy( char_list xs ) {
  int i ;
  int n = length( xs ) ;
  char_list accu = NULL ;
  for( i = n-1; i >= 0; i-- ) {
    accu = cons( nth( xs, i ), accu ) ;
  }
  return accu ;
}
```

Answer to 6.11: Using the similarity between copy and append, the following programming can be derived straightforwardly:

```
char_list append( char_list xs, char_list ys ) {
  char_list accu = ys ;
  char_list *last = &accu ;
  while( xs != NULL ) {
    char_list new = cons( head( xs ), ys ) ;
    *last = new ;
    last = &new->list_tail ;
    xs = tail( xs ) ;
  }
  return accu ;
}
```

Answer to 6.12: ♣ Using the similarity between map and copy, the following function can be derived:

```
char_list map( char (*f)( char ), char_list xs ) {
  char_list accu = NULL ;
  char_list *last = &accu ;
  while( xs != NULL ) {
    char_list new = cons( f( head( xs ) ), NULL ) ;
    *last = new ;
    last = &new->list_tail ;
    xs = tail( xs ) ;
  }
  return accu ;
}
```

Note that by using the advanced pointer technique the resulting function is shorter.

Answer to 6.13: Using the similarity between `copy` and `filter`, the following program can be derived:

```
char_list filter( bool (*pred)( char ), char_list xs ) {
  char_list accu = NULL ;
  char_list *last = &accu ;
  while( xs != NULL ) {
    const char x = head( xs ) ;
    if( pred( x ) ) {
      char_list new = cons( x, NULL ) ;
      *last = new ;
      last = &new->list_tail ;
    }
    xs = tail( xs ) ;
  }
  return accu ;
}
```

Answer to 6.14: ♣ The recursive function that transfers the elements of an array into a list follows. It is inefficient, since it needs an amount of stack space proportional to the length of the array. The solution uses an auxiliary function `traverse` to traverse the index range `l` … `u`.

```
(* array_to_list : char array -> char_list *)
fun array_to_list s
    = let
         val l = 0
         val u = length s - 1
         fun traverse s i u
             = if i < u
                  then Cons(sub(s, i), traverse s (i+1) u)
                  else Nil
      in
         traverse s l u
      end ;
```

The translation of `traverse` to C is an application of the multiple argument while-schema:

```
char_list traverse( char s[], int i, int u ) {
  if( i < u ) {
    return cons( s[i], traverse( s, i+1, u ) ) ;
  } else {
    return NULL ;
  }
}
```

```
char_list array_to_list( char s[], int n ) {
  int l = 0 ;
  int u = n - 1;
  return traverse( s, l, u ) ;
}
```

Answer to 6.18: The specification of `reverse` is:

$$\begin{array}{rcl} \text{reverse} & : & (\mathbb{N} \to \alpha) \to (\mathbb{N} \to \alpha) \\ \text{reverse}(s) & = & \{i \mapsto s(\#s - 1 - i) \mid i \in \text{domain}(s)\} \end{array}$$

Here is an efficient C function to reverse a list:

```
char_list reverse( char_list x_xs ) {
  char_list accu = NULL ;
  while( x_xs != NULL ) {
    accu = cons( head( x_xs ), accu ) ;
    x_xs = tail( x_xs ) ;
  }
  return accu ;
}
```

Answer to 6.19: ♣

(a) Here are the C `#define` and `typedef` declarations that describe a binary tree with integers at the leaves:

```
typedef enum {Branch, Leaf} tree_tag ;

#define tree_struct_size sizeof( struct tree_struct )

typedef struct tree_struct {
  tree_tag tag ;
  union {
    int leaf ;                          /* tag Leaf */
    struct  {                           /* tag Branch */
      struct tree_struct * left ;
      struct tree_struct * right ;
      } branch ;
    } alt ;
  } * tree_ptr ;
```

(b) The function `mkLeaf` creates a leaf node of a binary tree:

```
tree_ptr mkLeaf( int leaf ) {
  tree_ptr tree ;
  tree = malloc( tree_struct_size ) ;
  if( tree == NULL ) {
    printf( "mkLeaf: no space\n" ) ;
    abort( ) ;
  }
  tree->tag = Leaf ;
  tree->alt.leaf = leaf ;
  return tree ;
}
```

The function `mkBranch` creates an interior node of a binary tree:

```
tree_ptr mkBranch( tree_ptr left, tree_ptr right ) {
  tree_ptr tree ;
  tree = malloc( tree_struct_size ) ;
  if( tree == NULL ) {
    printf( "mkBranch: no space\n" ) ;
    abort( ) ;
  }
  tree->tag = Branch ;
  tree->alt.branch.left = left ;
  tree->alt.branch.right = right ;
  return tree ;
}
```

(c) Here is a function to print a binary tree:

```
void print( tree_ptr tree ) {
  switch( tree->tag ) {
  case Leaf :
    printf( "%d", tree->alt.leaf ) ;
    break ;
  case Branch :
    printf( "(" ) ;
    print( tree->alt.branch.left ) ;
    printf( "," ) ;
    print( tree->alt.branch.right ) ;
    printf( ")" ) ;
    break ;
  }
}
```

(d) Here is an SML function to rotate a tree:

```
(* rotate : tree -> tree *)
```

```
fun rotate (Leaf(key))   = Leaf(key)
  | rotate (Branch(l,r)) = Branch(rotate r,rotate l) ;
```

The C function below rotates a tree without changing the input tree.

```
tree_ptr rotate( tree_ptr tree ) {
  switch( tree->tag ) {
  case Leaf :
    return mkLeaf ( tree->alt.leaf ) ;
  case Branch :
    return mkBranch ( rotate ( tree->alt.branch.right ),
                      rotate ( tree->alt.branch.left  ) ) ;
  }
}
```

Answer to 6.20:

(a) The type `tree_ptr` defines a pointer to a structure that can either accommodate a binding or a node of a tree. Each tree node contains a `key` field that will correspond to the constructor in SML, and it contains a (pointer to an) array with sub trees. The number of sub trees is stored in the `size`-member.

(b) The functions `mkBind` and `mkData` allocate a `tree_struct`. Both functions print out the data such that the execution of the functions can be traced. The pointers returned by `malloc` should be checked for a `NULL` value.

```
tree_ptr mkBind( tree_ptr * b ) {
  tree_ptr tree ;
  tree = malloc( sizeof( struct tree_struct ) ) ;
  tree->tag = Bind ;
  tree->alt.bind = b ;
  printf( "mkBind( %p ): %p\n", b, tree ) ;
  return tree ;
}

tree_ptr mkData( char k, int s, ... ) {
  va_list ap ;
  int i ;
  tree_ptr tree ;
  tree = malloc( sizeof( struct tree_struct ) ) ;
  tree->tag = Data ;
  tree->alt.comp.key = k ;
  tree->alt.comp.size = s ;
  tree->alt.comp.data = calloc( s, sizeof( tree_ptr ) ) ;
  printf( "mkData( %d, %d", k, s ) ;
  va_start( ap, s ) ;
  for( i = 0; i < s; i++ ) {
    tree_ptr d = va_arg( ap, tree_ptr ) ;
    tree->alt.comp.data[i] = d ;
    printf( ", %p", d ) ;
  }
  va_end( ap ) ;
  printf( " ): %p\n", tree ) ;
  return tree ;
}
```

The %p format prints a pointer.

(c) The function match performs case analysis on the argument pat. If it is a binding, the value of exp is stored in the appropriate location. If we are dealing with Data, sub trees will be matched, provided that the key and size of the pattern and expression agree.

```
bool match( tree_ptr pat, tree_ptr exp ) {
  switch( pat->tag ) {
    case Bind :
      * (pat->alt.bind) = exp ;
      return true ;
    case Data :
      if ( exp->tag == Data &&
           exp->alt.comp.key == pat->alt.comp.key &&
           exp->alt.comp.size == pat->alt.comp.size ) {
        int i ;
        for( i = 0; i < pat->alt.comp.size; i++ ) {
          if( ! match( pat->alt.comp.data[i],
                       exp->alt.comp.data[i] ) ) {
            return false ;
          }
        }
        return true ;
      } else {
        return false ;
      }
  }
  abort() ;
}
```

(d) The main function below creates a pattern and an expression. It then tries to match both, which will succeed. This binds the sub tree with key B to the variable b, the sub tree C to c and the sub tree D to d. The second call to match fails, because the keys and sizes of pat and b disagree. The third call to match also fails, because the keys of pat and c disagree, even though their sizes do agree. The last call to match succeeds.

```
int main( void ) {
  tree_ptr a, b, c, d ;
  tree_ptr exp = mkData(  A , 3,
                   mkData(  B , 0 ),
                   mkData(  C , 0 ),
                   mkData(  D , 3,
                     mkData(  E , 0 ),
                     mkData(  F , 0 ),
                     mkData(  G , 0 ) ) ) ;
  tree_ptr pat = mkData(  A , 3,
                   mkBind( &b ),
                   mkBind( &c ),
                   mkBind( &d ) ) ;
  if( match( pat, exp ) ) {
    printf( "1: b=%p, c=%p, d=%p\n", b, c, d ) ;
  }
  if( match( pat, b ) ) {
    printf( "2: b=%p, c=%p, d=%p\n", b, c, d ) ;
  }
  if( match( pat, c ) ) {
    printf( "3: b=%p, c=%p, d=%p\n", b, c, d ) ;
  }
  if( match( mkBind( &a ), exp ) ) {
    printf( "4: a=%p\n", a ) ;
  }
  return 0 ;
}
```

Answers to the exercises of Chapter 7

Answer to 7.2: ♣ Here is `stream_to_list` using pointers to pointers.

```
char_list stream_to_list( FILE * stream ) {
  char_list accu = NULL ;
  char_list *last = &accu ;
  int c ;
  while( ( c = getc( stream ) ) != EOF ) {
    char_list new = cons( c, NULL ) ;
    *last = new ;
    last = & new->list_tail ;
  }
  return accu ;
}
```

Answer to 7.4: ♣ Here is a side effecting SML function to output a list to a stream:

```
(* list_to_stream : outstream -> char list -> unit *)
fun list_to_stream stream []
    = ()
  | list_to_stream stream (x::xs)
    = (output(stream,x); list_to_stream stream xs) ;
```

The C equivalent is:

```
void list_to_stream( FILE * stream, char_list list ) {
  while( list != NULL ) {
    putc( head( list ), stream ) ;
    list = tail( list ) ;
  }
}
```

Answer to 7.6: ♣ A tail recursive version of `word_count` is:

```
(* word_count : char list -> char list -> int *)
fun word_count ws ts
    = let
           fun count accu ws ts
               = if ts = []
                     then accu
                     else if match ws ts
                             then count (accu+1) ws (tail ts)
```

```
                              else count accu     ws (tail ts)
      in
         count 0 ws ts
      end ;
```

The C implementation of the tail-recursive `word_count` is:

```
int word_count( char_list ws, char_list ts ) {
  int accu = 0 ;
  while( true ) {
    if( ts == NULL ) {
      return accu ;
    } else {
      if( match( ws, ts ) ) {
        accu++ ;
      }
      ts = tail( ts ) ;
    }
  }
}
```

Answer to 7.7: ♣ An efficient C implementation of `match` is:

```
bool match( char_list ws, char_list ts ) {
  while( true ) {
    if( ws == NULL ) {
      return true ;
    } else if( ts == NULL ) {
      return false ;
    } else if( head( ws ) == head( ts ) ) {
      ws = tail( ws ) ;
      ts = tail( ts ) ;
    } else {
      return false ;
    }
  }
}
```

Answer to 7.8: ♣ Here is a `main` program to count the number of occurrences of the word "cucumber" on the `stdin` stream:

```
int main( void ) {
  char_list word = array_to_list( "cucumber", 8 ) ;
  char_list text = stream_to_list( stdin ) ;
  printf( "%d\n", word_count( word, text ) ) ;
  return 0 ;
}
```

Answer to 7.9: Here is the SML version of `stream_to_list` which accesses at most `n` elements:

```
(* stream_to_list : instream -> int -> char list *)
fun stream_to_list stream n
    = if end_of_stream stream orelse n = 0
         then []
         else input(stream,1) :: stream_to_list stream (n-1) ;
```

The corresponding C version is:

```
char_list stream_to_list( FILE * stream, int n ) {
  int c = getc( stream ) ;
  if( c == EOF || n == 0 ) {
    return NULL ;
  } else {
    return cons( c, stream_to_list( stream, n-1 ) ) ;
  }
}
```

Answer to 7.10: ♣ Here is a main program to initialise the queue and count the number of occurrences of some word in a text.

```
int main( void ) {
  char_list word = array_to_list( "cucumber", 8 ) ;
  char_queue text = create( stdin, length( word ) ) ;
  printf( "cucumber: %d times\n", word_count( word, text ) ) ;
  return 0 ;
}
```

Answer to 7.11: ♣ The function `valid` returns the number of valid elements in the array based queue.

```
(* valid :  a queue -> int *)
fun valid (Queue (stream, valid, array)) = valid ;
```

In C, this becomes:

```
int valid( char_queue q ) {
  return q->queue_valid ;
}
```

Answer to 7.12: ♣ Here are the SML and C versions of `word_count` and `match` using arrays.

```
(* list_match : char list -> char list -> bool *)
fun list_match []      t       = true
  | list_match (w::ws) []      = false
  | list_match (w::ws) (t::ts) = w = (t:char) andalso
                                 list_match ws ts ;

(* match : char list -> char array -> bool *)
fun match ws ts = list_match ws (array_to_list ts) ;
```

```
bool list_match( char_list ws, char_list ts ) {
  while( true ) {
    if( ws == NULL ) {
      return true ;
    } else if( ts == NULL ) {
      return false ;
    } else if( head( ws ) == head( ts ) ) {
      ws = tail( ws ) ;
      ts = tail( ts ) ;
    } else {
      return false ;
    }
  }
}

bool match( char_list ws, char ts[], int n ) {
  return list_match( ws, array_to_list( ts, n ) ) ;
}

int word_count( char_list ws, char_queue ts ) {
  int accu = 0 ;
  while( ! is_empty( ts ) ) {
    if( match( ws, fetch( ts ), valid( ts ) ) ) {
      accu++ ;
    }
    advance( ts ) ;
  }
  return accu ;
}
```

Answer to 7.14: We need an auxiliary function to free a list; this function has been defined in Chapter 6.

```
void free_list( char_list xs ) {
  while( xs != NULL ) {
    const char_list last = xs ;
    xs = tail( xs ) ;
    free( last ) ;
  }
}
```

The function `qsort` is going to be modified so that `qsort(x)` will only allocate those cells that are necessary to hold the result list. It will not allocate more cells, nor will it destroy any cells of the input list.

```
char_list qsort( char_list p_xs ) {
  if( p_xs == NULL) {
    return NULL ;
  } else {
    char      p  = head( p_xs ) ;
    char_list xs = tail( p_xs ) ;
    char_list less  = extra_filter( less_eq, &p, xs ) ;
    char_list more  = extra_filter( greater, &p, xs ) ;
    char_list sorted_less = qsort( less ) ;
    char_list sorted_more = qsort( more ) ;
    char_list pivot  = cons( p, NULL ) ;
    char_list sorted = append( sorted_less,
                         append( pivot, sorted_more ) ) ;
    free_list( pivot ) ;
    free_list( less ) ;
    free_list( more ) ;
    free_list( sorted_less ) ;
    return sorted ;
  }
}
```

There are a number of interesting points to note. The function `append` is asymmetrical in that it copies its first argument but reuses the second argument. Therefore, the lists `sorted_more` and the result of the function `append( pivot, sorted_more )` should not be deallocated. Note that one can optimise the code slightly further, as it is not necessary to create the `pivot`-node with a `NULL-tail`, because the next statement appends `sorted_more`. A more efficient solution is where `pivot` is not copied by `append` and deallocated later on:

```
    ...
    char_list pivot  = cons( p, sorted_more ) ;
    char_list sorted = append( sorted_less, pivot ) ;
    free_list( less ) ;
    ...
```

Adding the garbage collection is essential, no system can work with a memory leak. It is also clear that cluttering the code with deallocations is not pretty. Sometimes this can be improved by defining functions in such a way that they *do* destroy their arguments. As an example, we could split the list `p_xs` destructively into `pivot`, `less`, and `more`, and `append` could destructively append its arguments. In that case, `qsort` would destroy its argument, and one can write a quicksort that does not have to allocate or free any cells. In order to create a functional interface to the function, one needs an auxiliary function `quicksort` that first copies its argument before calling the destructive `qsort`.

Answer to 7.15: ♣ Here is an SML function `main` to call `qsort` on the data:

```
(* main : char list *)
val main = qsort (explode "ECFBACG") ;
```

Here is the corresponding C version:

```
int main( void ) {
  char_list sorted = qsort( array_to_list( "ECFBACG", 7 ) ) ;
  list_to_stream( stdout, sorted ) ;
  putchar(  \n  ) ;
  return 0 ;
}
```

Answer to 7.16:

```
void qsort( char data [], int l, int r ) {
  if( l < r) {
    int p = l ;
    char data_p = data[p] ;
    int i = l ;
    int j = r ;
    while( true ) {
      i = up(   data_p, data, i, r ) ;
      j = down( data_p, data, l, j ) ;
      if( i < j ) {
        swap( data, i, j ) ;
        i++ ;
        j-- ;
      } else if( i < p ) {
        swap( data, i, p ) ;
        i++ ;
        break ;
      } else if( p < j ) {
        swap( data, p, j ) ;
        j-- ;
        break ;
      } else {
        break ;
      }
    }
    qsort( data, l, j ) ;
    qsort( data, i, r ) ;
  }
}
```

Answer to 7.19: ♣ Here is a complete C program to bubble sort its arguments supplied through `argc` and `argv`:

```
  #include <stdio.h>
  #include <string.h>

  void swap( char * data[], int i, int j ) {
    char * data_i = data[i] ;
    char * data_j = data[j] ;
    data[i] = data_j ;
    data[j] = data_i ;
  }

  void bubble_sort( char * data [], int l, int r ) {
    int i,k ;
    for( i = l; i < r; i++ ) {
      for( k = i+1; k <= r; k++ ) {
        if( strcmp( data[i], data[k] ) > 0 ) {
          swap( data, i, k ) ;
        }
      }
    }
  }

  int main( int argc, char * argv [] ) {
    int i ;
    bubble_sort( argv, 1, argc-1 ) ;
    printf( "%s", argv[0] ) ;
    for( i = 1; i < argc; i++ ) {
      printf( " %s", argv[i] ) ;
    }
    printf( "\n" ) ;
    return 0 ;
  }
```

Answer to 7.21: Here is a complete C program implementing the Caesar cipher encryption and decryption method for streams.

```
#include <stdio.h>
#include <stdlib.h>

void crypt( FILE * in, FILE * out, int k ) {
  char c ;
  while( (c = getc( in ) ) != EOF ) {
    putc( c + k, out ) ;
  }
}

int main( int argc, char * argv [] ) {
  if( argc != 2 ) {
    printf( "usage: %s [+|-]number\n", argv[0] ) ;
    return 1 ;
  } else {
    char * rest ;
    int k = strtol( argv[1], &rest, 10 ) ;
    if( *rest !=  \0  ) {
      printf( "usage: %s illegal argument %s\n",
              argv[0], argv[1] ) ;
      return 1 ;
    } else {
      crypt( stdin, stdout, k ) ;
      return 0 ;
    }
  }
}
```

Answers to the exercises of Chapter 8

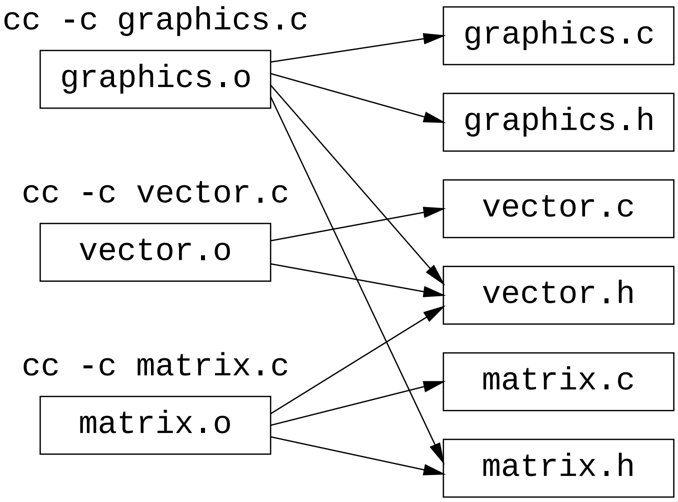
Answer to 8.1:

```
graphics.o: graphics.c graphics.h matrix.h vector.h
        cc -c graphics.c
```

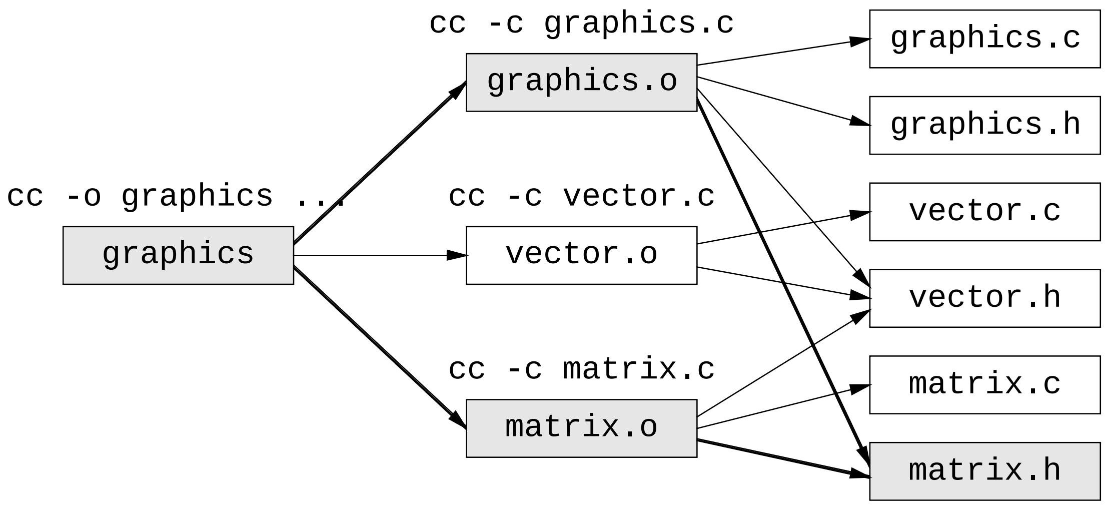
Answer to 8.2: The combined make file is:

```
vector.o: vector.c vector.h
        cc -c vector.c
matrix.o: matrix.c matrix.h vector.h
        cc -c matrix.c
graphics.o: graphics.c graphics.h matrix.h vector.h
        cc -c graphics.c
```

The dependency graph is:



Answer to 8.3: Following the arrows shows that `graphics.o`, `matrix.o`, and `graphics` need to be remade if `matrix.h` is changed.



The objects `graphics.o` and `matrix.o` must be remade before `graphics` is made, but they can be compiled in any order.

Answer to 8.8: The iterative version of `length` which operates on polymorphic lists is:

```
int length( list x_xs ) {
  int accu = 0 ;
  while( x_xs != NULL ) {
    accu++ ;
    x_xs = tail( x_xs ) ;
  }
  return accu ;
}
```

The iterative version of `nth` for polymorphic lists is:

```
void * nth( list x_xs, int n ) {
  while( n != 0 ) {
    x_xs = tail( x_xs ) ;
    n-- ;
  }
  return head( x_xs ) ;
}
```

Answer to 8.9: The polymorphic open list version of `append` with advanced use of pointers is:

```
list append( list xs, list ys, int size ) {
  list accu = ys ;
  list *lastptr = &accu ;
  while( xs != NULL ) {
    list new = cons( head( xs ), ys, size ) ;
    *lastptr = new ;
    lastptr = & new->list_tail ;
    xs = tail( xs ) ;
  }
  return accu ;
}
```

Answer to 8.10: Here is a memoising version of the factorial function:

```
#define MAX 12

int fac( int n ) {
  static int memo[MAX] = {1,0} ;
  if( n < 1 || n > MAX ) {
    abort( ) ;
  } else {
    int r = memo[n-1] ;
    if( r == 0 ) {
      r = n * fac( n-1 ) ;
      memo[n-1] = r ;
    }
    return r ;
  }
}
```

Answer to 8.11: This answer only gives the header file for the polymorphic array ADT:

```
typedef struct Array *Array ;

extern Array arraynew( int l, int u, int size ) ;
extern void  arrayextend( Array a, int l, int u ) ;
extern void  arraystore( Array a, int index, void *value ) ;
extern void* arrayload( Array a, int index ) ;
```

The structure used in the implementation is:

```
struct Array {
  int l, u ;    /* Store current bounds of array */
  int size ;    /* Store size of elements in array */
  void **data ; /* Store pointers to elements */
} ;
```

Answer to 8.13: ♣

(a) Here are the functions `snoc`, `head` and `tail` that operate on a snoc list:

```
snoc_list snoc( snoc_list head, void * tail ) {
  snoc_list l = malloc( sizeof( struct snoc_struct ) ) ;
  if( l == NULL ) {
    printf( "snoc: no space\n" ) ;
    abort( ) ;
  }
  l->snoc_head = head ;
  l->snoc_tail = tail ;
  return l ;
}

snoc_list head( snoc_list l ) {
  return l->snoc_head ;
}

void * tail( snoc_list l ) {
  return l->snoc_tail ;
}
```

(b) The function `sprint` is recursive because it must print the elements of the snoc list in the correct order. Care has been taken to avoid printing a comma at the beginning or at the end of the list.

```
void sprint( void (* print) ( void * ), snoc_list xs ) {
  if( xs != NULL ) {
    snoc_list ys = head( xs ) ;
    if( ys != NULL ) {
      sprint( print, ys ) ;
      printf( "," ) ;
    }
    print( tail( xs ) ) ;
  }
}
```

(c) Here is a main function and an auxiliary function to test the snoc type and associated functions:

```
void print_elem( void * p ) {
  int i = (int) p ;
  printf( "%d", i ) ;
}

int main( void ) {
  snoc_list sl = snoc( snoc( NULL, (void *) 1 ), (void *) 2 ) ;
  sprint( print_elem, sl ) ;
  printf( "\n" ) ;
  return 0 ;
}
```

Answer to 8.14: ♣

(a) Here are the C type definitions that represent an `ntree`:

```
typedef enum {Br, Lf} ntree_tag ;

typedef struct ntree_struct {
  ntree_tag tag ;
  union {
    int       lf ;  /* tag Lf */
    snoc_list br ;  /* tag Br */
  } alt ;
} * ntree ;
```

(b) Here are the functions `nlf` and `nbr`:

```
ntree nlf( int lf ) {
  ntree t ;
  t = malloc( sizeof( struct ntree_struct ) ) ;
  if( t == NULL ) {
    printf( "nlf: no space\n" ) ;
    abort( ) ;
  }
  t->tag = Lf ;
  t->alt.lf = lf ;
  return t ;
}

ntree nbr( snoc_list br ) {
  ntree t ;
  t = malloc( sizeof( struct ntree_struct ) ) ;
  if( t == NULL ) {
    printf( "nbr: no space\n" ) ;
    abort( ) ;
  }
  t->tag = Br ;
  t->alt.br = br ;
  return t ;
}
```

(c) Here is the `nprint` function and its auxiliary function to print an element of the snoc list.

```
void print_elem( void * e ) {
  ntree t = e ;
  nprint( t ) ;
}

void nprint( ntree t ) {
  switch( t->tag ) {
```

```
  case Lf :
    printf( "%d", t->alt.lf ) ;
    break ;
  case Br :
    printf( "(" ) ;
    sprint( print_elem, t->alt.br ) ;
    printf( ")" ) ;
    break ;
  }
}
```

(d) Here is a main program to test the `ntree` data types and associated functions:

```
int main( void ) {
  snoc_list l2 = snoc( snoc( snoc( NULL,
                                   nlf( 2 )
                                 ),
                             nlf( 3 )
                           ),
                       nlf( 4 )
                     ) ;
  snoc_list l1 = snoc( snoc( NULL,
                             nlf( 1 )
                           ),
                       nbr( l2 )
                     );
  ntree t1 = nbr( l1 ) ;
  ntree t2 = nbr( NULL ) ;
  nprint( t1 ) ;
  printf( "\n" ) ;
  nprint( t2 ) ;
  printf( "\n" ) ;
  return 0 ;
}
```

Answer to 8.15: ♣

(a) Here is the C interface `snoclist.h` for the snoc list module:

```
#ifndef SNOC_LIST_H
#define SNOC_LIST_H
typedef struct snoc_struct * snoc_list ;

extern snoc_list snoc( snoc_list head, void * tail ) ;

extern snoc_list head( snoc_list l ) ;
extern void * tail( snoc_list l ) ;
extern void sprint( void (* print) ( void * ), snoc_list l ) ;
#endif /* SNOC_LIST_H */
```

Here is the corresponding C implementation `snoclist.c` for the snoc list module:

```
#include <stdlib.h>
#include <stdio.h>
#include "snoclist.h"

struct snoc_struct {
  void                * snoc_tail ;
  struct snoc_struct  * snoc_head ;
} ;

snoc_list snoc( snoc_list head, void * tail ) {
  snoc_list l = malloc( sizeof( struct snoc_struct ) ) ;
  if( l == NULL ) {
    printf( "snoc: no space\n" ) ;
    abort( ) ;
  }
  l->snoc_head = head ;
  l->snoc_tail = tail ;
  return l ;
}

snoc_list head( snoc_list l ) {
  return l->snoc_head ;
}

void * tail( snoc_list l ) {
  return l->snoc_tail ;
}

void sprint( void (* print) ( void * ), snoc_list xs ) {
  if( xs != NULL ) {
    snoc_list ys = head( xs ) ;
    if( ys != NULL ) {
      sprint( print, ys ) ;
      printf( "," ) ;
    }
    print( tail( xs ) ) ;
  }
}
```

(b) Here is the C interface `ntree.h` for the `ntree` data type and its functions.

```
#ifndef NTREE_H
#define NTREE_H
typedef struct ntree_struct * ntree ;
```

```
extern ntree nlf( int lf ) ;
extern ntree nbr( snoc_list br ) ;

extern void nprint( ntree t ) ;
#endif /* NTREE_H */
```

Here is the implementation of the `ntree` module:

```
#include <stdlib.h>
#include <stdio.h>
#include "snoclist.h"
#include "ntree.h"

typedef enum {Br, Lf} ntree_tag ;

struct ntree_struct {
  ntree_tag tag ;
  union {
    int       lf ;  /* tag Lf */
    snoc_list br ;  /* tag Br */
  } alt ;
} ;

ntree nlf( int lf ) {
  ntree t ;
  t = malloc( sizeof( struct ntree_struct ) ) ;
  if( t == NULL ) {
    printf( "nlf: no space\n" ) ;
    abort( ) ;
  }
  t->tag = Lf ;
  t->alt.lf = lf ;
  return t ;
}

ntree nbr( snoc_list br ) {
  ntree t ;
  t = malloc( sizeof( struct ntree_struct ) ) ;
  if( t == NULL ) {
    printf( "nbr: no space\n" ) ;
    abort( ) ;
  }
  t->tag = Br ;
  t->alt.br = br ;
  return t ;
}

void nprint( ntree t ) ;
void print_elem( void * e ) {
  ntree t = e ;
  nprint( t ) ;
}

void nprint( ntree t ) {
  switch( t->tag ) {
  case Lf :
    printf( "%d", t->alt.lf ) ;
    break ;
  case Br :
    printf( "(" ) ;
    sprint( print_elem, t->alt.br ) ;
    printf( ")" ) ;
    break ;
  }
}
```

(c) Here is a main program to test the modules:

```
#include <stdio.h>
#include "snoclist.h"
#include "ntree.h"

int main( void ) {
  snoc_list l2 = snoc( snoc( snoc( NULL,
                                   nlf( 2 )
                                 ),
                             nlf( 3 )
                           ),
                       nlf( 4 )
                     ) ;
  snoc_list l1 = snoc( snoc( NULL,
                             nlf( 1 )
                           ),
                       nbr( l2 )
                     );
  ntree t1 = nbr( l1 ) ;
  ntree t2 = nbr( NULL ) ;
  nprint( t1 ) ;
  printf( "\n" ) ;
  nprint( t2 ) ;
  printf( "\n" ) ;
  return 0 ;
}
```

Answers to the exercises of Chapter 9

Answer to 9.1: The C preprocessor performs a textual substitution, so any occurrence of the identifiers WIDTH and HEIGHT would be replaced. This results in troubles for the functions window_to_complex and draw_Mandelbrot, as WIDTH and HEIGHT appear in their argument lists. The header of the function window_to_complex is:

```
complex window_to_complex( int X, int Y, int WIDTH,
                           int HEIGHT, double x, double y,
                           double width, double height )
```

After substitution of WIDTH and HEIGHT it would read:

```
complex window_to_complex( int X, int Y, int 100,
                           int 100, double x, double y,
                           double width, double height )
```

This is not legal C. The same substitution would take place in the argument list of draw_Mandelbrot.

Answer to 9.3: The following two elements are needed:

```
  void  (*draw_circle)( void *g, int x0, int y0, int r ) ;
  void  (*draw_ellipse)( void *g, int x0, int y0, int r, int dx ) ;
```

Answer to 9.4: The SML data structure uses a type variable a. This type is returned by the Open function and used by subsequent functions. The C data structure uses void * as replacement for polymorphic typing. The C compiler cannot check whether the appropriate data structure is passed from the open-function to the draw-functions (see Section 4.5).

Answer to 9.6: The PostScript driver for the device independent graphics is:

```
#include "PSdriver.h"
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  FILE *file ;
} PSInfo ;

void *PS_open( void *what ) {
  PSInfo *i = malloc( sizeof( PSInfo ) ) ;
  char *filename = what ;
  i->file = fopen( filename, "w" ) ;
  if( i->file == NULL ) {
    return NULL ;
  }
  fprintf( i->file, "%%!PS\n" ) ;
  return (void *) i ;
}

void PS_draw_line(void *g, int x0, int y0, int x1, int y1) {
  PSInfo *i = g ;
  fprintf( i->file, "newpath\n" ) ;
  fprintf( i->file, "%d %d moveto\n", x0, y0 ) ;
  fprintf( i->file, "%d %d lineto\n", x1, y1 ) ;
  fprintf( i->file, "closepath\n" ) ;
  fprintf( i->file, "stroke\n" ) ;
}

void PS_draw_box(void *g, int x0, int y0, int x1, int y1) {
  PSInfo *i = g ;
  fprintf( i->file, "newpath\n" ) ;
  fprintf( i->file, "%d %d moveto\n", x0, y0 ) ;
  fprintf( i->file, "%d %d lineto\n", x0, y1 ) ;
  fprintf( i->file, "%d %d lineto\n", x1, y1 ) ;
  fprintf( i->file, "%d %d lineto\n", x1, y0 ) ;
  fprintf( i->file, "%d %d lineto\n", x0, y0 ) ;
  fprintf( i->file, "closepath\n" ) ;
  fprintf( i->file, "stroke\n" ) ;
}

void PS_close( void *g ) {
  PSInfo *i = g ;
  fprintf( i->file, "%%%%Eof\n" ) ;
  fclose( i->file ) ;
  free( i ) ;
}

graphics_driver PSDriver = {
  PS_open,
  PS_draw_line,
  PS_draw_box,
  /*C other functions of the driver*/
  PS_close
} ;
```

Appendix B

A brief review of SML

In this appendix we will present a brief overview of the salient features of SML for the reader who is familiar with another functional programming language. We cover just enough material to make the reader feel comfortable when reading our book. We should point out that SML has more to offer than we use in this book; the interested reader might wish to consult one of the many textbooks available on programming in SML [15, 9, 16].

B.1 About the four main functional languages

The most important functional languages to date are Lisp, SML, Miranda, and Haskell. There are many small differences between these four languages and only a few major differences.

The main difference between Lisp and the other three is that Lisp is dynamically typed, whereas the other three languages are statically typed. In a statically typed language the compiler will reject expressions that are incorrectly typed; in a dynamically typed languages such errors are detected at run time.

The compilers for SML, Miranda, and Haskell will automatically infer the types of all functions. In addition, Miranda and Haskell allow the programmer to explicitly declare the type of a function. The compiler will check that the explicit type declaration is consistent with type information as has been inferred. Explicit type declarations are a useful form of documentation, which help the reader to understand the purpose of a function. The fact that this form of documentation can be checked for consistency makes it also a reliable form of documentation. SML unfortunately does not permit the explicit declaration of function types. Therefore we have resorted to giving the type of each function in this book as a comment. (Comments in SML are enclosed in the symbols `(*` and `*)`).

The main difference between SML and Miranda on the one hand and Haskell on the other hand is that the type system of Haskell provides proper support for overloaded functions by means of the type class system. A function (or more appropriately an operator) such as $+$ is overloaded if it can be applied to arguments (or operands) of different types. For example, in the expression $1+2$, the operands are integers and, in the expression $1.0+3.14$, the operands are reals.

The main difference between Lisp and SML on the one hand and Miranda and Haskell on the other hand is that Lisp and SML are eager languages whereas Miranda and Haskell are lazy languages. In an eager language, the argument to a function is always evaluated before the function is called. In a lazy language, the evaluation of an argument to a function may be postponed until the evaluation of the argument is necessary. If the function does not actually use its argument, it will never be evaluated, thus saving some execution time. The functions that we use in this book work in an eager language. It would thus be possible to interpret all our SML functions in Haskell or Miranda.

The module systems of the four programming languages are all rather different. SML has the most sophisticated module system. The module systems of Haskell and Miranda are simpler but effective. Many different Lisp systems exists, with a wide variation of module systems, ranging from the primitive to the sophisticated.

Below we discuss the basic elements of a functional program, and describe the particular features offered by SML. In its basic form, a module from a functional program consists of a number of data type and function definitions, and an expression to be evaluated.

B.2 Functions, pattern matching, and integers

All functional languages offer the possibility to define a function by giving its name, a list of arguments and the function body. We give as an example the definition of a function to calculate a binomial coefficient. The function has two integer arguments and computes an integer result. We use the following equation to compute the binomial coefficient, assuming that $0 \leq m \leq n$:

$$\binom{n}{m} = \begin{cases} 1, & \text{if } m = 0 \vee m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1}, & \text{otherwise} \end{cases}$$

Here is the SML function `over` which uses this equation. The type of the function is given as a comment. It states that `over` has two integer arguments and that its function result is also an integer.

```
(* over : int -> int -> int *)
fun over n 0 = 1
  | over n m = if n = m
                  then 1
                  else over (n-1) m + over (n-1) (m-1) ;
```

A function definition in SML is introduced by the keyword `fun` and terminated by a semicolon `;`. Layout is not significant in SML; in particular, there is no off side rule as in Miranda and Haskell. The function name is `over`. The function name is then followed by the names of the formal arguments. The function `over` has two alternative definitions, separated by the vertical bar `|`. The first clause applies when the second argument has the value `0`. The second clause applies when the second argument, m, is not equal to 0. The two defining clauses of the function are distinguished by pattern matching on the argument(s). Clauses are tried top down, and arguments are matched left to right.

The function result of the first clause of over is 1; that of the second clause is determined by the conditional expression if ... then ... else .... If the values of n and m are equal, the function result is also 1. If n and m are unequal, two recursive calls are made and the results are added.

In most functional languages, parentheses serve to build expressions out of groups of symbols and not to delineate function arguments. This explains why the recursive call to the function over looks like this:

```
over (n-1) (m-1)
```

One might have expected it to look like this:

```
over(n-1,m-1)
```

This latter notation would be more consistent with common mathematical use of parentheses to delineate function arguments. In SML, but also in Haskell and Miranda, it is possible to enclose functional arguments in parentheses and to separate the arguments by commas. With this notation, we can define a new function over to look like this (an identifier may contain an apostrophe as a legal character):

```
(* over  : int * int -> int *)
fun over (n,0) = 1
  | over (n,m) = if n = m
                    then 1
                    else over (n-1,m) + over (n-1,m-1) ;
```

The two arguments n and m are now treated as a tuple consisting of two integers. This is indicated in the type of the function, which indicates that it expects one argument of type (int * int). This notation is called the uncurried notation. In the curried notation a function takes it arguments one by one. The curried notation is commonly used in lazy languages such as Haskell or Miranda programs. It is easy to mix the two notations inadvertently, but the compilers will discover such errors because of the ensuing type mismatches.

Haskell and Miranda do not distinguish between functions with arguments and functions without arguments. However, SML does distinguish between the two. A function without arguments is a value, which may be given a name using the val declaration. The following defines two values, over_4_3 and over_4_2. Both are of type int:

```
(* over_4_3,over_4_2 : int *)
val over_4_3 = over 4 3 ;
val over_4_2 = over 4 2 ;
```

The results printed by the SML system are 4 and 6 respectively.

B.3 Local function and value definitions

SML permits the definition of local functions and values within an expression. Local definitions are often useful to avoid the recomputation of a value that is used several times. In the function `over` defined earlier, the expression `n-1` appears twice. This value can be computed just once by giving it a name in a local `let` definition:

```
(* over : int -> int -> int *)
fun over n 0 = 1
  | over n m = if n = m
                  then 1
                  else let
                          val k = n-1
                       in
                          over k m + over k (m-1)
                       end ;
```

The construct `let` … `in` … `end` represents a single expression with one or more local definitions. After the keyword `let`, any number of function and value definitions may appear, each introduced by the relevant keyword `fun` or `val`. The `let` expressions of SML are more general than the `where` clauses of Miranda, because `let` expressions may be arbitrarily nested. Miranda's `where` clauses apply to function definitions and can only be nested if the function definitions are nested. Haskell provides both facilities.

B.4 Reals, booleans, strings, and lists

The basic data types of SML are integers, reals, booleans, strings, and lists. In the previous section we have only used integers; here we will define a function that works on other data types to see how they could be used. The function `locate` below is a polymorphic function of two arguments. The second argument is a list of values of some polymorphic type ` a`, in which the function `locate` will be searching for an occurrence of the first argument. In SML type variables are written as ` a`, ` b`, and so on. In Haskell we would write `a`, `b` and in Miranda one would use `*`, `**`, and so on.

The first argument of `locate` is of the same type ` a` as the elements of the list. If the element is found in the list, the boolean value `true` is returned, `false` otherwise. The term 'polymorphic' stems from the fact that a polymorphic function has really many forms; it adapts itself to the form of its actual argument.

```
(* locate :  a ->  a list -> bool *)
fun locate a []      = false
  | locate a (x::xs) = (a = x) orelse locate a xs ;
```

The function `locate` uses pattern matching on the second argument. If this represents the empty list `[]`, the function result is `false`. Otherwise, the list is deemed to be non empty, with a head element `x` and tail `xs`. The double colon `::` is the pattern symbol for lists. In Haskell and Miranda : is the pattern symbol for lists and :: introduces a type.

The operator orelse in SML is the logical disjunction. It evaluates its left operand first. If this yields true, then it does not evaluate the right operand. If the leftmost operand evaluates to false, then the rightmost operand of orelse is evaluated. The orelse operator is said to have *short-circuit* semantics. The orelse operator has two companions: the andalso operator for the logical conjunction and the conditional construct if … then … else. All three have short-circuit semantics. These are the only exceptions to the rule that SML functions and operators always evaluate their arguments first.

The function locate is polymorphic in the type of the element to be looked up in the list. Therefore the following are all valid uses of locate:

```
(* locate_nil,locate_bool,locate_int : bool *)
(* locate_char,locate_string,locate_real : bool *)
val locate_nil    = locate 2 [] ;
val locate_bool   = locate true [false,false] ;
val locate_int    = locate 2 [1,2,3] ;
val locate_char   = locate "x" ["a","b","c","d"] ;
val locate_string = locate "foo" ["foo","bar"] ;
val locate_real   = locate 3.14 [0.0, 6.02E23] ;
```

The notation [1,2,3] is shorthand for 1::(2::(3::[])), where [] is the empty list and :: is the concatenation of a new element to the head of a list.

SML does not offer characters as a basic type. Instead, it provides character strings enclosed in double quotes ". By convention, a string with a single character is used where one would use a character in Haskell or Miranda. A further property of SML is that a string is not a list of characters but a separate data type. The functions explode and implode convert between an SML string and a list of single element strings. The two equations below are both true:

```
(* true_explode,true_implode : bool *)
val true_explode = (explode "foo" = ["f","o","o"]) ;
val true_implode = (implode ["b","a","r"] = "bar") ;
```

B.5 Type synonyms and algebraic data types

User defined data types are introduced either by type synonyms or by algebraic data type declarations. A type synonym gives a name to an existing type. For example, a pair of two integers has type int * int. Instead of spelling this out each time we use this type, we can give it a name, say int_pair, and use the name instead of int * int. This is how we would define the type synonym in SML:

```
type int_pair = int * int ;
```

The keyword type indicates that a type synonym is going to be defined. This is the same as in Haskell. In Miranda the symbol == introduces a type synonym.

New types can be created using a algebraic data type declaration. Here is how a binary tree would be defined with integers at the leaf nodes:

```
datatype int_tree = Int_Branch of int_tree * int_tree
                  | Int_Leaf of int ;
```

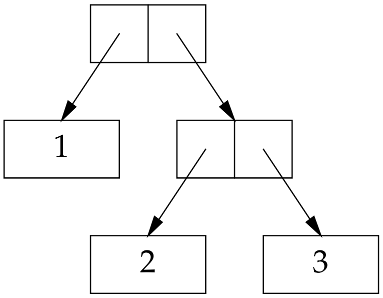
The keyword `datatype` announces the declaration of an algebraic data type. Here the identifier `Int_Branch` is a data constructor, which can be viewed as a function. This function should be applied to a tuple consisting of a left subtree and a right subtree. Both subtrees are values of type `int_tree`. The identifier `Int_Leaf` is again a data constructor, this time taking an integer value as an argument. SML does not permit curried constructors; Haskell and Miranda allow both curried and uncurried constructors but the former are more common. Here is the usual curried Miranda notation:

```
num_tree ::= Num_Branch num_tree num_tree |
             Num_Leaf num ;
```

Let us now use the `int_tree` data type definition to create a sample trees in SML:

```
(* sample_int_tree : int_tree *)
val sample_int_tree
    = Int_Branch(Int_Leaf 1,
                 Int_Branch(Int_Leaf 2,Int_Leaf 3)) ;
```

A graphical representation of this tree would be as follows:



The function `walk_add`, as given below, traverses a tree, adding the numbers stored in the leaves. The function definition uses pattern matching on the constructors of the data structure:

```
(* walk_add : int_tree -> int *)
fun walk_add (Int_Branch(left,right))
    = walk_add left + walk_add right
  | walk_add (Int_Leaf data)
    = data ;
```

The type of `walk_add` states that the function has one argument with values of type `int_tree` and that the function result is of the type `int`. The definition of `walk_add` has two clauses. The first clause applies when an interior node is encountered. As interior nodes do not contain data, the function result returned by this clause is the sum of the results from the left and right branch. The function result returned by the second clause is the data stored in a leaf node.

The value add_int_main below represents walk_add applied to our sample tree of integers:

```
(* add_int_main : int *)
val add_int_main = walk_add sample_int_tree ;
```

The value computed by add_int_main is 6.

B.6 Higher order functions

The walk_add function from the previous section is a combination of two things: it embodies a tree traversal algorithm and it encodes a particular operation (addition) over the tree. These two issues can be separated to make the code usable in a wider context. This separation requires the introduction of a polymorphic data type tree and a pure tree walk function poly_walk. Here is the polymorphic tree data type:

```
datatype  a tree = Branch of  a tree *  a tree
                 | Leaf of  a ;
```

The tree data type has a type parameter, indicated by the type variable a, for which any type may be substituted. Our sample tree can be encoded using the tree data type as follows:

```
(* sample_poly_tree : int tree *)
val sample_poly_tree
    = Branch(Leaf 1,
             Branch(Leaf 2,Leaf 3)) ;
```

The type of sample_poly_tree is int tree, which indicates that we have instantiated the polymorphic tree data type to a concrete tree with data of type int at the leaves.

Not only the data type tree but also the poly_walk function carries a parameter. This parameter (called comb below) represents the binary function applied when results of the left and right branch are combined.

```
(* poly_walk : ( a-> a-> a) ->  a tree ->  a *)
fun poly_walk comb (Branch(left,right))
    = comb (poly_walk comb left) (poly_walk comb right)
  | poly_walk comb (Leaf data)
    = data ;
```

The type of poly_walk indicates that the comb function has type a-> a-> a. It should produce a result of the same type as that of its two arguments. A value of that same type should be stored in the tree, and a value of this type will also be produced as the final result of poly_walk.

The value poly_add_int_main below applies the general poly_walk function to the concrete binary tree sample_poly_tree, which has integers at its leaf nodes. Here add is the curried version of the addition operator:

```
(* poly_add_int_main : int *)
val poly_add_int_main
```

```
      = let
           fun add x y = x+y
        in
           poly_walk add sample_poly_tree
        end ;
```

The following identity relates the two tree walk functions that have been defined thus far:

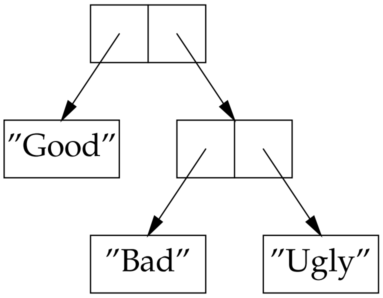
`poly_walk add a_tree = walk_add a_tree`

It is possible to prove that this equality holds for all finite trees by induction on the structure of the argument `a_tree`.

Here is another tree built using the `tree` data type. This time, we have strings at the leaves:

```
  (* string_poly_tree : string tree *)
  val string_poly_tree
      = Branch(Leaf "Good",
               Branch(Leaf "Bad",Leaf "Ugly")) ;
```

A graphical representation of this tree would be:



A function suitable to traverse this tree would be one that concatenates the strings found at the leaves. To make the output look pretty it also inserts a space between the words found in the tree:

```
  (* film : string *)
  val film
      = let
           fun concat x y = x ^ " " ^ y
        in
           poly_walk concat string_poly_tree
        end ;
```

The string value of `film` is thus `"Good Bad Ugly"`. Polymorphism has helped to produce two different functions with code reuse.

B.7 Modules

A module system serves to gather related type, data type, function, and value declarations together so that the collection of these items can be stored, used, and manipulated as a unit. The module system of SML is one of the most advanced available to date in any programming language. We will discuss here the basics of the module system. The most striking feature of the module system is its systematic design. An SML structure is a collection of types, data types, functions, and values. A signature basically gives just the types of these items in a structure. The relation between structures and signatures is roughly the same as the relation between types on the one hand and functions and values on the other. Put differently, signatures (types) are an abstraction of structures (values and functions).

Consider the following structure as an example. It collects an extended polymorphic tree data type `tree`, an empty tree `empty`, and an extended tree walk function `walk` into a structure called `Tree`. (The extensions were made to create a more interesting example, not because of limitations of the module mechanism.)

```
structure Tree = struct

   datatype  a tree = Branch of  a tree *  a tree
                    | Leaf of  a
                    | Empty ;

   (* empty :  a tree *)
   val empty = Empty ;

   (* walk : ( a-> a-> a) ->  a ->  a tree ->  a *)
   fun walk comb default (Branch(left,right))
       = comb (walk comb default left)
              (walk comb default right)
     | walk comb default (Leaf data)
       = data
     | walk comb default (Empty)
       = default

end ;
```

The keyword `structure` announces that the declaration of a structure follows. This is similar to the use of the keywords `fun` and `val`. The keyword `struct` is paired with the keyword `end`. These two keywords delineate the declaration of the three components of the structure. The declarations of the components of the structure (the data type `tree`, the value `empty`, and the function `walk`) are created according to the normal rules for declaring such items.

With the structure declaration in place, we can create a sample tree by qualifying each identifier from the structure by the name of the structure. This time, we will create a tree of reals as an example:

```
(* sample_real_tree : real Tree.tree *)
val sample_real_tree
    = Tree.Branch(Tree.Leaf 1.0,
                  Tree.Branch(Tree.Leaf 2.0,Tree.Leaf 3.0));
```

It is necessary to indicate from which structure a particular component emerges, because it is possible to use the same component names in different structures. As a convenient shorthand, we can *open* the structure `Tree` and use its component names unqualified as follows:

```
(* sample_real_tree : real Tree.tree *)
val sample_real_tree
    = let
         open Tree
      in
         Branch(Leaf 1.0,
                Branch(Leaf 2.0,Leaf 3.0))
      end ;
```

Traversing the nodes of the sample tree and adding the data values is achieved by the following code:

```
(* add_real_main : real *)
val add_real_main
    = let
         fun add x y = x+y
      in
         Tree.walk add 0.0 sample_real_tree
      end ;
```

The `walk` function is the only component used from the structure.

The *signature* of a structure is the interface to the structure. Here is the signature of the `Tree` structure:

```
signature TREE = sig
   datatype  a tree = Branch of  a tree *  a tree
                    | Leaf of  a
                    | Empty ;
   val empty :  a tree ;
   val walk  : ( a-> a-> a) ->  a ->  a tree ->  a
end ;
```

As with the structure, function, and value declarations, a keyword, `signature`, indicates that a signature declaration follows. The contents of the signature declaration are enclosed between the keywords `sig` and `end`.

The signature `TREE` shows that the structure `Tree` has three components. The first is a data type named `tree`. The second component is a value `empty`, and the third is a function that takes three arguments (a function, a value, and a tree). All components are polymorphic in a type parameter `a`.

We could now create a new structure, say `My_Tree`, which explicitly declares the structure to have the signature `TREE` given above:

```
structure My_Tree : TREE = Tree ;
```

The signature of the structure `My_Tree` is the same as that inferred for the structure `Tree`. Thus nothing is to be gained by introducing this new structure. A more interesting use of signatures is to *hide* some of the components of a structure. We could define a new structure, say `Non_Empty_Tree`, which does not provide the value empty. This effectively restricts the use that can be made of components of a structure. Such restrictions are useful to prevent auxiliary types, data types, functions, and values from being used outside the structure.

```
signature NON_EMPTY_TREE = sig
   datatype  a tree = Branch of  a tree *  a tree
                    | Leaf of  a
                    | Empty ;
   val walk : ( a-> a-> a) ->  a ->  a tree ->  a
end ;

structure Non_Empty_Tree : NON_EMPTY_TREE = Tree ;
```

It is even possible to parametrise structures using so-called functors. These more advanced features are not used in the book, so we do not discuss them here.

B.8 Libraries

The SML language provides a set of predefined operators and functions. Furthermore, different implementations of SML may each offer an extensive set of libraries. We have made as little use as possible of the wealth of library functions that are available to the SML programmer. Firstly, having to know about a minimal core of SML and its libraries makes it easier to concentrate on learning C. Secondly, by using only a few library functions the book is only loosely tied to a particular implementation of SML.

We use a small number of predefined operators and functions and only four functions from the SML/NJ array library. In addition, we have defined a number of functions of our own which are similar to the SML library functions of most implementations.

Here are the predefined operators and their types as they are being used throughout the book:

operator	type
+, -, *, div, mod	int * int -> int
+, -, *, /	real * real -> real
<, <=, <>, =, >=, >	int * int -> bool
<, <=, <>, =, >=, >	real * real -> bool
ˆ	string * string -> string
size	string -> int
ord	string -> int
chr	int -> string
::	a * a list -> a list
@	a list * a list -> a list

The list processing functions below are similar to those found in Haskell and Miranda. The SML versions that we use are not from a standard library. Instead, they have been defined in the text and are used in many places. Several of these functions are used in a specific monomorphic context. We only give the polymorphic forms here. The name and type of each function is accompanied by the number of the page where the function is defined and described. The list is in alphabetical order.

function	type	page number
`(--)`	`int * int -> int list`	80
`append`	` a list ->  a list ->  a list`	186
`filter`	`( a -> bool) ->  a list ->  a list`	92
`foldl`	`( a ->  b ->  a) ->  a ->  b list ->  a`	80
`foldr`	`( b ->  a ->  a) ->  a ->  b list ->  a`	86
`head`	` a list ->  a`	183
`length`	` a list -> int`	185
`map`	`( a ->  b) ->  a list ->  b list`	89
`nth`	` a list -> int ->  a`	186
`prod`	`int list -> int`	81
`sum`	`int list -> int`	89
`tail`	` a list ->  a list`	183

Here are the four array functions that we use from the SML/NJ array module. They are similar to the array processing functions from Haskell.

function	type	page number
`array`	`int *  a ->  a array`	136
`length`	` a array -> int`	137
`sub`	` a array * int ->  a`	137
`tabulate`	`int * (int-> a) ->  a array`	137

We add to this repertoire of basic array functions three further generally useful functions:

function	type	page number
`concatenate`	` a array *  a array ->  a array`	138
`slice`	` a array * int * int ->  a array`	138
`upd`	` a array * int *  a ->  a array`	137

This concludes the presentation of the relatively small core of SML that we use in the book to study programming in C.

Appendix C

Standard Libraries

C offers a rich variety of library functions. Some of these functions are used so often that a short reference is indispensable to the reader of this book. This chapter provides such a reference. For the complete set of standard libraries one has to consult the C reference manual [7].

Besides the standard libraries many *system calls* are usually directly available to the C programmer. As an example, functions for accessing files under UNIX, Windows, or Macintosh and usually also functions to manage processes or network connections are readily available.

In addition to these, there are many other libraries available. We have seen a small part of the X-window library in Chapter 9. Other libraries that exist are for example cryptographic libraries [11], and numerical libraries [10]. For all these, we refer the reader to the appropriate documentation.

The standard library consists of a collection of modules. Each of these modules requires a different interface file to be loaded. Most computer systems will not require you to specify that you wish to link any of the libraries, with the exception of the mathematics library under UNIX. Below, five modules are explained in detail: I/O, strings, character types, mathematics, and utilities. The final section summarises the purpose of the modules that we have not described.

C.1 Standard I/O

The standard I/O library provides functions that allow the programmer to perform input and output operations. Input and output can be performed on files, which have a type `FILE *`. There are three standard file descriptors: `stdin`, which is the standard input of the programming, normally from the keyboard; `stdout`, the standard output of the program, normally directed to the screen; and `stderr`, the file to write error messages, normally directed to the screen.

To use any of the I/O facilities, the file `stdio.h` has to be included:

```
#include <stdio.h>
```

The most important data and functions of this library are listed below. A complete reference is given in the C reference manual [7]:

`FILE` Is a type that stores the information related to a file. Normally only references to this types are passed, which are thus of type `FILE *`. An entity of type `FILE *` is referred to as a file pointer or a stream.

`FILE *stdin` Is a global identifier referring to the standard input stream.

`FILE *stdout` Is a global identifier referring to the standard output stream.

`FILE *stderr` Is a global identifier referring to the standard error stream.

`void putchar( char c )` Puts a character on the standard output stream.

`void printf( char *format, ... )` Prints its arguments on the standard output stream. The first argument is the *format string*, which specifies how the other parameters are printed. The format string is copied onto the output, with the exception of `%` specifiers. Whenever a `%` is encountered, one of the arguments is formatted and printed. For example, a `%d` indicates an integer argument, `%f` a floating point number, `%s` a string, `%c` a character, and `%p` a pointer.

`int getchar( void )` Reads one character from the standard input. It returns this character as an integer, or, if there are no more characters, the special value `EOF` (End Of File). Note that `EOF` is an integer which cannot be represented as a character.

`int scanf( char *format, ... )` Reads values from the standard input stream. The first argument is a string that specifies what types of values to read. The format of the string is similar to that of `printf`: `%d` specifies reading an integer, `%f` a floating point number, and so on. The subsequent parameters must be *pointers* to variables with the appropriate type: `int` for `%d`, `char []` for `%s`, and so on. Types are not checked; therefore, accidentally forgetting an `&` may have disastrous results. It is particularly important to notice the difference between the formats `%f` and `%lf`. The first one expects a pointer to a `float`, the second expects a pointer to a `double`. You will probably need the latter one.

The function `scanf` will read through the input stream, matching the input with the formats specified. When a format is successfully matched, the resulting value is stored via the associated pointer. If a match fails, then `scanf` will give up and return, leaving the input positioned at the first unrecognised character. The return value of `scanf` equals the number of items that were matched and stored. When the input stream is empty, `EOF` will be returned.

`FILE *fopen( char *filename, char *mode )` Creates a file descriptor that is associated with a file of your file system. The first argument specifies the filename, the second the mode. Two frequently used modes are `"r"` (opens the file for reading, you can use it with functions like `scanf`) and `"w"` (opens the file for writing). If the file cannot be opened, a `NULL` pointer is returned.

`void fprintf( FILE *out, char *format, ... )` Is like `printf`, but the first argument specifies on which file to print. The file can be one of `stdout`, `stderr`, or any file opened with `fopen`.

`void sprintf( char *out, char *format, ... )` Is like `printf`, but the first argument specifies an array of characters where the output is to be stored. The array must be large enough to hold the output, no checking is performed.

`void putc( char c, FILE *out )` Is like `putchar`, but on a specific file.

`int fscanf( FILE *in, char *format, ... )` Is like `scanf`, but scans from a specific file.

`int sscanf( char *in, char *format, ... )` Is like `scanf`, but scans from a string.

`int getc( FILE *in )` Is like `getchar`, but from a specific file.

An important note: the functions `putchar`, `getchar`, `getc`, and `putc` are usually implemented with macros. The macro call semantics (Section 8.1.5) can cause strange results when the arguments of these macros have a side effect. Unexpected results can be avoided by using `fputc` and `fgetc` instead.

C.2 Strings

A string is represented as an array of characters (see also Section 5.5). To manipulate these arrays of characters, a string library is provided. To use this library, the file `string.h` must be included:

```
 #include <string.h>
```

The most important functions of this library are:

`int strlen( char *string )` Returns the length of a string.

`char *strncpy( char *out, char *in, int n )` Copies a string. The third argument limits the number of characters that will be copied. The first argument is the destination array, the second the source array.

`char *strcpy( char *out, char *in )` Is like `strncpy`, but has no safeguard against copying too many characters.

`char *strdup( char *string )` Allocates heap space using `malloc` to hold a copy of the string and copies the string into it. If no space is available, a NULL pointer is returned. When the string is no longer needed it should be destroyed using `free`.

`char *strncat( char *out, char *in, int n )` Appends a string to an existing string. The third argument limits the number of characters that will be appended. The first argument should contain the prefix of the string and will contain the concatenated string when the function returns.

`char *strcat( char *out, char *in )` Is like `strncat`, but has no safeguard against copying too many characters.

`int strcmp( char *s, char *t )` Performs a relational operation on two strings. The functions returns a negative number (if `s < t`), zero (if `s == t`), or a positive number (if `s > t`).

`int strncmp( char *s, char *t, int n )` Is like `strcmp` but it compares at most `n` characters. If the strings are equal up to the `n`-th character, `0` is returned.

`char *strchr( char *s, char c )` Finds the first occurrence of the character `c` in the string pointed to by `s`. A pointer to his first occurrence is returned. If `c` does not occur in `s` the NULL pointer is returned.

`char *strstr( char *s, char *t )` Finds the first occurrence of the string `t` in the string pointed to by `s`. A pointer to his first occurrence is returned. If `t` does not occur in `s` the NULL pointer is returned.

Apart from these operations on `\0` terminated character strings, there is a series of functions that operate on blocks of memory. These functions treat the NULL-character as any other character. The length of the block of memory must be passed to each of these functions.

`void *memcpy( void *in, void *out, int n )` Is like `strncpy`.

`void *memmove( void *in, void *out, int n )` Is like `memcpy`, but also works if `in` and `out` are overlapping parts of the same array (aliases).

`void *memchr( void *s, char c, int n )` Is like `strchr`.

`int memcmp( void *s, void *t, int n )` Is like `strncmp`.

`void *memset( void *in, int c, int n )` Fills the first `n` bytes of `in` with the value `c`.

C.3 Character classes

The representation of characters may differ from one machine to another. To write portable programs, the character class library provides predicates that yield true if a character belongs to a certain class. The predicates can be used after including the file `ctype.h`.

```
#include <ctype.h>
```

The functions available in this library are:

`bool isdigit(char c)` Tests if `c` is a digit.

`bool isalpha(char c)` Tests if `c` is a letter.

`bool isupper(char c)` Tests if `c` is an uppercase letter.

`bool islower(char c)` Tests if `c` is a lowercase letter.

`bool isalnum(char c)` Tests if `c` is a letter or a digit.

`bool isxdigit(char c)` Tests if `c` is a hexadecimal digit.

`bool isspace(char c)` Tests if `c` is a white space.

`bool isprint(char c)` Tests if `c` is a printable character.

`bool isgraph(char c)` Tests if `c` is a printable character but not a space.

`bool ispunct(char c)` Tests if `c` is a printable character but not a space, letter or digit.

`bool iscntrl(char c)` Tests if `c` is a control character.

`char toupper(char c)` Converts `c` to an uppercase letter.

`char tolower(char c)` Converts `c` to a lowercase letter.

C.4 Mathematics

The mathematics library provides a number of general mathematical functions. More specialised functions and numerical algorithms are provided by other libraries. It is essential to import the file `math.h`; the compiler might not warn you if it is not included, but the functions will return random results.

```
#include <math.h>
```

The functions available in this library are:

`double sin( double rad )` Calculates the sine of an angle. The angle should be in radians.

`double cos( double rad )` Calculates the cosine of an angle

`double tan( double rad )` Calculates the tangent of an angle.

`double asin( double x )` Calculates the arc sine of `x`.

`double acos( double x )` Calculates the arc cosine of `x`.

`double atan( double x )` Calculates the arc tangent of `x`.

`double atan2( double x, double y )` Calculates the arc tangent of `y/x`. (A proper result is returned when `x` is 0.)

`double sinh( double rad )` Calculates the hyperbolic sine of `x`.

`double cosh( double rad )` Calculates the hyperbolic cosine of `x`.

`double tanh( double rad )` Calculates the hyperbolic tangent of `x`.

`double exp( double x )` Calculates the exponential function of a number, e^x.

`double log( double x )` Calculates the base e (natural) logarithm of `x`.

`double log10( double x )` Calculates the base 10 logarithm of `x`.

`double pow( double x, double p )` Calculates `x` to the power `p`, x^p.

`double sqrt( double x )` Calculates the square root of `x`, $\sqrt{x}$.

`double ceil( double x )` Calculate $\lceil x \rceil$, the smallest integer not less than x. The function returns a `double`, not an `int`. It does not perform a coercion, but it only rounds a floating point number.

`double floor( double x )` Calculate $\lceil x \rceil$, the largest integer not greater than x.

`double fabs( double x )` Returns the absolute value of `x`, $|x|$.

`double ldexp( double x, int n )` Returns $x2^n$ as a floating point number.

`double frexp( double x, int * n )` Splits `x` into a fraction f and a power of 2, p, such that $x = f2^p$ and $1/2 \leq f < 1$. The fraction is the return value of `frexp` and the power is assigned to `*n`.

`double modf( double x, double *i )` Takes a double, and splits it into its integer and fractional part. The integer part is returned via `i`, the fractional part is the return value of the function.

`double fmod( double x, double y )` Calculates `x` modulo `y`.

C.5 Variable argument lists

The include file `stdarg.h` provides the facilities needed to work with variable argument lists. This is how to include it into a module:

```
#include <stdarg.h>
```

Examples of functions that are implemented using this facility are `printf` and `scanf`. A function using the variable argument list facility must have at least one proper argument. The last proper argument must be followed by ellipses in the prototype. Here is an example:

```
void printf( char *format, ... ) ;
```

The variable argument list module provides the following type and macro definitions:

`va_list` This is the type of the data structure that provides access to the variable arguments. A variable of type `va_list` must be declared in each function using variable argument lists.

`va_start( va_list ap,` x `)` The `va_start` macro must be called once before processing of the variable argument list begins. The variable x should be the last proper argument before the ellipses (`...`) in the function prototype. In the case of `printf` above the x would be `format`.

t `va_arg( va_list ap,` t `)` The `va_arg` macro will deliver the next item from the argument list. This value has type t. Each call to `va_arg` advances to the next argument, until the argument list is exhausted.

`va_end( va_list ap )` The `va_end` macro terminates processing of the argument list. It should be called once before a function using the variable argument facility terminates.

C.6 Miscellaneous

The utility library is a collection of miscellaneous routines that did not fit anywhere else. The utility library can be used by including `stdlib.h`:

```
#include <stdlib.h>
```

This module contains a large number of functions. We discuss only the most important functions below:

`int abs( int x )` Returns the absolute value of an integer.

`int atoi( char *string )` Converts a string to an integer (the name stands for **a**scii **to** **i**nteger): `atoi( "123" )` is 123.

`double atof( char *string )` Converts a string to a floating point number.

`void *calloc( int x, int y )` Allocates heap space: sufficient space is allocated to hold `x` cells of size `y`. All space is initialised to 0. This function returns `NULL` if it cannot allocate sufficient space. You can use `sizeof` to find out how many bytes a certain type needs. The call `calloc( 4, sizeof( int ) )` will return a pointer to an area of store large enough to store 4 integers.

`void *malloc( int x )` Allocates `x` bytes of heap space.

`void free( void *ptr )` Frees a previously allocated block of heap space.

`void abort( void )` Stops the execution of the program immediately and dramatically. It might invoke a debugger; or leave a dump for post mortem examination. This function is used to terminate a program in case of a suspected programming error.

`void exit( int status )` Stops the program gracefully. The value of `status` is known as the *exit status* of the program. It signals whether the program has successfully accomplished its task. Zero indicates success, any non null value means a kind of failure (it is up to the programmer to specify and document which value indicates which failure). If the program terminates because `main` returns, the exit status is the value returned by `main`.

C.7 Other modules

There are 6 more modules in the C standard library. Each of these modules has an associated include file and a number of library functions. Below we give a brief description of each of these modules.

Diagnostics, include file `assert.h`. This module allows the programmer to verify that certain conditions are met (consistency check), the program is aborted if the condition fails.

Non local jumps, include file `setjmp.h`. A non local jump 'returns' from a number of nested function calls all at once and continues at a predetermined place somewhere else in the program.

Signals, include file `signal.h`. Signals are similar to SML exceptions. Signals can be caught and sent. The system may send signals to indicate that something went wrong, for example a reference through a dangling pointer.

Date and Time, include file `time.h`. These are functions to find out what time it is, and to convert time and date information into strings.

Integer limits, include file `limits.h`. This module defines constants that denote the maximum values that variables of type `int`, `char`, and so on, can take.

Floating point limits, include file `float.h`. This module defines constants that denote the maximum floating point number that can be represented, the number of bits in the mantissa and exponent, and so on.

Appendix D

ISO-C syntax diagrams

This chapter summarises the complete syntax of ISO-C using *railroad diagrams*. A railroad diagram has a name, a beginning at the top left hand side and an end at the top right hand side. A diagram is read starting from the beginning and following the lines and arcs to the end. Similar to what real trains on real rail roads can do, you must always follow smooth corners and never take a sharp turn.

On your way through a rail road diagram, you will encounter various symbols. There are two kinds of symbols. A symbol in a circle or an oval stands for itself. This is called a *terminal symbol*. Such a symbol represents text that may be typed as part of a syntactically correct C program. A symbol in a rectangular box is the name of another rail road diagram. This is a *non-terminal symbol*. To find out what such a symbol stands for you must lookup the corresponding diagram. Railroad diagrams can be recursive, when a non terminal is referring to the present diagram.

The rail road diagrams can be used for two purposes. The first is to check that a given C program uses the correct syntax. This should be done by starting at the first diagram (*translation˙unit*), and trying to find a path through the diagrams such that all symbols in the program are matched to symbols found on the way.

Railroad diagrams are also useful as a reminder of what the syntax exactly looks like. A path through the diagrams corresponds to an ordering on the symbols that you may use. For example if you what to know what a `for`-statement looks like, you should look up the diagram called statement, find the keyword `for`, and follow a path to the end of the diagram to see what you may write to create a `for`-statement. Note that the railroad diagrams only describe the *syntax* of the language. A syntactically correct program is not necessarily accepted by the compiler as it may contain *semantic* errors (for example an illegal combination of types).

The diagrams that represent the ISO-C syntax are ordered in a top down fashion. We explain a few diagrams in some detail to help you find out for yourself how to work with them. Let us study what a program or *translation˙unit* may look like.

The interpretation of this diagram is as follows: a *translation˙unit* can be either a *function˙definition* or a *declaration*, optionally followed by another *function˙definition* or *declaration*, and so on. It is thus possible to give an arbitrary number of either *function˙definition* or *declaration*. There must be at least one of either.

Now that we know what the two main components of a program are, we should like to know more about each of them. Their definitions are:

As an example to show how the railroad diagram applies to a C constant declaration, consider the following:

```
const double eps=0.001, delta=0.0001 ;
```

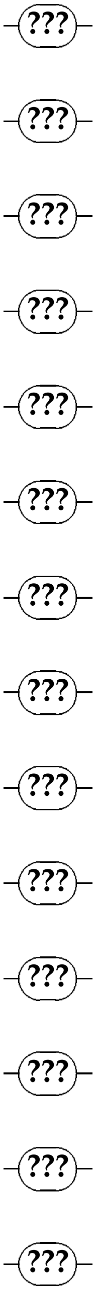
The words `const double` are matched by the diagram *declaration˙specifiers* (via *type˙specifier* and *type˙qualifier*). We then take the road to the lower part of *declaration*, where `eps` matches *declarator*, we take the road to the `=`, where the equal sign matches, the constant `0.001` matches *initializer*. The comma brings us back at the *declarator*, which matches `delta`, and so on.

The *type˙qualifier* diagram shows that we have not explained everything there is to know about C, because it shows a new keyword `volatile`. We are not going to explain such new features here, we should just like to point out that the syntax given as rail road diagrams is complete. If you want to find out more bout the keyword `volatile` you should consult the C reference manual [7].

Types are constructed using structures, unions and enumerations, as discussed in Chapter 4.

The following syntax is used to declare arguments of functions as well as local and global variables. This syntax includes the definitions of function types (to be passed to higher order functions), array types, and pointer types.

The priorities of operators are expressed by a series of railroad diagrams that show increasing priority. Note that the priorities can pose a few surprises, such as that the bitwise operators `&`, `|` and `^` have a lower priority than the relational operators: `x&1 == 0` does not test whether the last bit of `x` is `0`, but instead calculates `x&(1==0)` which happens to be `0` for all `x`.



There are no diagrams for the symbols that represent identifiers, constants and strings, they would not give much useful information. Instead we give an informal definition of each of these terms:

identifier A sequence composed of characters, underscores, and digits that does not start with a digit.

integer˙constant A decimal number, or a hexadecimal number prefixed with `0x`, or an octal number prefixed with a `0`.

character˙constant A single character enclosed in characters, or a backslash escape (a full list is given in Section 2.3.3).

floating˙constant A number consisting of a integral part, a fractional part, and an exponent. The exponent must be prefixed with the letter `e` (or `E`), the fractional part must be prefixed with a decimal point. One of the fractional and exponential part is optional.

enumeration˙constant An identifier that has appeared in an `enum` declaration.

string A sequence of characters enclosed between double quotes, `"`.