Functional Programming Using Standard ML

By

Dr. E. K. Mugisa

Department of Mathematics & Computer Science

The University of the West Indies (Mona)

0. An overview of functional programming

0.0 This Manuscript Introduces The Functional Programming Style

At one time almost all programming was sequential and imperative (or procedural or statement-oriented). This is what the von Neumann machine architecture dictates. So if you are programming to the underlying von Neumann architecture you program sequentially and imperatively. In recent years, mathematical models of programming have been gaining in importance at the expense of the traditional styles. In the 1980's Computer Science departments placed a lot of emphasis on structured programming as in Pascal or a similar language. In the 1990's structured programming is no longer enough; mathematical models are going to become increasingly important. Two such important models are Logic and Functional programming.

Logic Programming and Functional Programming came via Artificial Intelligence. This should not be surprising, after all AI deals with problems that traditional approaches cannot handle. With the ever increasing need for mathematics to take its proper place in Computer Science, today's graduate needs to master logic programming and functional programming as examples of programming using a mathematical rather than a machine model. These two styles of programming are no longer restricted to Artificial Intelligence, however. They have now become truly general-purpose, and therefore their coverage is no longer a trip into specialist territory.

This course will introduce all the important features of functional programming. These include the following:

1. Functions:

A function as a mapping from one set of values (type) called the domain to another set called the range.

2. Mathematical variables:

Variables in functional programs cannot be updated. They get their values (values are bound to them) from outside or by definition.

3. Expressions:

An expression is made up of function applications and is evaluated (reduced) in Applicative Order or in Normal Order.

4. Referential Transparency:

The value of an expression is determined solely from the values of its sub-expressions. Two sub-expressions are equivalent if they have the same value. Equivalent sub-expressions may be interchanged.

5. Typing:

Most functions are data-specific i.e. they are defined for one type of object only. However, some functions are overloaded and others are polymorphic. An overloaded function has several meanings while a polymorphic function is type-independent. Polymorphism is desirable in many applications e.g. in designing generic structures.

6. Data Structures:

Data structures enable us to construct large data objects from other data objects.

7. Abstract Data Types:

In abstract data types the concrete type is hidden from the user. Functions are abstract.

8. Lists:

The list is the basic data structure of functional programs.

9. Lazy Evaluation:

Lazy evaluation enables us to manipulate infinite lists like [1,2, 3, ...]. Without lazy evaluation it is difficult to give infinite lists computational meaning.

10. Recursion:

Recursion eliminates looping constructs. Recursion is easier to understand than iteration.

11. Lambda binding:

It is useful for anonymous functions.

12. Higher Order Functions:

Functions themselves are computational values and therefore can be manipulated in expressions. A higher order function is a function that operates on other functions.

These concepts will be illustrated with several examples. We shall start by looking at imperative programming, since that is where we are coming from with our Pascal tradition.

1. A Quick Overview of Imperative Programming

1.0 Imperative Programming Has Three Main Characteristics

Imperative programming is sometimes called procedural programming or statement-oriented programming. In an imperative program the basic "unit of work" is the statement. The effect of a statement is to change the state of the machine on which the statement is executed. The imperative style of programming is therefore characterised by its use of

1. variables to stand for storage cells,
2. the assignment statement to change the contents of a cell and therefore the state of the machine,
3. repetition (looping) to construct complicated applications.

The following Pascal program illustrates the imperative programming style. It is taken from (Ghezzi, 1987) page 261. Note the use of variables as storage cells, the assignment statement and repetition. The interdependence of the two loops makes it difficult to follow the program.

```
(* print prime numbers in the range 2 .. n *)

program primes (input, output);

  const n = 50;
  var  i : 2 .. n;
    j : 2 .. 25;
    i_is_prime : boolean;
  begin
    for i := 2 to n do
      begin (* is i prime ?*)
        j := 2; i_is_prime := true;
        while i_is_prime and (j <= i div 2) do
          if ((i mod j) <> 0) then

            j := j + 1

          else i_is_prime := false;

        (* if so, print its value *)
        if i_is_prime then write (i)
      end

  end.
```

This Pascal program takes each number in the range and checks to see if the number has any divisors. If it does not then it is prime.

1.1 There Are Problems With Imperative Programming

1. Programming at the level of manipulating the memory store is uncomfortably low, especially if the application is complicated.
2. von Neumann programs are therefore very difficult to reason about. That is, it is difficult to attempt to show (or prove) that such a program is correct.
   The execution of a von Neumann program defines a sequence of states of computation. A state of computation is defined by the contents of memory cells. This state changes every time an assignment statement is executed. A complicated application therefore requires several states of execution, actually too many to reason comfortably about. No wonder we still have the software crisis!

   It is interesting to note that these problems were ignored for 2 to 3 decades in favour of efficiency. In other words, faster execution of possibly incorrect programs was (and still is?) preferred to slower execution of correct programs!

References

1. Ghezzi, c., Jazareyi, M. (1987), 'Programming Language Concepts', 2/E : John Wiley, Chapter 7, Section 1, pp 259 - 264

2. Functions Are At The Center Of Functional Programming

2.0 Values Instead Of States Of Computation

Functional programming is expression-oriented and an expression consists of function applications. A function application, and therefore an expression, has a value. We have seen that statement-oriented imperative programming has the effect of changing the state of computation of the underlying machine. Expression-oriented functional programming on the other hand produces values. These values are all want; we do not care about the state of the machine on which the expressions are evaluated.

Let us begin by contrasting imperative functions and functional functions. We shall consider Euclid's Algorithm for computing the Greatest Common Divisor (GCD) of two natural numbers.

The algorithm states that

gcd(0,n) = n
gcd(m,n) = gcd(n mod m, m) for m > 0

A typical imperative gcd function looks like this (taken from (Paulson, 1991)):

```
function gcd (m,n : integer) : integer;
var prevm : integer;
 begin
   while m <> 0 do
     begin
       prevm := m;
       m := n mod m;
       n := prevm
     end;

    gcd := n

 end;
```

If we use recursion it looks like this:

```
function gcd (m, n : integer) : integer;
begin
   if m = 0 then
     gcd := n
    else
      gcd := gcd(n mod m, m)
end;
```

The recursive version is usually avoided because "recursion is inefficient." In a functional style of programming Euclid's GCD function is always coded as in the following Standard ML function

```
fun gcd(m,n) = if m = 0 then n  else gcd (n mod m, m);
```

Here correctness is more important than efficiency. This function is obviously a correct implementation of Euclid's algorithm. It is not so easy to prove that the iterative implementation in Pascal is correct. If we test it on a carefully selected subset of the domain we may convince ourselves (and perhaps others too) that it is good enough. But is such testing enough? There is abundant evidence in the form of programs with bugs to suggest that testing is not enough! Testing has traditionally been used to show the presence of errors but not to show that they are absent.

2.1 A Function As A Mapping

Functions in functional programs define a mapping from one set of values (a type) called the domain to another set of values called the range.

A function definition defines the domain, the range and the mapping rule. Let us look at simple examples:

```
- fun add1 x = x + 1;          (* we type in this definition *)
- val add1 = fn : int → int    (* we get this response *)
```

add1 is a function which maps integers to integers. The mapping rule adds 1 to the domain value. Applying add1, we have:

add1 5 to give 5 + 1 which evaluates to 6. add1 5 = 5 + 1 = 6

```
- fun positive x = if x > 0 then true else false;
    - val positive = fn : int → bool
```

positive is a function which maps integers to booleans. The mapping rule maps integers greater than zero to true and all others to false.

positive 6 gives true.

```
- fun gcd (m, n) = if m = 0 then n else gcd (n mod m, m);
    - val gcd = fn : int * int → int
```

gcd maps a pair (or 2-tuple) of integers to an integer. The mapping rule is a bit more involved than in 1 and 2.

gcd(65, 39) = gcd (39, 65) = gcd (26, 39) = gcd (13, 26) = gcd (0, 13) = 13

The three examples are intended to show clearly that

1. A function is indeed a mapping from one set of values (or type) to another.
2. The mapping rule is given by an expression, sometimes aided by a structuring mechanism such as selection in the if expression.
3. There is no idea of a computational state, only values.
4. There are no assignments.

Have you noticed that the von Neumann architecture has disappeared from the scene? Where is it? It is hidden behind the scenes, as it should be. In fact it can even be replaced. Our functions do not care what architecture they are evaluated on because they are based on a mathematical model and not on a particular machine architecture. If one machine architecture can implement the functions more efficiently than another, then we will probably choose the more efficient one but correctness will not be compromised. Advances in hardware technology (speed, reliability, etc) and software technology (language implementation techniques, etc) have reduced the efficiency gap between functional and imperative programs. Efficiency is therefore no longer a strong issue against functional programs. Imperative programs on the other hand are still difficult to reason about.

Let us end with the factorial function.

```
- fun fact n = if n = 0 then 1 else n * fact n-1;
- val fact = fn : int → int
```

Let us try to evaluate fact 3

```
fact 3 = 3 * fact 2
  = 3 * 2 * fact 1
  = 3 * 2 * 1 * fact 0
  = 3 * 2 * 1 * 1
  =6    *1*1
  =6    * 1
 = 6
```

2.2 Exercises

2.1 Define the following functions. In each case identify the domain, range and mapping rule.

(i) The square of a number
(ii) The maximum of two numbers
(iii) The maximum of three numbers
(iv) Is a given number prime (True/False)?

2.2 What differences do you notice between imperative functions and functional functions?

References

1. Paulson, L.(2000), 'ML for the Working Programmer' 2nd Edition: Cambridge University Press

3. Expressions

3.0 Expressions Instead Of Statements

Expressions are to functional programming what statements are to imperative programming. “Execution” in a functional program means evaluating an expression. An expression has a value. When an expression is evaluated it is reduced to its value. This is very easily done. Isolate reducible sub-expressions of the expression and reduce each such reducible sub-expression (or redex) to its value. Keep doing this until you get a non-reducible sub-expression. Simply said, replace equals by equals in the expression.

On the other hand execution in an imperative program means executing statements. This means changing the computational state by assigning new values to variables, creating new (local) variables, destroying variables, etc. All this is very difficult to define and to follow.

3.1 An Expression Contains Function Applications

An imperative programming language gives us basic statements like the assignment statement and basic constructors like selection etc, which we then use to construct larger statements. In a functional program like Standard ML we start off with a set of basic functions which we can use to form simple expressions. In order to form more complicated expressions we structure these simpler expressions into a series of function applications using the structuring mechanisms provided.

3.1.1 Standard ML’s Predefined Functions

Here is a full list of predefined functions in Standard ML. They are categorised by type.

Integers (ML type int)

```
+     addition
-     subtraction
~     unary minus

*     multiplication
div   division
mod   remainder
real  conversion to real
abs   absolute value
<,>,<=,>=,=,<>    relational operators
```

Reals (ML type real)

```
+     addition
-     subtraction
~     unary minus

*     multiplication

/     division
sqrt  square root
sin, cos, arctan    trigonometric functions
exp   exponentiation
ln    natural logarithm
abs   absolute value
floor conversion to integer
<,>   relational operators
```

Character strings (ML type string)

```
^           concatenation
size        number of characters in a string
```

```
ord, chr    string - ASCII code conversion
explode     string to list conversion
implode     list to string conversion
```

Booleans (ML type bool)

```
not         negation
orelse      disjunction
andalso     conjunction
=, <>       relational operators
```

Tuples

#i select component i (e.g. #1, #2, ...)

Records

#label field selection

Lists (ML type 'a list)

```
nil, []     empty list
::    constructor
@     append
rev   reverse
map   apply a function to each element of a list
```

Functions (ML type fn)

function application

; function composition

3.1.2 A Function Application May Be Infix Or Postfix

An infix function (otherwise called an infix operator) is written between its two arguments as in a1 f a2, where a1 and a2 are the arguments and f is the infix function. Familiar examples of infix syntax include the following: a + b; 5 * 4; a mod i.

Normally, functions are applied using prefix notation, whereby the function precedes its arguments, as in f (a1, a2). The brackets may be left out to give f a1 a2. Standard ML allows us to define infix functions as in the following example definition of the xor function (exclusive or).

```
infix xor
fun p xor q = (p orelse q) andalso not (p andalso q);
```

The function xor is now applied using infix syntax as follows:

```
true xor false; a xor b;
```

Note that the value of the function or its mapping rule is the same whether the syntax of its application is infix or prefix.

In prefix notation the xor function would be defined as follows:

```
fun xor p q = (p orelse q) andalso not (p andalso q);
```

and it would be applied as follows:

```
xor true false; xor a b
```

3.1.3 There Are Several Ways Of Evaluating Expressions

An expression stands for a value. The expression true xor false stands for the value true just as the expression 1 + 2 stands for the value 3. Expression evaluation involves rewriting the expression or reducing the expression to its "normal" form. Thus the expression true xor false is reduced to true and 1 + 2 is reduced to 3. A reducible expression or sub-expression is called a redex. Four reduction strategies are equivalent. They are

outermost rightmost
outermost leftmost
innermost rightmost
innermost leftmost

Let us illustrate these four reduction strategies with an example.

Define function square.

```
fun square x = x * x;
```

Now evaluate the expression *square (2+2)* using the four strategies mentioned above:

(i) Outermost Rightmost Reduction

Apply the outermost function first and proceed from right to left.

```
square (2+2) = (2+2) * (2+2)          - apply square
             = (2+2) * 4              - reduce rightmost redex
             = 4 * 4
             = 16
```

(ii) Outermost Leftmost Reduction [Also called Normal Order Reduction]

Apply the outermost function first and proceed from left to right.

```
square (2+2) = (2+2) * (2+2)          - apply square
             = 4 * (2+2)              - reduce leftmost redex
             = 4 * 4
             = 16
```

(iii)Innermost Rightmost Reduction

Apply the innermost function first and proceed from right to left.

```
square (2+2) = square (4)             - apply +
             = 4 * 4
             = 16
```

(iv) Innermost Leftmost Reduction [Also called Applicative Order Reduction]

Apply the innermost function first and proceed from left to right.

```
square (2+2) = square (4)             - apply +
             = 4  * 4
             = 16
```

It can be proved that all the four reduction strategies are equivalent in that they all produce the same value for a given redex. The Applicative Order reduction strategy (in fact all innermost first strategies) implements call-by-value (or strict evaluation). Call-by-value means that we evaluate the innermost redex first. In other words evaluate the arguments of a function first and then call the function with the evaluated arguments. One problem with the innermost first strategy is that sometimes it performs redundant evaluations.

Here is an example.

```
fun zero x = 0;
```

Evaluating zero (4+6) using Applicative Order gives

zero(4+6) = zero(10)

= 0

It was unnecessary to evaluate the expression 4+6. Whatever argument it is applied to, function zero will always return (will reduce to) the value 0. It would be useful to have a reduction strategy that exploits this fact in this case. Normal Order does just that.

The Normal Order (or Outermost First) reduction strategy implements call-by-name (or call-by-need or lazy evaluation). Call by name here means that we evaluate the outermost redex first. In other words apply the function to its un-simplified arguments. The danger here is that we may duplicate computation. For example we duplicate the computation of (2+2) in (i) and (ii) above.

If an expression has a value then Normal Order will get it but Applicative Order may fail. This happens with functions that violate rules of mathematics. Let us look at a simple example:

What is the value of zero *(1 div 0)*? What about *zero ("You" + 5)*?

Since the arguments cannot be evaluated, Applicative Order will fail. Normal Order on the other hand will succeed since it does not need to evaluate the arguments in this case.

ML uses Applicative Order because it is "more efficient". More recent functional languages tend to use Normal Order because it is mathematically more sound.

3.2 Conditional Expressions Give Us Choice

The conditional expression in Standard ML has the form

if E then E1 else E2

E is a boolean expression. If E has the value true then the value of the conditional expression is the value of expression E1. If the value of E is false then the value of the conditional expression is that of expression E2. The un-chosen expression is not evaluated; this is very important for recursive functions. The conditional expression was used in the definition of the factorial function. That definition is repeated here.

fun fact n = if n = 0 then 1 else n * fact(n-1)

ML's boolean operators *andalso* and *orelse* are equivalent to boolean expressions as follows:

```
E1 andalso E2    is equivalent to       if E1 then E2 else false
E1 orelse E2     is equivalent to       if E1 then true else E2
```

3.3 Exercises

3.1 Using the definition of square and zero given above evaluate the following in a) Applicative Order b) Normal Order.

(i) square(square(square(2)))
(ii) zero(square(square(2)))

3.2 Evaluate fact(5), where fact is defined as
fun fact n = if n = 0 then 1 else n * fact (n-1)

3.3 Evaluate facti(5,1), where facti is given by

fun facti (n, p) = if n = 0 then p else facti(n-1, n*p);

3.4 Perform a reduction of the expressions powoftwo(8) and powoftwo(12) given the following definitions :

fun even n = (n mod 2 = 0);

fun powoftwo n = (n=1) orelse (even(n) andalso powoftwo(n div 2));

3.5 We are given the following definitions for cond and badf:

```
fun cond(p, x, y) = if p then x else y;
fun badf n = cond (n = 0, 1, n*badf(n-1);
```

Try to reduce the expression badf(0).

Why does the reduction fail ?
Suggest a way or ways of overcoming the problem.

References

1. Paulson L. C. (1991), 'ML for the Working Programmer' : Cambridge University Press. Chapter 2
2. Sethi, R (1989), 'Programming Languages - Concepts and Constructs' : Addison-Wesley. Chapter 2

4. Functional Variables are Mathematical Variables

4.0 Functional Variables Cannot Be Updated

Variables in imperative programs stand for storage cells in the von Neumann computer's store. Such variables can be assigned different values at different stages of execution of an imperative program. This is how the state of computation changes in an imperative program.

In functional programs we do not have the concept of a von Neumann store. Variables stand for values not storage cells. A variable is simply a name for a value. A variable identifies a value. The concept of updating a functional variable is meaningless.

In a function a variable is simply a placeholder used in defining the mapping rule, which associates a domain value with a range value. A value is bound to the variable before function application. If a variable were correctly used in a function application, it would be standing for a value that it got by definition, before function application.

4.1 Standard ML Uses Static Binding Of Variables

In Standard ML values are bound to variables in a val declaration. The syntax is as follows:

```
val varname1   = subexp1
[and varname2  = subexp2
. . .
. . .
and varnamen  = subexpn];
```

The sub-expressions are all evaluated and their values bound to their respective variables in parallel.

The val declaration

```
val x = 5;
```

may be interpreted to be saying that the value of variable x is 5.

4.1.2 A Global Binding Remains in Force Until the Variable is Re-Bound

A variable may be bound globally as in the following declarations:

```
val pi = 3.14159;
val r  = 2.0;
val area = pi * r * r;
```

Variables *pi, r* and *area* may be used in subsequent expressions to stand for the values they have been assigned above. These variables will keep their values until they are redeclared (or re-bound) in subsequent val declarations.

4.1.3 A Local Binding Survives Only One Expression or Declaration

Variables may be bound locally as in the following :

```
let val x = 3 and y = 5
in x * y - (x + y) end;
```

or

```
local val x = 70 div 3
in val y = (x, x*3) end;
```

In each of these two examples x is bound outside the expression in which it is used and yet the binding remains local to the relevant expression or declaration.

The syntax of local binding is

```
let <declaration> in <expression> end;
local <declaration> in <declaration> end;
```

We use a local binding if we want a variable to have (be bound to) a particular value only for the duration of a particular expression (for let) or a particular declaration (for local). Outside the particular expression or declaration of interest the variable loses the value it got in the local binding.

In the case of a global binding, a variable keeps its value until it is re-declared. In a local binding a variable keeps its value only during the local expression or declaration. The declarations used in the above examples are value declarations (using val). The syntax of value declarations in its simplest form was given earlier. There are twelve different declarations in Standard ML. See the language definition (e.g. Syntax Charts) for details.

Static binding means that when we overwrite some value binding, the value of the identifier does not change, but instead a new identifier with the same name is created.

An example may help illustrate the point.

```
- val y = 12;
- val y = 12 : int
- fun addy x = x + y;
- val addy = fn : int → int
- addy 3;
- 15 : int
```

Now change y's binding

```
- val y = true;
- val y = true : bool
- addy 3;
- 15 : int
```

The value of y in addy is still 12. This value cannot be changed since variables cannot be updated. However, the original y has been hidden by the new one since they both have the same name. Lisp does not use static binding.

4.2 Exercises

4.1 What is the value of each of the following expressions?

(i) let val x = 2 in x * x end;
(ii) let val z = 3 in z * z end;
(iii)let val x = 2;

```
let val x = 3
and val y = x
in x + y end;
```

(iv) let val x = 3 in let val y = 4 in x * x + y + y end end;
(v) let val x = 2 in let val x = x + 1 in x * x end end;
(vi) let val x = 3 val y = 4 in x * x + y * y end;
(vii)let val x = 3 and y = 5 in x * y - (x + y) end;
(viii)local val x = 70 div 3 in val y = (x, x+3) end;
(ix) let val x = 64.0 in sqrt x end;
(x) (fn x => sqrt x + x) 64.0
(xi) Does the following swap work? Explain.

```
val one = "Bang";

val two = "Big";
val one = two and two = one;
```

4.2 What values are given by each of the following in the sequence?

```
fun f1 x = x * 2;
fun f2 x = f1 x;
f2 2;
fun f1 x = x - 3;
f2 2;
```

4.3 Reduce the following expression.

```
let fun f1 x = x + 2
in let fun f2 x = f1 x
in let fun f1 x = x - 3 in f2 2 end end end;
```

4.4 Rewrite the following function using local declarations :

Reducing a fraction to least terms :

```
fun fraction (n,d) = (n div gcd(n,d), d div gcd(n,d));
```

4.5 The following is taken from Paulson, 1991). Study it until it makes sense.

To compute the square root of a positive number a using the Newton-Raphson method we proceed as follows :

- choose any positive number, say 1, as the first approximation.
- if x is the current approximation then the next approximation is (a/x + x) / 2
- stop as soon as the difference becomes small enough.

Here is an implementation.

```
fun sqroot a =
        let val acc = 1.0E~10
fun findroot x =
        let val nextx = (a/x + x) / 2.0
        in if abs(x - nextx) < acc * x
              then nextx
              else findroot nextx
    end

    in findroot 1.0

    end;
```

5. Referential Transparency Aids Correctness Proofs

5.0 An Expression Always Has The Same Value

Referential transparency means that an expression always has the same value. Whether you evaluate the expression on planet Earth or whether you evaluate it on Mars, you get the same value. This emphasises the point that the only important thing about an expression is its value. The environment within which it is evaluated or reduced is not important. If we take a sub-expression of an expression and replace this sub-expression with another sub-expression that has the same value then the original expression has not changed. Another way of saying this is that "an expression does not change if we replace equals with equals."

In fact an entire expression may be replaced by its value. Let us look at an example. The commutative law of addition says that E1 + E2 can be replaced by E2 + E1. Generally, expressions can be transformed using mathematical laws. The sub-expressions can be evaluated in any order or even in parallel.

(Ghezzi, 1987) has the following to say on referential transparency (page 263): "A system is said to be referentially transparent if the meaning of the whole can be determined from the meaning of its parts. Mathematical expressions - indeed, all mathematical concepts - are referentially transparent. For example, in the mathematical expression f(x)+g(x), we can substitute another function f' for f if we know that it has the same value as f. If the same expression is an expression in Pascal (or any other conventional imperative language), we are not assured of this property. Indeed, if f or g change the values of their ... by-reference parameter or modify some global variables, we are not even assured that f(x) + g(x) = g(x) + f(x) or f(x) + f(x) = 2 * f(x), that is, the meaning of the expression depends on the history of computation of the sub-expressions. Lack of referential transparency makes programs hard to read, modify and prove correct." It is interesting to note that the mechanisms that harm referential transparency are introduced into imperative languages to achieve execution efficiency on a von Neumann computer. These mechanisms include global variables, call-by-reference parameters and side effects in general.

Because of referential transparency we are able to say, quite often with great authority, that a given functional function is correct. On the other hand, because of the absence of referential transparency we usually are unable to say, without doubt, whether a given imperative function is definitely correct. If we are concerned about program correctness we have to enforce, or at least encourage, referential transparency in our functions.

Exercises

6. Lists Are The Basic Data Structure Of Functional Programs

6.0 Lists for arrays

Programmers process collections of items using any one of several structuring methods: records if the items are few, related and (normally) of different types; arrays if the collection is too large to handle comfortably as a record and if the items are of the same type.

Arrays have been and probably still are at the center of data structuring in imperative programming. Their main asset is their great efficiency. The major advantage of the array is that it takes the same length of time to access the first element of the array as it does to access the last or any other element of the array. As a result the array is called a random-access structure. Efficient access to individual array elements keeps the array ahead of its competitors. There are serious disadvantages too. The main problem seems to be subscripting. If we remove subscripting as in stacks, queues, etc programmers tend to be more comfortable with the resulting structures. Such structures do not suffer from the subscripting errors, which are a cause of many bugs in manipulations of arrays. Many bugs have resulted from using non-existent array subscripts. Functional programs process collections of items as lists. From a mathematical point of view lists are more elegant than arrays.

6.1 A List Is A Finite Sequence Of Elements

Since a list is a sequence the order of elements is important. Here are a few lists.

```
(i)   []   the empty list.
(ii)  nil  the empty list again.
(iii) [1]  a singleton list.
(iv)  [1,4,5]   the sequence of elements 1,4,5
(v)   [”one”, “tea”, “book”]   a list of strings
(vi)  [(1,”one”), (2, “two”), (3, “three”)]   a list of pairs or 2-tuples
```

All the elements of a list must be of the same type.

6.2 A List Is Built From Nil Using cons (::)

The starting point for list construction is the empty list nil or [] and the constructor primitive::, called cons (for constructor or construct). A list has a head and a tail. The two are related by cons as follows: list = head :: tail.

e.g.

[1] = 1 :: []
[1] = 1 :: nil
[1,4,5] = 1 :: [4,5]
[5,1,4] = 5 :: [1,4] = 5 :: (1 :: (4 :: []))
[”one”, “tea”, “book”] = “one” :: [”tea”, “book”] = “one” :: (“tea” :: (“book” :: []))

6.3 Lists Have A Rich Toolkit

The toolkit contains tools to do the following :

- determine the length of a list
- determine the last element of a list
- reverse a list
- append two lists to form a new list
- apply a function to each element of a list
- convert a list of lists into a flat list

The functions hd (head), tl (tail), rev (reverse) and @ (append) are normally predefined. The rest of the list operators are added to the toolkit as user-defined functions. Let us write a function to determine the length of a list.

```
fun len x = if x = [] then 0
            else 1 + len(tl x);
```

Let us now try to write the function *hd*. We soon discover that it is not easy to write it in the style of *len*. To greatly simplify the implementation of list functions we need to turn to the concept of pattern matching.

6.3 Pattern Matching Simplifies List Functions

Let us start by rewriting function len using pattern matching.

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

Since we do not use variable *x* in the expression, we should eliminate it from the function definition.

```
fun len [] = 0
    | len (_ :: xs) = 1 + len(xs);
```

The function has one clause for each argument pattern. The first pattern, which matches the empty list, is separated from the second by a vertical bar. The second pattern is invoked only if the first fails to match the actual argument. The patterns must therefore be mutually exclusive. They should also be exhaustive. This means that an actual argument should not fail to find a matching pattern.

Let us now write the rest of the list toolkit, starting with *hd*. Some of these patterns are not exhaustive. We shall deal with this problem later.

1. head : the head of a list

```
fun hd (x :: _) = x;
```

2. tail : the tail of a list (what is left after removing the head)

```
fun tl (_ :: xs) = xs;
```

3. append : append two lists

```
fun append ([], y) = y
| append (x, []) = x
| append (x :: xs, y) = x :: append (xs, y);
```

This function could be defined as

```
infix @;
fun [] @ y = y
    | x @ [] = x
    | (x :: xs) @ y = x :: (xs @ y);
```

4. rev : reverse a list

```
fun rev [] = []
| rev (x :: xs) = (rev xs) @ [x];
```

5. last : the last element in the list

```
fun last x :: nil = x
        | last x :: xs = last xs;
        OR
fun last x = hd rev x;
```

6. map : apply a function to each element of a list

```
fun map f [] = []
| map f (x :: xs) = f x :: map f xs;
```

7. flat : flatten a list of lists

```
fun flat [] = []
| flat (x :: xs) = [x] @ flat xs;
```

The list is a very powerful structure in functional programs. With lists we can do almost anything. The exercises that follow are supposed to illustrate the versatility of the list.

6.6 Exercises

6.1 Write a function nth(l, n) to return the nth element of list l. The head is element 0.

6.2 Write a function to transpose a matrix. For this function, the matrix can be seen as a list of lists.

6.3 Write a simple version of Quicksort using lists.

6.4 Implement set operations using lists. the operations are :

```
(i) convert a list to a set by removing duplicate elements
(ii) cardinality
(iii) membership
(iv) intersection
(v) union
(vi) difference
```

6.5 Implement a line editor. The line to be edited will be represented as a pair of lists. The first list of the pair will contain the items to the left of the cursor. The second list will contain the items to the right of the cursor. Write functions to do the following:

(i) move the cursor to the left (left)
(ii) move the cursor to the right (right)
(iii) delete the element immediately to the left of the cursor (delete)
(iv) insert an item to the immediate left of the cursor (insert)

7. Typing

7.0 Values Are Partitioned Into Types

In section 2.0 we stated that the value is at the center of functional programming. Expressions and function applications are important for the values they can be reduced to. The concept of typing partitions the universe of values into organised collections or sets, called types. In section 3.1.1, while looking at ML's predefined functions we encountered eight types: integer, real, string, boolean, tuple, record, list, function. We may divide these types into basic types and derived types as follows:

SML types
  Basic types (the values are given as primitives)
    integer, real, string, boolean
  Derived types (constructed from other types)

    tuple, record, list, function

Let us start by looking at each of these types. The predefined functions for each of these types are given in section

3.1.1

    Integer (int)
    Real (real)
    Boolean (bool)

These types are similar to the corresponding types in other programming languages like Pascal and Modula-2.

7.0.1 Character string (string)

ML strings are enclosed in double quotes (“) as are Pascal and C strings. Modula-2 strings are enclosed in single quotes (‘) e.g. “a”, “UWI-Mona”. Apart from relational operators section 3.1.1 introduced six specialist string functions.

Let us play around.

concatenation (^)

```
"comp" ^ "uter";          (this is what you enter)
"computer" : string       (this is the response)
```

size

```
size "what is my size?"
16 : nit
```

ordinal value (or position) in ASCII character set (ord)

```
ord "a";
97 : int
```

chr - the inverse of ord

```
chr 97;
"a" : string
chr 53;
"5" : string
```

You will have noticed that there is no need for a character type. A character is simply a string of length 1.

convert a string to a list of characters (explode)

```
explode “E.K.Mugisa”;
> [”E”,”.”,”K”,”.”,”M”,”u”,”g”,”i”,”s”,”a”] : string list
```

convert a list of strings to a string (implode)

```
implode [”I”,”did”,”it”];
“Ididit” : string
```

Let us now write a function to reverse a string.

convert the string into a list (explode)
reverse the list (rev)
convert the list into a string (implode)

```
- fun revstring s = implode(rev(explode s));
- revstring = fn : string → string
```

Is a given string a palindrome? A palindrome reads the same from left to right as from right to left e.g. “madam”, “ABBA”.

```
- fun palindrome s = s = revstring s;
- palindrome = fn : string → bool
```

Since we have the two functions explode and implode, we can use list functions to manipulate strings. This eliminates the need for several specialist string functions.

7.0.2 Tuples

A tuple is an ordered collection of values much like the Cartesian product of sets. The simplest tuple is the pair. The following are all tuples : (1, “a”); (5,12,13); (“uwi”, 1948, 3, 6).

The tuple (1,”a”) is formed by selecting 1 from the set of integers (or the type int) and “a” from the type string. This may be represented by the Cartesian product of the two sets int X string. Indeed the type of the tuple (1,”a”) is int * string.

A tuple may contain values of any type. This is a valid tuple : (2, (true, .6)), it has the type int * (bool * real). Let us look at some uses of tuples:

A function to test if a number is odd

```
fun odd x = if x mod 2 = 0
      then (x, “is even”)
      else (x, “is odd”);
val odd = fn : int → int * string
```

A function to do ordinary primary school “quotient-remainder” division:

```
fun divide (x, y) = (x div y, x mod y);
val divide = fn : int * int → int * int
```

Note that if we leave out the brackets we get a different argument type:

```
fun divide x y = (x div y, x mod y);
val divide = fn : int → (int → int * int)
```

What we see in this signature is called Currying and we shall say more about it later. We note from the signature (or type) of the function divide (x,y) that it has only one argument, not two. The single argument is a pair. Therefore the function application divide (11,3) has the vale (3,2), while the attempted application divide 11 3 will be rejected because of the wrong type of argument.

Finally on tuples let us look at their selectors. #1 selects the first element. Thus #1(“a”, 1) is “a” and #2(2, (true, 5.6)) is (true, 5.6). Generalising, selector #i will select the i^{th} element of an n-tuple, n > i.

7.0.3 Records

A record may be seen as a tuple whose components have labels. The components of a tuple are identified by position. The components of a record are identified by their labels. Let us look at a student record.

```
val student1 = {name="Jones", idnum=901234, faculty="Nat Sci"};
val student1 =  ...
```

To select the id number we use #idnum student1

```
#idnum student1;
901234 : int
```

#name {name="Brown", idnum=894321, faculty="FAGS"} has the value "Brown".

Tuples are a special case of records. Here is why.

a positive integer can be used as a label as in {1=x1, 2=x2, 3=x3}
to select the second component we use #2{1=x1, 2=x2, 3=x3}
as a tuple this would have been written as (x1,x2,x3) and the second component would have been selected as #2(x1,x2,x3).

We shall encounter records again.

7.0.4 Lists

We saw these in chapter 6.

7.0.5 Functions

Functions are types as can be seen from the following examples:

Function divide that we saw earlier

```
fun divide x y = (x div y, x mod y);
val divide = fn : int → (int → int * int)
```

The signature states that divide is a function that takes a value of type int and returns a value of type (int → int * int). But the signature of the returned value is a function signature. This means that the returned value is of type function. The returned value is a function.

Functional composition is a standard function defined as follows :

```
infix o;
fun (f o g) x = f(g x);
val o = fn : ('b → 'c) * ('a → 'b) → ('a → 'c)
```

Here 'a, 'b, 'c are called type variables. They take on values at function application time. A concrete example may help demystify the signature of o. Take the function revstring that we defined earlier.

```
- fun revstring s = implode (rev (explode s));
- val revstring = fn : string → string
```

We could define revstring using functional composition.

```
- fun revstring s = (implode o rev o explode) s;
```

It should be possible to simply say

```
- val revstring = implode o rev o explode;
```

The functions will be applied from right to left. If the rightmost function, explode, has signature ‘a → ‘b, then the next, rev, must have the signature ‘b → ‘c and the third, implode, ‘c → ‘d. The composite function has the signature

‘a → ‘d.

Overloading

A function is overloaded if it has different definitions under different types. Take the function +. + sometimes means integer addition and sometimes it means floating point addition. The two definitions are different. You the programmer cannot define overloaded functions.

7.1 Polymorphism

In a strongly typed language like ML the type of a function can be inferred from its expression alone. We do not have to include type declarations except when absolutely necessary as in the following function:

```
fun add x y = x + y;
```

which should be written

```
fun add x y : int = x + y; or as
fun add x y : real = x + y;
```

Some function signatures do not contain specific types. Instead they use type variables which take on concrete types only when the functions are applied. Such functions are said to have polymorphic types, or the functions are said to be polymorphic. An object is polymorphic if it has many (or multiple) forms; in our case multiple types. Let us look at some examples:

```
fun id x = x;
val id = fn : ‘a → ‘a
```

Here ‘a is a type variable and it can take on any type. ‘a is a polymorphic type and id is a polymorphic function.

```
fun pairself x = (x,x);
val pairself = fn : ‘a → ‘a * ‘a
fun fst (x, y) = x;
val fst = fn : ‘a * ‘b → ‘a
fun swap (x, y) = (y, x);
val swap = fn : ‘a * ‘b → ‘b * ‘a
```

Polymorphism is good because it simplifies programming. We do not have to restrict functions unnecessarily. The concept of a “generic” function is made possible by polymorphism.

7.2 User-defined types

In addition to the types mentioned so far, an ML programmer is allowed to introduce her own types. The simplest user-defined type is a type abbreviation.

```
type vec = real * real;
type student = { name : string,
                 idnum : int;
                 faculty : string};
```

A true type declaration defines a new type along with its constructors. This is done using a datatype declaration. Here are some examples:

```
datatype DAY = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype DAY = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
con Mon = Mon : DAY
con Tue = Tue : DAY
. . .
```

This is an enumeration type.

Define a new data type money

```
datatype money = None | cash of int | cheque of string * int;
datatype money = cash of int | cheque of string * int | None
con None = None : money
con cash = fn : int → money
con cheque = fn : (string * int) → money
```

Define function amount.

```
fun amount None = 0
    | amount (cash (mon)) = mon
    | amount (cheque ( _, mon)) = mon;

val amount = fn : money → int
```

7.3 Abstract data types

ML has a facility for defining abstract data types. These are data types or concrete types together with their own interface functions. A user of an abstract type can only use the interface provided. Let us implement a set as an abstract type in order to show how abstract types work.

```
abstype 'a SET = set of 'a list
with

val EmptySet = set ([])
fun addset el (set (l1)) = if mem l1 el then set l1
else set (el :: l1)

fun member (set l1) el = mem l1 el
fun union (set l1) (set l2) =
set (l1 @ filter (not o mem l1) l2)

end;


>type 'a  SET
>val EmptySet = - : 'a SET
>val addset = fn : "a → "a SET → "a SET
>val member = fn : "a SET → "a → bool
>val union = fn : "a SET → "a SET → "a SET
```

Assume that mem and filter are pre-defined functions. mem l el checks to see if element el is in list l. It may be defined as follows:

```
fun mem nil a = false
    | mem (x :: xs) a = a = x orelse mem xs a;
```

Function filter filters from a list those elements that satisfy the given predicate. It may be defined as follows:

```
fun filter p nil = nil
   | filter p (x :: xs) = if p x  then x :: filter p xs else filter p xs ;
```

We may use the abstract type ‘a SET as follows:

```
val set1 = addset 4 (addset 5 ( addset 6 EmptySet));
val set1 = - : int SET
val set2 = addset 9 (addset 3 ( addset 4 EmptySet));
val set2 = - : int SET
union set1 set2;
val it = - : int SET
member (addset “David” (addset “Tom” (addset “Pat” EmptySet))) “Tom”;
val it = true : bool
member (addset “David” (addset “Tom” (addset “Pat” EmptySet))) “Tim”;
val it = false : bool
addset (“me”, None, 1991) (addset (“Him”, cash(1000), 1990) (addset (“Her”, cheque(“big”, 10000), 1988)
EmptySet));
val it = - : (string * money * int) SET
```

7.4 Exercises

7.1 Write a function position that gives the position of a substring within another string.
7.2 Write a function bintodec that converts a binary number into its decimal equivalent.

7.3 Write a function dectobin that converts a decimal number into its binary equivalent.

7.4 A date is represented by a 3-tuple or triple (day, month, ear) of integers. Define a function age, which takes two dates, the birthdate of some individual and the current data, and returns the age of the individual in years and months.

8. Searching

The syllabus states that we should look at applications of functional programming in Artificial Intelligence. AI has traditionally been the most fertile area for functional programming. Nowadays, however, functional programming is beginning to claim its place alongside other programming paradigms. At any rate, logic programming (especially through Prolog) is a serious competitor especially in expert systems. It is true that the need for ML arose in theorem proving but it is no longer restricted to that application area.

I propose to look at the following applications

- search in game playing
- problem solving
- theorem proving
- expert systems

We start with search.

In the early days of Artificial Intelligence it was thought that many problems could be solved by simply searching the solution (or problem) space until the desired solution was found e.g. playing chess. This approach has now been abandoned. However, interesting results have been left behind and these can be usefully applied to specific problem areas.

Take the problem from [Winston, 1984] page 89.

Starting from S we would like to find a path to some other node on the diagram. If the nodes represent towns or cities, then we may be looking for a route from town S to another town. The numbers along the arcs could represent the cost of taking a particular path or route if we are interested in the best path and not just any path. This diagram represents a search problem.

Our first step is to convert the above diagram, which is a net, into a tree. The tree is more structured than the net and easier to work with.

There are two basic search strategies: breadth first and depth first. If we are looking for a path from S to E breadth-first search visits the nodes S, A, D, B, D, A, E in that order to return the path S, D, E. Depth-first search on the other hand visits the nodes S, A, B, C, E in that order to return the path S, A, B, E. Breadth-first search visits all the nodes at one level of the tree before proceeding to the next level. It returns the path containing the least number of nodes (the shallowest path). Depth-first search on the other hand performs a pre-order traversal of the tree and returns the leftmost path. This could be the shallowest path if we are lucky or the deepest if we are unfortunate.

Let us now look at these strategies more closely.

Depth-First Search

The subtrees below a node are visited from left to right. Each subtree is fully searched before its brother to the right. The tree will be represented as a list. Each node of the tree has two components:

1. the node label
2. the subtrees of the node - a finite list of trees.

We shall allow our trees to have infinite depth. They will, however, have finite breadth.

Since we cannot store infinite trees (and subtrees) explicitly we shall represent them as functions and generate them (or parts of them) whenever we require them. This technique is called lazy evaluation. It is useful for manipulating infinite structures. Lazy evaluation is based on the observation that infinite structures are never needed in their entirety. Rather, only finite parts of these infinite structures are needed from time to time. Lazy evaluation therefore uses a function to generate the required finite part of the infinite list. The node has type ‘a * (‘a → ‘a list) : ‘a is the label type and (‘a → ‘a list) is the type of the function that generates the list of subtrees. Let us use a stack to hold the nodes to be visited next. The top of the stack is the current node. Here is a function in Standard Ml from Paulson, page 179.

```
fun depthfirst next pred x =
          let fun dfs [] = Nil
| dfs(y::ys) = if pred y then Cons(y, fn()=> dfs(next y @ ys))
                  else dfs(next y @ ys)
     in  dfs [x]  end;
```

This function returns the solution path as a sequence of node labels. A sequence here, since it can be infinite, is actually a lazy list. It has the user-defined type

datatype ‘a seq = Nil | Cons of ‘a * (unit → ‘a seq);

unit is a type with no values. Remember that a type is a collection of values. Type unit has no values. A function whose argument type is unit produces something from nothing. (Confused??)

Breadth First Search

The implementation of breadth-first search is similar to that of depth-first search except that a queue is used in order to insert new nodes at the rear instead of at the front as with the stack.

```
fun breadthfirst next pred x =
        let fun bfs [] = Nil
              | bfs(y::ys) =
          if pred y then Cons(y, fn()=> bfs(ys @ next y))
        else bfs(ys @ next y)
    in  bfs [x]  end;
```

Let us now use both search techniques to generate a sequence of palindromes over the alphabet {A, B, C} (See Paulson page 180).

The tree for this problem is as follows.

The nodes emanating from the node labelled l in the tree can be generated by function nextlist :

fun nextlist l = [”A” :: l, “B” :: l, “C” :: l];

A palindrome is a list that equals its reverse.

fun ispalindrome l = l = rev l;

Let us look more closely at the functions breadthfirst and depthfirst.

Depthfirst takes three arguments

1. The lazy function
2. A predicate to test nodes to see if they satisfy the search condition
3. The root node.

This is what happens:

1. The search is initiated with the stack containing the root node.
2. If a node satisfies the predicate, it is appended to the initially empty solution sequence and the search continues with the subtrees of y.
3. If y does not satisfy the predicate, it is replaced on the stack by its successor subtrees.

If we execute

```
depthfirst nextlist ispalindrome [];
```

we get the sequence

```
Cons([], fn) of type string list seq
```

depthfirst nextlist ispalindrome [”B”]; gives Cons([”B”], fn)

How can we generate the sequences now represented by Cons([], fn) and Cons([”B”], fn) ?

Let us define a function to do the job. Paulson uses the name takeq for this function. You may, however, want to rename it to something like genseq for example.

genseq n xq returns the first n elements of the (possibly infinite) sequence xq.

```
fun genseq 0 xq = []
    | genseq n Nil = []
    | genseq n (Cons(x,xf)) = x :: genseq (n-1) (xf());
```

The higher order function mapq applies a function to every element of a sequence. It has the same functionality as map has on lists.

```
fun mapq f Nil  = Nil
    | mapq f (Cons(x,xf)) = Cons(f x, fn()=> mapq f (xf()));
```

The following runs are instructive. Remember that implode concatenates several strings into one.

1.

```
depthfirst nextlist ispalindrome [];
val it = Cons ([],fn) : string list seq
genseq 10 (mapq implode it);
val it = [””,”A”,”AA”,”AAA”,”AAAA”,”AAAAA”,”AAAAAA”,”AAAAAAA”,”AAAAAAAA”,”AAAAAAAAA”]
: string list
```

2.

```
breadthfirst nextlist ispalindrome [];
val it = Cons ([],fn) : string list seq
genseq 15  (mapq implode it);
val it = [””,”A”,”B”,”C”,”AA”,”BB”,”CC”,”AAA”,”ABA”,”ACA”,”BAB”,”BBB”,...] : string list
```

3.

```
breadthfirst nextlist ispalindrome [”B”];
val it = Cons ([”B”],fn) : string list seq
genseq 15 (mapq implode it);
val it =
[”B”,”BB”,”BAB”,”BBB”,”BCB”,”BAAB”,”BBBB”,”BCCB”,”BAAAB”,”BABAB”,”BACAB”,”BBABB”,...] :
string list
```

4.

```
depthfirst nextlist ispalindrome [”B”];
val it = Cons ([”B”],fn) : string list seq
```

This one runs for ever.

```
genseq 10 (mapq implode it);
```

The Eight Queens Problem

The problem requires that we arrange eight queens on a chessboard such that no queen may attack another. This means that the queens must be arranged in such a way that no two queens may share a row, column or diagonal. A board position will be represented as a list of row numbers. [r1, r2, r3, ..., r8] means that there is a queen in row r1 of column 1, row r2 of column 2, row r3 of column 3, ..., row r8 of column 8. Given this representation how can place a queen safely? We shall have to place it in a safe column, a safe row and a safe diagonal. If we place a queen in row newr of column 1 then a safe diagonal with respect to a queen already placed in row ri of column i is given by |newr - ri| <> i. Function safequeen tests whether a new queen at position newr in column 1 will be safe.

```
fun safequeen oldqs newq =
        let fun nodiag (i, []) = true
| nodiag (i, q::qs) =
                                                abs(newq-q)<>i andalso nodiag(i+1,qs)
    in  not (newq mem oldqs) andalso nodiag (1,oldqs)
    end;
```

Function nextqueen generates a list of safe positions with a new queen added.

```
fun nextqueen n qs =
        map (secr op:: qs)
(filter (safequeen qs) (upto 1 n));
```

Function upto generates integers in the closed range given by its arguments. Thus upto m n generates the closed range m .. n.

```
fun upto m n =

    if m>n then []  else  m :: upto (m+1) n;
```

Function secr enables function op:: to be mapped to a list containing its right arguments. Normally a function is mapped to the left.

```
fun secr f y x = f x y;

fun secl x f y = f x y;
```

A board is full if it has the maximum number of queens allowed.

```
fun isfull n qs = (length qs=n);
```

Let us try to solve the n queens problem with progressively larger values of n.

```
        depthfirst (nextqueen 1) (isfull 1) [];
        val it = Cons ([1],fn) : int list seq
        genseq 10 it;
        val it = [[1]] : int list list
        depthfirst (nextqueen 2) (isfull 2) [];
        val it = Nil : int list seq
        depthfirst (nextqueen 3) (isfull 3) [];
        val it = Nil : int list seq
        depthfirst (nextqueen 4) (isfull 4) [];
        val it = Cons ([3,1,4,2],fn) : int list seq
        genseq 10 it;
val it = [[3,1,4,2],[2,4,1,3]] : int list list
-

        depthfirst (nextqueen 5) (isfull 5) [];
        val it = Cons ([4,2,5,3,1],fn) : int list seq
        genseq 10 it;
        val it =
        [[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],[5,2,4,1,3],[1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],[3,1,4,2,5],[2,4,1,3,5]] : int
        list list
        depthfirst (nextqueen 6) (isfull 6) [];
        val it = Cons ([5,3,1,6,4,2],fn) : int list seq
        genseq 10 it;
        val it = [[5,3,1,6,4,2],[4,1,5,2,6,3],[3,6,2,5,1,4],[2,4,6,1,3,5]] : int list list

        depthfirst (nextqueen 7) (isfull 7) [];
        val it = Cons ([6,4,2,7,5,3,1],fn) : int list seq
        genseq 100 it;
        val it =
        [[6,4,2,7,5,3,1],[5,2,6,3,7,4,1],[4,7,3,6,2,5,1],[3,5,7,2,4,6,1],[6,3,5,7,1,4,2],[7,5,3,1,6,4,2],[6,3,7,4,1,5,2],[6,4,7,1,3,5,2]
        ,[6,3,1,4,7,5,2],[5,1,4,7,3,6,2],[4,6,1,3,5,7,2],[4,7,5,2,6,1,3],...] : int list list
        depthfirst (nextqueen 8) (isfull 8) [];
        val it = Cons ([4,2,7,3,6,8,5,1],fn) : int list seq
        genseq 100 it;
        val it =
        [[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1],[3,5,2,8,6,4,7,1],[3,6,4,2,8,5,7,1],[5,7,1,3,8,6,4,2],[4,6,8,3,1,7,5,2],[3,6,8,1,4,7,5,2],[
        5,3,8,4,7,1,6,2],[5,7,4,1,3,8,6,2],[4,1,5,8,6,3,7,2],[3,6,4,1,8,5,7,2],[4,7,5,3,1,6,8,2],...] : int list list
-
```

The application area of interest here is the set of two-person games like chess. These games are played as follows:

from a given initial position
two players move alternately
until no further moves are possible.

Either one of the two players wins or the game is drawn.

We need to define a position. A move generates possible new positions from a given position. Let us use a specific game to make the ideas more concrete. We shall use tic-tac-toe. A position may be represented by a list of three lists representing the three rows in the tic-tac-toe game. A 1 in a list position indicates that the square is occupied by player 1, a 2 indicates that player 2 occupies the square and a 0 indicates that the square is empty. Thus the position

is represented as [[1,0,0], [2,2,1], [1,0,2]].

Function move generates a tree with the current position as the root. Starting from the above position move generates the following tree.

To each of the four leaf positions there are several responses by player 1 and to each of these responses player 2 will have several responses and so on. In some cases the sequence of moves and counter moves can generate an infinite tree or a very large tree at least. Before going too far, let us make the tree more concrete.

A tree is defined as having a root node and a list of subtrees, which are themselves, trees. The following type definition is suitable for the tree.

```
datatype 'a Tree = Node of 'a  * ('a Tree) list;
```

Function gametree generates a tree from a given position p by applying the moves function to p.

```
fun gametree p = reptree moves p;
fun reptree f x  = Node (x, (map (reptree f) (f x)));
```

It is customary to prune large trees to a manageable depth.

```
fun prune 0 Node(x, ts) = Node(x, [])
    | prune (n+1) Node(x, ts) = Node(x, map (prune n) ts);
```

Thus the expression prune n t; will cut off all nodes further than n from the root of t.

Each position in the tree is evaluated (maybe using a rough estimate) to determine whether it is a winning position or a losing position with respect to the player whose turn it is to move next. A player will therefore choose a move, which leads to the best possible end position. The opposing player will probably do the same. Therefore, while one player is trying to maximise her chances of winning the other player is trying to minimise those chances. The tree nodes therefore have minimax values. I minimise my opponent's chances of winning while I maximise mine. Take the following tree, which has been pruned, to depth 2 by player 1:

The leaf nodes represent the end positions with a depth 2 pruning. The best of the end positions for player 1 has a value of 60. However, if player 1 follows this move player 2 can force her into the worst position of -35. The best that player 1 can guarantee herself is a position with value 25. 25 is the minimax value of the tree and is therefore the value of the current position. Each node in the tree is labelled with its minimax value. If position p has successor positions with values v1, v2, v3, ..., vn then the minimax value of p is given by max (-v1, -v2, -v3, ..., -vn) or equivalently by -min(v1, v2, v3, ..., vn).

```
fun minimax Node(x, []) = x
    | minimax Node(x, ts) = -min (map minimax ts);
```

Note that this function assumes that the tree is labelled with its minimax values.

9.3 Abstract data types : the queue

Abstract data types provide an abstract interface to their users. This means that users of these objects do not need to know how the objects are implemented: i.e. what concrete types are used or how the operations are achieved or what algorithms are used. We shall use the queue to show how abstraction can be achieved in Standard ML.

One of the more efficient implementations of queues uses two lists. The head of the first list is the front of the queue and the head of the second list is the rear of the queue. Thus the queue with elements q1, q2, q3, q4, q5 mat be represented by the two lists [q1, q2, q3] and [q5, q4]. Elements are added to the rear of the queue and removed from the front. With this implementation both actions will involve accessing the head of a list. This is done in constant time however long the lists are. There is a price though. We have to normalise the queue each time an element is added or removed. We shall soon see what all this means.

First of all we define the signatures (types) of the queue operations.

```
signature QUEUE =

sig

type 'a T
exception Empty
val empty : 'a T
val enqueue : 'a T → 'a → 'a T
val isEmpty : 'a T → bool
val front : 'a T → 'a
val dequeue : 'a T → 'a T
end
```

The general syntax of a signature is

The signature serves to specify the abstract data type. Only the names specified in the signature will be exported (visible) to the rest of the world.

Next we define a structure which will contain implementations of the operations specified in the signature.

```
structure queue : QUEUE =

    struct

    datatype 'a T = Queue of ('a list * 'a list)
    exception Empty
    val empty = Queue ([],[])
    fun normalise (Queue ([], tails)) = Queue (rev tails, [])
        | normalise q  = q
fun enqueue (Queue (f, r)) x = normalise (Queue (f, x :: r))
fun isEmpty (Queue ([], [])) = true
   | isEmpty _ = false
fun front Queue (x :: _, _) = x
| front (Queue ([], _)) = raise Empty
    fun dequeue (Queue (x :: heads, tails)) = normalise (Queue (heads, tails))

    | dequeue (Queue ([], _)) = raise Empty

    end
```

Let us now use the queue.

```
    val q = queue.Queue ([1,2,3], [5,4]);
queue.null q;                  Result : false.
queue.enqueue q 6;             Result : Queue([1,2,3], [6,5,4])
```

We note that each queue operator has to be prefixed by the structure name. If we open the structure we make all its items available through their simple names. The above becomes:

```
open queue
val q = Queue([1,2,3], [5,4]);
empty q;
enqueue q 6;
```

Continuing,

```
dequeue q;
front q;
dequeue q; dequeue it; dequeue it;        Result : Queue([4,5], [])
```

Signatures and structures are the constructs that Standard ML uses to implement modules. Structures are themselves used by functors to produce new structures. We do not have the time to learn all about modules in Standard ML. I hope these few examples will make an impression. Let us now turn to trees.

9.4 Another abstract data type : trees

The binary tree is simpler, so we start with that. Afterwards we shall look at the general tree. We start with a signature specifying the tree.

```
signature TREE =
        sig
type 'a T
      val count : 'a T → int              (* how many nodes in the tree *)
      val depth : 'a T → int              (* depth of the tree *)
      val depthFirst : 'a T → 'a list           (* depth first traversal *)
      val breadthFirst : 'a T → 'a list   (* breadth first traversal *)
    end
```

Next we implement the specified tree items:

```
structure Tree =
    struct
    datatype 'a T = Lf | Br of 'a * 'a T * 'a T
    fun count Lf = 0
      | count (Br(v, t1, t2)) = 1 + count t1 + count t2
    fun depth Lf = 0
      | depth (Br(v, t1, t2)) = 1 + max(depth t1, depth t2)
    fun depthFirst Lf = []
      | depthFirst (Br(v, t1, t2)) = v :: (depthFirst t1) @ (depthFirst t2)
    end
```

The tree

(PUT THE TREE HERE)

is implemented as

```
open Tree
val t = Br(10,
    Br(30,
    Br(45,
    Br(15, Lf, Lf),
    Br(65, Lf, Lf)
                ) ,
```

```
    Br(20,
    Br(35, Lf, Lf),
    Lf
                )
              ),
    Br(25,
    Br(40, Lf, Lf),
    Br(55, Lf, Lf)
                )
          );
```

count t;
depth t;
depthFirst t;

Now let us try the general tree. Let us think about the signature. Isn't it interesting to note that the signature for the general tree does not differ from the binary tree signature ?

Assignment :

1. Add a function to the abstract data type to perform a breadth first traversal of the binary tree.
2. Implement an abstract data type for the general tree.

10. Reasoning About Functional Programs

As part of the effort to deal with the software crisis efforts are being made to attempt to prove the correctness of a program. It is well known, as documented by Donald Knuth in one of his many publications, that testing a program only reveals the presence of errors. Testing a program does not tell us whether errors are absent. Obviously with current software technology it is much easier to test programs than to prove them correct. Some proponents of testing are even trying to convince the world that testing is enough. There is a simple fact though. Imperative programs are inherently difficult to reason about because the computations they perform have to be seen in terms of state transitions from one von Neumann machine state to another. It is simply tedious to describe a single state on the von Neumann machine. Functional programs give us more than a ray of hope. These programs involve no machine state. In fact one could say that functional programs are abstracted form the real machine. They are machine independent (or almost so.) They can be written as mathematical objects (by making some sacrifices) and therefore all the rigour, formality and precision of mathematical treatment can be meted out at them. This includes mathematical proof. Proving a functional program correct is mainly an exercise in applying principles of mathematical proof. There are no von Neumann machine states to worry about. One such principle is structural induction.

10.1 Mathematical induction is based on natural numbers.

Let us first look at mathematical induction. Proof by mathematical induction goes like this.

Prove that property P holds for all natural numbers.

1. Prove the base case i.e. P(0) - property P holds for 0.
2. Prove the induction step. Prove that if P(k) then P(k+1).

   (i) Assume that the property holds for an arbitrary natural number k i.e. P(k)
   (ii) Prove that property P holds for natural number k+1 i.e. prove that P(k+1)

Mathematical induction is based on natural numbers. It uses the fact that the set of natural numbers is defined as follows:

1. 0 is a natural number
2. If k is a natural number then so is k+1

10.2 Structural Induction generalises Mathematical Induction to data types

Recall (from chapter 7) that data types in Standard ML are declared (or defined) in a way similar to the definition to the definition we saw for natural numbers. They are defined using constructors. For example the predefined data type list may have been defined as

```
datatype 's list = nil | :: of 'a * 'a list
```

This datatype has a constant nil (like 0 for natural numbers) and a constructor :: (like + for natural numbers). It appears therefore that we could extend the principle of mathematical induction to apply to lists as well. A similar argument will extend the principle of mathematical induction to other datatypes in Standard ML. Such a generalisation of mathematical induction to these data types is called structural induction.

10.3 Structural Induction Over lists

To prove that property P holds for all lists we need to prove the following :

1. Base case

   Prove that P holds for nil i.e. P(nil)

2. Induction step

   Prove that if P(l) then P(x :: l) for all x of the appropriate type.

   i. Assume that P(l)
   ii. Prove that P(x :: l) is true if P(l) is true

We shall use a simple function to show that we can use structural induction to prove things about functional programs.

Given the following definition of append

```
fun append x [] = x
| append [] x = x
| append (x :: xs, ys) = x :: append (xs, ys)
```

Let us prove by structural induction that

append(append(a,b), c) = append(a, append(b,c))

Written using @ this would be

(a @ b) @ c = a @ (b @ c)

i.e. append over lists is associative (i.e. both left associative and right associative)

Proof by structural induction on a

1. Base case : prove that it holds for a = []

   append (append ([], b), c) = append([], append(b,c))

   ```
   append(append([], b), c)  = append (b, c)  definition of append - pattern 2
                             =  append([], append(b, c))      -do-
   ```

2. Inductive step on a

   Prove that if it holds for a = k, then it holds for a = (e :: k)

   (i) Assume that it holds for a = k
   append(append(k, b), c) = append(k, append(b, c))

   (ii) Prove that it holds for e :: k

   ```
   append(append(e :: k, b), c)  = append(e :: append(k, b), c)     -definition - pattern 3
                                 = e :: append(append(k, b), c)     - definition : pattern 3
                                 = e :: append(k, append(b, c))     - inductive hypothesis
                                 = append(e :: k, append(b, c))     - definition : pattern 3
   ```

Here is a collection of more proofs by induction in Standard ML:

1. Theorem 8, 9, 10 pp 200 - 202 [Paulson]
2. Recursion and induction over lists chapter 5 Bird & Waddler