
Volume 2

# Programming: Advanced topics and techniques

Bsc Computing and Information Systems

R. Hierons

2003

2910**212**

This guide was prepared for the University of London External Programme by:

R. Hierons, MA, PhD, Lecturer in Computer Science, Brunel University.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the author is unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

The External Programme
Publications Office
University of London
34 Tavistock Square
London WC1H 9EZ
United Kingdom.

Web site: www.londonexternal.ac.uk

Published by: University of London Press

Printed by: Central Printing Service, University of London, England

# Contents

# Introduction

## *Learning objectives*

After reading this chapter you should be able to:
- describe the concept of a programming paradigm
- write brief notes on declarative programming languages
- explain the structure of this unit.

## *Overview*

There are now literally hundreds of programming languages and many of them are very similar. Normally what makes two programming languages similar is the paradigm: the idea behind the language. The actual syntax may be different but the underlying principles are the same. Paradigms thus lie at the heart of the study of programming languages and once you understand a programming paradigm you will find it relatively simple to learn a new language from that paradigm.

This subject guide covers two programming paradigms: *functional programming* and *logic programming*. You will use two of the most widely used languages from these paradigms: *Standard ML* (a functional language) and *Prolog* (a logic language).

Functional and Logic languages are declarative: they are closer to describing the problem rather than how it is solved. With practice you will therefore find that, in many ways, they are more flexible and powerful than imperative languages such as C and Pascal and object-oriented languages such as Java. You will find that both paradigms allow you to produce much more general, powerful programs. For example, in Prolog, you will be able to produce code to append two lists and then use the same code to split up lists. When solving a problem, these languages will also allow you to write much shorter and more understandable programs that are, in many ways, more elegant than the corresponding programs written in imperative programming languages. These benefits help explain the wide use of functional and logic languages within artificial intelligence. This unit thus will help you if you take the final year Artificial Intelligence unit.

## *Book list*

### Essential reading

There are two course texts, one for each paradigm. The text for functional programming is:

Hansen M. R. and Rischel H. *Programming Using SML*. (Harlow: Addison-Wesley, 1999) [ISBN: 0-201-39820-6].

The text for logic programming is:

Bratko I. *PROLOG Programming for Artificial Intelligence*. (Harlow: Addison-Wesley, 2001) third edition [ISBN: 0-201-40375-7]

**Further reading**

The following are useful texts on functional programming. While Chris Reade's book does not cover the latest version of Standard ML (which only differs slightly from the previous version), he has provided updates for this on the Web.

Paulson L.C. *ML For the Working Programmer*. (Cambridge: Cambridge University Press, 1996) second edition [ISBN: 0-521-56543-X].

Reade C. *Elements of Functional Programming*. (Wokingham: Addison-Wesley, 1989) [ISBN: 0201129159].

For more information on logic programming you might see any one of the following texts.

Clocksin W.F. and Mellish C.S. *Programming in Prolog*. (Berlin: Springer-Verlag, 1994) fourth edition [ISBN: 3-540-58350-5].

Clocksin W.F. *Clause and Effect: Prolog Programming for the Working Programmer*. (Berlin: Springer-Verlag, 1997) [ISBN: 3-540-62971-8].

### *Internet sites*

At the time of writing this subject guide, there were many relevant internet sites. These provide general information on the programming paradigms and public-domain (i.e. free) versions of the programming languages. It is important that you acquire versions of both Standard ML and Prolog. A number of useful sites are listed below. However, the list is far from comprehensive - try finding your own sites. Also, the dynamic nature of the internet means that by the time you read this, some of these sites may have moved or may no longer exist.

The most useful site for Standard ML is probably the FAQ (frequently asked questions) page. This contains general information and many links to other useful sites. In particular, the FAQ has links to sites containing public domain versions of Standard ML such as New-Jersey ML, Edinburgh ML and Moscow ML. The FAQ is at:

http://www.faqs.org/faqs/meta-lang-faq

An introductory document on Standard ML (called *A Gentle Introduction to ML*) can be found at the following site. There should be a link from the FAQ to this site.

http://www.dcs.napier.ac.uk/course-notes/sml/manual.html

There is a general site on logic programming. This site includes information about Prolog including links to sites that contain public domain versions of Prolog including SWI-Prolog. This site is at:

http://www.afm.sbu.ac.uk/logic-prog/

A number of public domain implementations can also be reached from:

http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/prolog/0.html

### *Aims*

This unit provides an overview of two programming paradigms: functional and logic. By studying this unit, you will be able to use and compare these two programming paradigms. You will explore these paradigms through studying the programming languages Standard ML and Prolog respectively.

### *Objectives*

After studying this unit you will be able to:

- describe the concept of a programming paradigm
- comprehend and write programs written using languages from the functional and logic paradigms
- discuss the relative merits of the logic and functional paradigms.

### *Declarative programming and abstraction*

Declarative programming languages are said to be abstract: when using them, you provide less information, about how a problem is to be solved, than when using traditional imperative programming languages such as C. Instead, you work closer to the level of the problem. This higher level of abstraction leads to more concise and understandable programs. Since programs may be produced more quickly in declarative languages, some people have considered them to be appropriate for prototyping. Their use has, however, been largely restricted to specialised areas such as artificial intelligence and automated theorem proving.

In order to illustrate the power of abstraction, consider the problem of calculating the factorial of an integer. Remember that the factorial of *0* is *1* and the factorial of a positive integer *n* is *n*(n-1)*...*2*1*. Factorial is not defined for negative integers. Then the factorial function may be defined by two statements: factorial of *0* is *1* and if *n>0* then factorial of *n* is *n* multiplied by the factorial of *n-1* (since factorial *n = n*(n-1)*...*2*1 = n*((n-1)*...*2*1) = n*factorial(n-1)*). Thus, *factorial(0) = 1* and if *n>0*, *factorial(n)=n*factorial(n-1)*. Using Standard ML it is possible to define the function factorial in exactly this way by:

```
fun factorial(0) = 1 |
     factorial(n) = n*factorial(n-1);
```

Alternatively, it might be written in the following manner.

```
fun factorial(n) = if n=0 then 1
     else n*factorial(n-1);
```

Later you will learn how to interpret and write such code, and see many ways in which the flexibility of declarative languages can help you, but it should be clear that both of these pieces of code are almost identical to the standard definition of the factorial function. They are also simpler than the corresponding code in standard imperative programming languages. It is important to note that, throughout this subject guide, code will be written in the `courier new` font.

Both the logic and imperative paradigms avoid the notion of a stored state (through the values of variables) and thus, in the definition above, any use of the variable *n* elsewhere has no impact on this definition. This greatly aids understandability: in order to understand this definition it is not necessary to consider the whole program. This contrasts with the imperative paradigm, as exemplified by languages such as C and Pascal, in which commands either access or update the internal state. Thus, for example, the effect of the statement `y:=x+1;` depends upon the preceding statements since these determine the value of the state (in particular, `x`), at this point. In fact, in imperative languages the behaviour of a program is exactly defined in terms of this internal state.

### *How to use this guide*

This guide is split into two main sections, one for each programming paradigm. The section on Standard ML is rather longer than that on Prolog. This is not intended to reflect some relative significance of these languages: it is a consequence of there being more to say about Standard ML due to the fact that, unlike Prolog, it has a type system. Each of these sections is split into a number of chapters containing activities and ends with some sample exam questions. Each chapter starts with a brief summary of its contents, the corresponding learning outcomes, and recommended reading. It is important that you complete the recommended reading. You should also install versions

of Standard ML and Prolog and use these to experiment with the ideas contained in each chapter.

Most chapters contain exercises. Some of these are pen and paper exercises and do not require the use of a computer. Other exercises ask you to write code that you might then run. However, where you are asked to write code, you should initially try to solve the problem using pen and paper and only then type in your solution. This will help the development of your understanding of the languages. Most of the exercises have very natural solutions (this is one of the great strengths of declarative programming languages) so the programs are generally short and, with practice, you should find that often you get the program right first time.

### *The examination*

There is one three-hour examination for 2910212. This will contain two main sections, the second section corresponding to this guide (and the first corresponding to Volume 1). For this reason, the sample examination paper included at the end of this guide covers section B only. In the examination you will be asked to answer three questions from each of these sections (A and B). At least one of your answers must be on Prolog.

You may be asked to produce SML or Prolog code. Where this is the case you should start by thinking about the problem: 'what is required and how are the entities related?'. The expressiveness of these languages means that given a formal description of the relationships of the entities involved, it will be relatively simple to actually produce the code. It is worth noting that as well as the second part of a sample paper, this guide contains separate samples question on SML and Prolog along with advice as to how these questions might be approached.

### *Learning outcomes*

After studying this chapter and doing the relevant reading you should be able to:

- describe the concept of a programming paradigm
- write brief notes on declarative programming languages
- explain the structure of this unit.

# Chapter 1: Introduction to Functional Programming and Standard ML

## *Learning objectives*

After reading this chapter you should be able to:
- define the basic types in SML
- make SML evaluate simple expressions
- explain how SML evaluates expressions through the use of reduction rules
- give the step-by-step reduction of an expression
- explain the role of operator precedence in SML
- comprehend and write conditional statements in SML

## *Reading*

The material in this chapter is covered in Chapters 1 and 2 of:

Hansen M. R. and Rischel H. *Programming Using SML*. (Harlow: Addison-Wesley, 1999) [ISBN: 0-201-39820-6].

Naturally, these chapters include some material that will appear in later chapters of this subject guide.

## *Introduction*

This chapter will introduce you to some of the basics of Standard ML (SML), where ML stands for 'Meta-language'. You will learn how to get SML to evaluate expressions for you. You will also learn about some of the basic types that SML provides. More sophisticated and powerful types shall be introduced in later chapters. Finally, you will learn a little about how SML actually evaluates expressions for you. While these basics are not enough to allow you to do more than evaluate simple expressions, they are the foundations upon which more powerful and sophisticated ideas sit.

## *Preliminaries*

As mentioned in the introduction, Standard ML is a declarative programming language based on mathematical functions. Functions are a natural form of representation for programs since much of the behaviour of a program in a conventional language can be thought of as a function mapping certain inputs to certain outputs. For example, a program which joins two lists of names together can be thought of as a function which maps two lists on to a third, such that the third is the result of joining the first and the

second. Functions do not, however, naturally capture the notion of side-effects, such as screen updating. This is, for example, required for graphical user-interfaces. While it is possible to develop graphical user-interfaces using functional programming languages, this problem will not be considered within the module: instead it will focus upon the core ideas from this paradigm.

One of the first things you should do is install an implementation of SML. There are a number of public domain versions and information on how to get hold of these was given in Chapter 1. Implementations of SML are normally interactive: you type in expressions or definitions and the system responds. This allows you to experiment.

At its simplest level, an SML program consists of an expression. In turn, these expressions consist of *operations* (or *operators*), *operands*, and *punctuation marks*. An example of an SML expression is given below.

```
(113 + 6) - (2 + 85);
```

Here `113, 6, 2, 85` are operands; `+` and `-` are operations; and `(,)` and `;` are punctuation marks. You will be able to enter such expressions into SML and it will evaluate them for you. Later you will see that it is possible to define functions based upon the idea of evaluating expression.

### *Using Standard ML*

Start up SML. You should see a prompt that will indicate that the SML system is ready to receive input from you. The actual symbol(s) that represent the prompt depend upon the implementation you are using and so will not be given here. Once you have the prompt, you can type in an expression and SML will evaluate it. You must, however, remember to end your expression with a semi-colon and then press `return`. Try entering the following expression followed by `return`:

```
1+2+3;
```

The system should respond with something like:

```
val it = 6 : int
```

This says that the result of evaluating the expression is an integer (indicated by `int`) with value `6`. It is also saying that this value may be accessed by referring to the special variable `it`: `it` holds the value of the last expression evaluated. You could now use this value within an expression. Try entering the following expression:

```
it+4;
```

The system should respond with something like:

```
val it = 10 : int
```

There are two main kinds of operators in Standard ML: infix and prefix. Infix operations occur between their operands: the arithmetic operators used above were infix operators. Prefix operators precede their operands. An example of a prefix operator in SML is `abs`. The operator `abs` takes a number and returns the absolute value of this number. Thus, you might enter one of the following:

```
abs(3.0);
abs(4);
```

Sometimes you will wish to use the result of an evaluation in some later expression. In general it is not sufficient to use `it`, since this only allows you to refer to the result of evaluating one expression. Instead you can use constants to represent values. This is achieved through the use of `val`. The basic syntax for this is `val` followed by the constant name, followed by an equals sign, and then the expression. As usual, the statement is completed by a semi-colon and the word `val` is separated from the variable name by one or more spaces. Thus, for example, the following leads to the constant `five` representing the integer `5`:

```
val five = 2+3;
```

When this is entered the Standard ML system will respond with something like:

```
val five = 5 : int
```

which means that the constant `five` has the integer value `5`.

**Activities**

1. Get SML to evaluate the following expressions (don't forget to type a ';' before pressing enter) and note the answers:

a) `2 - 3 - 4`
b) `2 - (3 - 4)`
c) `100 div 4 div 5`
d) `100 div (4 div 5)`
e) `(3 + 4 - 5) = (3 - 5 + 4)`

Can you explain the values output?

2. Write an SML expression to calculate the temperature in Celsius corresponding to 98° F. The conversion rule is to subtract 32 from the temperature in Fahrenheit and then multiply the result by 5/9. Test your expression by typing it in with values for the temperature in Fahrenheit.

3. Write an SML expression to calculate the distance covered by a rocket in 24 hours if its initial speed is 100km/h and its acceleration is $50kmh^{-2}$. The distance covered in time t by a body moving in a straight line with initial velocity $u$ and acceleration $a$ is given by the expression $ut+1/2at^2$. Test your expression by typing it in with different values for u, a, and t.

## *The basic types*

Standard ML is a strongly typed language. This means that the types of all objects are determined before execution begins. There are a number of advantages and disadvantages to strong typing. The advantages include the ability of the type system to identify some errors at compile time. The main disadvantage is that it may reduce the flexibility of the language. Later we shall see a language, Prolog, that is not strongly typed.

There are a number of primitive types of values in SML. These types include integers, reals, booleans, characters, and strings. These shall be described below. The datatype of strings simply gives sequences of characters. The Standard ML names for these types are given in the table below.

| Type | Terminology in Standard ML |
|---|---|
| integer | `int` |
| reals | `real` |
| boolean | `bool` |
| characters | `char` |
| strings | `string` |

**Built-in types in Standard ML**

Naturally SML allows you to define your own types and has more complex types such as tuples, records and list. These will be described later. As you will have noticed, when Standard ML displays the result of evaluating an expression, it also displays the type of the result. The basic types of SML shall now be described.

### The integers

Integers are represented in expressions as sequences of digits. While it is normal to express negative integers by using the minus sign, in SML this symbol is used to express

subtraction only and instead the symbol ~ is used. Thus in order to represent the number -5 you should type in `~5`.

There are five standard arithmetic *dyadic* operators: `+, -, *, div, mod`. Dyadic operators have two operands. The operator `div` carries out integer division, and `mod` gives the remainder after integer division. Thus, for example, the following are SML expressions with their corresponding results:

```
10 div 3;
val it = 3 : int

10 mod 3;
val it = 1 : int
```

The operators `+, -,` and `*` take on their normal meaning. All these operators correspond to binary functions that map two integers to a third. In the functional notation of SML this mapping (between types) is written:

```
int*int -> int;
```

This means that it takes two integers (int*int) and returns an integer (the type on the right hand side of the arrow). Here, the * is used to combine arguments and -> is used to separate the type of the input from the type of the output. The symbol * is usually read as *and*.

In addition to these five dyadic operators and the *monadic* operator ~ (one operand), Standard ML has the monadic operator `abs` described earlier which yields the absolute value (magnitude) of a number. The type of `abs` is:

```
int -> int;
```

For example if you type in:

```
abs(~3);
```

the system will respond with:

```
val it = 3 : int
```

## The reals

Each real number in Standard ML can be represented by a sequence of digits, followed by a decimal point, followed by another sequence of digits, and possibly followed by the letter `E` followed by a sequence of digits. Without the letter `E` we simply get numbers such as 1.5, with the `E` we get a number that is multiplied by a power of `10`. Thus, for

example, `1.5E2` actually represents the number $1.5\times10^2$ which is equal to 1.5*100 and thus 150.

It is important to remember that the reals and the integers are separate types and so if a function acts on integers it cannot be applied to reals while if a function acts on reals it cannot be applied to integers. Thus, for example, Standard ML will not accept the expression `4.0 div 2.0` since `div` is defined to be a function that acts on the integers and `4.0` and `2.0` are reals. Similarly, it will not accept the expression `1+2.0`. There are some exceptions to this rule, such as <, and these will be discussed in Chapter 5 where Polymorphism and Overloading are described.

There are a number of standard functions that act on the reals. These include the basic infix functions `*`,`+`,`/`,`-` (all have type real*real->real) and some special monadic functions such as `sqrt`, `abs`, `sin`, `cos`, `exp`, and `ln`. Again, `-` is used for subtraction while ~ is used to input negative numbers. There are also the functions `real` and `floor` that allow us to convert between the reals and integers. Thus, for example:

```
real(3);
val it = 3.0 : real
```

and:

```
floor(4.53);
val it = 4 : int
```

## The booleans

The booleans represent truth-values. There are thus two boolean values: `true` and `false`. There are a number of comparison operators, such as <, and three logical operators: `andalso`, `orelse`, and `not`. The comparison operators map two numbers (either both are integers or both are reals) to a boolean. For example:

```
5 < 4;
val it = false : bool
```

The comparison operators are: =, <>, >, >=, <=, and <. The meaning of each of these should be clear except, possibly, <> which means 'is not equal to'.

The logical operators `andalso` and `orelse` have type:

bool*bool -> bool

The operator `andalso` takes two booleans and evaluates to `true` if, and only if, both are `true`. The operator `orelse` takes two booleans and evaluates to `true` if, and only if, one or more of these values is `true`. For example:

```
(3 > 2) orelse (3 <  2);
val it = true : bool
```

The logic operator `not` is monadic. It inverts or *negates* truth values, and is of type:

bool -> bool

For example:

```
not false;
val it = true : bool
```

## Characters

The type of characters in SML represents exactly what you might expect. Thus a character represents either a standard printable character (such as a letter) or a special control character. A printable character is represented by preceding it by #" and following it with ". Thus, for example:

```
#"a";
val it = #"a" : char
```

Naturally, non-printable characters cannot be entered in this way. Instead they are entered using escape sequences that start with the 'backslash' symbol \. For example:

1. \b (input using `#"\b"`) represents the backspace character;
2. \r (input using `#"\r"`) represents carriage return; and
3. \n (input using `#"\n"`) represents end of line.

For a list of escape sequences see Hansen and Rischel [1999], page 347. Since escape characters begin with the symbol \ it is necessary to have some way of representing \ itself. This is achieved through using `\\` for backslash. Similarly, the quote symbol itself is represented by the escape sequence `\"`.

The characters are ordered by their ASCII codes and thus may be compared using the standard order-based operators such as > and <. Thus, for example:

```
#"a" > #"b";
val it = false : bool

#"a" < #"f";
val it = true : bool
```

It is possible to move between characters and their ASCII value. The function `chr` gives the character corresponding to an integer while `ord` gives the integer corresponding to a character. For example:

```
chr(65);
val it = #"A" : char

ord(#"c");
val it = 99 : int
```

## Strings

A string is a sequence of characters. Where the characters are printable, it is possible to input a string by just surrounding the string with quotes in a similar manner to characters (the # symbol is not required). Thus, for example:

```
"abcd";
val it = "abcd" : string
```

A character might be seen to be similar to a string that has length 1: i.e. a string that contains only one element. In fact, SML has a function `str` for moving from a character to a string. Thus:

```
str(#"a");
val it = "a" : string
```

Since a string may contain several characters, there is no function for producing a character from a string. However, there is a function called `explode` that takes a string and returns the list of characters contained in it (you will learn about lists later). Thus:

```
explode("abcd");
val it = [#"a",#"b",#"c",#"d"] : char list
```

All of the arithmetic comparison operators also apply to strings as well as characters, and this is based upon their lexical ordering. It is worth briefly mentioning that this is an example of *overloading*: the operator names take on several types and for each type there is a corresponding operator. Overloading will be described in Chapter 5. The comparison operators have the additional types:

```
char*char -> bool
string*string -> bool
```

For example:

```
"abc" > "ab";
val it = true : bool
```

```
"AZ" < "BC";
val it = true : bool
```

Note that lexical ordering is equivalent to the ordering given by the ASCII values of characters so that, for example:

```
"A" < "B";
"a" < "b";
```

both evaluate to `true`. In particular, however:

```
"A" < "b";
```

evaluates to `true`, whereas:

```
"a" < "B";
```

evaluates to `false`. This is because all lower-case letters have higher ASCII values than upper-case letters. When comparing two strings of length greater than 1, the comparison is based on the first point where the two strings differ.

There are a number of standard in-built operators such as:

- `size` which determines the length of a string; and
- `^` which links together (concatenates) strings.

We therefore get the following:

```
size("abc");
val it = 3 : int

"ab"^"cd";
val it = "abcd" : string
```

**Activities**

Determine which of the following expressions are valid. For those that are valid, state the type of the result of evaluation. Check your answers by typing the expressions into SML.

a) `2 * 3 = 6`
b) `not (2 * 3 = 10)`
c) `(3 = 4) = true`
d) `1+3-2.1`
e) `53 > 12`
f) `1+1=0 orelse 3.4 = 2.1`
g) `"abd">"xr"`

### *Evaluating expressions*

We have seen that SML is capable of evaluating expressions. In fact, the behaviour of SML can exactly be seen in terms of evaluating expressions through the use of reduction rules. In this section we shall initially describe the main concepts that relate to reduction rules, and their use in evaluating an expression, and then describe the rules that determine the order in which parts of an expression are evaluated.

A *rewrite rule* is a rule that has a left-hand side and a right-hand side such that these two sides are equivalent. The two sides of the rule are separated by an arrow that indicates that the left-hand side of the rule can be replaced by the right hand side of the rule. Thus, for example, 1+0 → 1 is a rewrite rule for integers. This rule may be generalised to give x+0 → x (whatever the value of x the corresponding rule is correct).

In order to evaluate an expression we might apply a set of rewrite rules until no more rules can be applied. Thus, for example, using the rule given above and the rule 1+1 → 2 we see that (1+0)+1 rewrites to 1+1 and then to 2. In each case the rule was applied to a *redex*: a statement that can be reduced. Thus, in the first case the redex was 1+0 and in the second case the redex was 1+1. Rewriting continues until no rule can be applied and thus the result has been produced. When an expression cannot be rewritten any further it is said to be in *normal form*. Thus evaluation might be seen to be a process of applying rewrite rules (to redexes) until the expression is in normal form.

There is one further complication and this concerns termination. Consider the rewrite rule x+y → y+x for integers. This rule is valid but consider what happens if we apply it to the redex 1+2. This is rewritten to 2+1 but now the rule may be applied again. Thus we get 1+2 → 2+1 → 1+2 → and so on. Thus, if we are not careful, the repeated application of rewrite rules can lead to non-termination - clearly not a property we want! The solution to this problem is to insist that the rules, in some way, reduce the complexity of the expression. Rewrite rules that reduce the complexity are called *reduction rules*. Thus evaluation in SML actually proceeds by the repeated application of reduction rules to redexes until normal form (and thus the result) is achieved.

When rewriting expressions, SML will sometimes introduce temporary variables. In order to consider rewriting in the presence of such temporary variables, it is normal to use an environment that contains bindings to these variables. You will find this described in Chapter 2 of Hansen and Rischel [1999] if you are interested in learning more. However, you are not required to do so for this module.

You should now appreciate the role of redexes, reduction rules and normal forms in SML. There is, however, one further issue to be covered. This concerns the order in which reduction rules are applied. Consider, for example, the expression 1-2-2. Is this (1-2)-2 → ~ 1-2 → ~3 or 1-(2-2) → 1-0 → 1? The two orders of reduction lead to different answers, so it is important to be clear as to which is used and how SML decides this.

As in arithmetic, every infix SML operation is assigned a precedence and this precedence is represented by a number. SML evaluates operations with the highest precedence first. For example, an operator with precedence 6 is evaluated before an operator with precedence 4. A list of these values for some built-in SML operations is given below.

| Operation | Precedence |
|---|---|
| `/ * div mod` | 7 |
| `+ - ^` | 6 |
| `@ ::` | 5 |
| `= <> < > <= >=` | 4 |

**Some Standard ML operators and precedences**

So, for example, the expression:

```
2 + 3 * 4;
```

evaluates to `14` rather than `20` because `*` has a higher precedence than `+` and thus it is evaluated as if it had been written:

```
2 + (3 * 4);
```

Operations with equal precedence are normally evaluated from left to right. So the following expression:

```
9 - 5 + 3;
```

is treated as though it had been written:

```
(9 - 5) + 3;
```

The precedence of operations can be overridden by inserting brackets that force a part of the expression to be evaluated first. Thus the expression:

```
(2 + 3) * 4 ;
```

evaluates to `20` rather than `14`.

The priority of all monadic operators (those with one argument) is higher than that of dyadic ones (those with 2 arguments), so that, for example:

```
abs 0 - 3;
```

evaluates to `~3` since it is interpreted as:

```
(abs 0) - 3;
```

**Activities**

1. Determine which of the following expressions will evaluate correctly. For each that does not evaluate correctly, add brackets so that it does.

a) `1+2=3+4`
b) `~5-7`
c) `1<2=3<4`
d) `4 div 3 - 3`

2. For each of the above (corrected) expressions, give its type and give the steps involved in evaluating it. Note: you should give each reduction step involved and be careful about the order in which they are applied.

### *Conditional expressions*

Standard ML provides conditional expressions that evaluate to two possible results. Conditional expressions have the form:

```
if Expression1
     then Expression2
     else Expression3
```

`Expression1` must be of type bool and is known as the *predicate*. `Expression2` can be of any type, and is referred to as the *consequent*. `Expression3` must be of the same type as `Expression2`, and is called the *alternative*. The consequent and alternative must have the same type as otherwise the conditional expression would not have a well-defined type (there would be two possibilities). Here is an example of a conditional expression and how it would be evaluated in SML:

```
if (3 * 2) = (2 * 3)
     then "y"
     else "n";
val it = "y" : string
```

The whole of a conditional expression in Standard ML corresponds to a triadic operator with three arguments: the predicate, the consequent, and the alternative. This is known as a *distfix* (distributed fix) operator in analogy with prefix and infix.

There are only two rewrite rules for conditionals:

```
if true
     then consequent-expression
     else alternative-expression
```

rewrites to:

```
consequent-expression
```

and:

```
if false
     then consequent-expression
     else alternative-expression
```

rewrites to:

```
alternative-expression
```

This means that a conditional expression cannot be reduced (i.e. is not a redex) until its predicate has been reduced to either `true` or `false`. For example, the conditional expression:

```
if 4 = 4
     then 2 * 3
     else 3 * 3
```

has three redexes: `4=4`, `2*3`, and `3*3`. SML always evaluates the predicate first, giving:

```
if true
     then 2 * 3
     else 3 * 3
```

of which the whole expression is a redex, reducing to:

```
2 * 3
```

It is worth noting that in SML, a conditional statement must have both a consequent and an alternative and these must have the same type. This is because the expression, formed by the conditional, must evaluate to produce a value (of the appropriate type). Each expression in SML has an associated type and is reduced to a value from that type. This contrasts with imperative languages, such as C, where statements access or alter the

internal store. In imperative languages it is often acceptable to have a consequent only, since the conditional is simply a control-flow construct that determines which statements shall be executed.

**Activities**

Give the step-by-step process of evaluating the following expressions

a) `if 1+1>2 then 3+4 else 1-2;`
b) `if 1=3 then 1=1 else 2=3;`
c) `if 1<2 then if 1+1=2 then 7+1 else 2+1 else 5+2;`

## *Summary*

We have seen that it is possible to enter expressions into SML and it will evaluate these, returning the result of the evaluation. This evaluation is achieved through reduction rules being applied to redexes until the expression has been reduced to normal form.

Your use of expressions in SML is supported by a number of basic types such as the integers and the booleans. The range of expressions that can be used is also extended by the existence of conditional statements.

## *Learning outcomes*

After studying this chapter and completing the exercises and reading you should be able to:

- define the basic types in SML
- make SML evaluate simple expressions
- explain how SML evaluates expressions through the use of reduction rules
- give the step-by-step reduction of an expression
- explain the role of operator precedence in SML
- comprehend and write conditional statements in SML.

# Chapter 2: Defining functions in Standard ML

## *Learning objectives*

After reading this chapter you should be able to:
- comprehend and write recursive and non-recursive functions in SML
- give the step-by-step evaluation of a function application
- use and explain mutual recursion
- define and read infix functions
- comprehend and use the let statement
- explain the role of exceptions in programming
- read and write functions containing exceptions
- handle exceptions.

## *Reading*

Most of the material covered in this chapter is covered in Chapters 1 and 2 of Hansen and Rischel [1999]. However, mutual recursion first appears in Chapter 8 and the let construct first appears in Chapter 4. Exceptions are discussed in Chapter 7.

Hansen M. R. and Rischel H. *Programming Using SML*. (Harlow: Addison-Wesley, 1999) [ISBN: 0-201-39820-6].

## *Introduction*

This chapter will teach you how to define functions within Standard ML. Initially you will learn how to define simple non-recursive functions. You will then be introduced to writing recursive functions in SML and then to mutual recursion.

The sections dedicated to defining functions shall be followed by three that describe some useful constructs: one allows user-defined functions to be used as infix operators and the second allows local association of a name to the value of an expression, using the `let` statement. Finally, exceptions and their role in SML shall be described.

## *Non-recursive functions*

Part of the power of SML comes from its ability to apply a complex combination of operations repeatedly using formulae expressed as *functions*. You can combine these functions to create more complicated and sophisticated functions. To understand this, consider the sequence of operations necessary to convert 50$^{o}$F to $^{o}$C:

```
(50 - 32) * 5 div 9;
10 : int
```

Now, in order to convert 100°F to °C we could re-apply the operations:

```
(100 - 32) * 5 div 9;
37 : int
```

However, SML provides a much more powerful alternative. It is possible to define a formula to tell SML how to convert between Fahrenheit and Celsius. Such a formula is known as a *function.* A function uses symbols to replace the variable parts of the expression. A suitable function for the conversion is as follows:

```
(f - 32) * 5 div 9;
```

The Fahrenheit temperature is represented by the symbol named `f`. This is known as a *formal parameter*. The name of a formal parameter may contain letters, digits, primes and under-scores and should start with a letter. The name cannot be one of Standard ML's reserved words. For example, the variable name cannot be `val` or `chr`.

In order to tell SML to use this formula to convert from Fahrenheit to Celsius, it is necessary to define a new operation that shall be called *celsius*. The rules governing the choice of symbols and characters for operation names are the same as for formal parameters. This operation will apply the abstraction expression to an integer representing the Fahrenheit temperature, and produce another integer representing the equivalent Celsius temperature. For example, we would expect the following behaviour:

```
celsius 100;
val it = 37 : int
```

In order to define the function Celsius, we must give Standard ML a reduction rule that tells it how to reduce an expression that contains applications of this function:

```
fun celsius f = (f - 32) * 5 div 9;
```

The word `fun` tells Standard ML that a function definition is about to be given; this is followed by the function name applied to a pattern (in this case a single variable) followed by = and the expression that the function application should be rewritten to. The part of the equation before the = is known as the *left-hand side*, and the part after, the *right-hand side*. If you type this into your SML system you should get a response similar to:

```
val celsius = fn : int -> int;
```

This states that `celsius` is equal to a function with type `int->int`: SML has *determined the type* of your function. Whenever you type in an expression or function definition, SML will determine the type, returning an error if you have broken the type rules. This process of determining the type for you, is called *type inference*. The presence of type inference means that normally you do not have to provide information about the types of, for example, variables within your function definitions. However, it is good practice to determine the type of a function before you input the definition. This allows you to check that the type reported by SML is that expected.

Variables in SML are similar to variables in mathematics and rather different to variables in imperative programming languages. A variable in SML might be seen as a place-holder for a value and thus once a value is bound to a variable name, all instances of the variable name (within the scope) refer to this value. In contrast, in imperative languages a variable name refers to the value at a memory location and thus may be altered. For example, the statement x=x+1 makes sense in imperative languages but not in functional languages.

Despite SML using type inference, you must be careful to provide SML with enough information about the types. Consider, for example, the following function definition:

```
fun add(x,y) = x+y;
```

Here SML knows that both `x` and `y` are arguments of the operator + and thus can check the type of this operator. Unfortunately + is overloaded and allows two different types: int*int->int and real*real->real. Thus, all SML knows is that either `x` and `y` are both integers, or they are both reals. In earlier versions of SML, the above function definition would have led to an error. However, now SML assumes that where there is such ambiguity, the variables are integers. Thus SML would reply:

```
val add = fn: int*int->int;
```

Thus, the function `add` cannot be applied to real numbers. For example, `add(1.0,2.0)` would lead to a type error. If you wish to define such a function that can be applied to real numbers, you need to provide SML with enough information for it to deduce that the arguments are reals. The following definition suffices:

```
fun add_real(x:real,y) = x+y;
```

Here SML has been told that x is a real number. From this it may deduce that the operator + is taking on the type real*real->real and thus that y is a real. It thus responds to this definition with the following:

```
val add_real = fn: real*real->real;
```

Naturally, there are a number of ways of providing SML with this information. The following is one of the alternatives, in which the type of the result is stated:

```
fun add_real2(x,y) = x+y:real;
```

The above function definitions may be seen to be definitions of reduction rules. Thus, for example, the definition of `celsius` can be seen as saying that a term of the form `celsius(t)` (for any term `t`) may be reduced to the expression `(f-32)*5 div 9` with the actual parameter `t` substituted in for the formal parameter `f`. The rule can only be applied when `t` is in normal form. Where `t` is not in normal form, it is reduced to normal form before this reduction rule is applied. If `t` is in normal form, the reduction rule is applied and then further reductions may take place.

It is also important to note that prefix operators are given higher priority than the infix operators. Thus, for example, the expression:

```
Celsius 100 + 20
```

is interpreted as:

```
(Celsius 100) + 20
```

This is then reduced to normal form through the following sequence:

```
(Celsius 100) + 20 →
((100-32)*5 div 9) +20 →
(68*5 div 9) + 20 →
(340 div 9) + 20 →
37 + 20 →
57
```

While in all of the above definitions there was only one reduction rule, we may give more than one rule. An example is the definition of the following function:

```
fun zero 0 = true |
    zero n = false;
```

The symbol | tells Standard ML that there is another rule and effectively means *or*. When evaluating the application of this function, we can apply the first rule or the second rule. The rules define the reduction for different input patterns and thus the rule applied depends upon which pattern the input matches. Here, when evaluating `zero` applied to an integer term `t`, `t` is reduced to normal form and then one of two rules is applied:

1. If `t` reduces to `0` then the first rule is applied and the expression reduces to `true`.
2. If `t` reduces to an integer other than `0`, the second rule is applied and the expression evaluates to `false`.

Thus, for example:

```
zero(1+1-2) →
zero(2-2) →
zero(0) →
true
```

and:

```
zero(1*4) →
zero(4) →
false
```

Naturally, in SML you can define a function in terms of other functions. Suppose, for example, we have the following function definitions:

```
fun double(x) = x+x;

fun increase(x) = x+1;
```

We can define a function that takes an integer n and returns 2n+1 by:

```
fun doub_inc(x) = increase(double(x));
```

By defining functions in terms of others, it is possible to tackle complex problems in the standard reductionist manner, by dividing the problem into sub-problems that can be solved separately.

**Activities**

1. Define an SML function `mult` that takes two integers and returns their product (one multiplied by the other). Now produce a version of `mult` that takes two real numbers and returns a real number.

2. Define an SML function `distance` that takes the initial velocity of an object, its acceleration during the journey and the duration of the journey and returns the distance travelled. Note: you may wish to look through your solutions to the exercises in the previous chapter.

3. Write each of the following functions in SML, in each case giving the type of your function:

a) A function `square` that takes an integer and returns the square of the number. Thus, for example, `square(2)` should evaluate to `4` and `square(3)` should evaluate to `9`.

b) A function `is_factor` that takes two integers `x` and `y` and evaluates to `true` if and only if `x` divides exactly into `y`. Thus, for example, `is_factor(3,12)` should evaluate to `true` and `is_factor(3,10)` should evaluate to `false`.

c) A function `ordered` that takes a pair of integers and returns `true` if and only if the first integer is less than the second integer.

4. Using pattern matching (and thus more than one rule) define a function `not2` that takes a boolean argument and: if the argument is `true` the function application evaluates to `false`; otherwise it evaluates to `true`. Note: do not use any boolean operators.

### *Recursive functions*

Many algorithms, and thus programs, involve repetition of some process. In imperative programming languages this is often achieved through the use of looping constructs such as while loops. However, these looping constructs are based on some condition, on the values of one or more variables, that says when the program can leave the loop. This cannot be achieved in functional languages since, once a value has been bound to a variable, this value cannot be altered. Instead we use recursion: we define functions in terms of themselves.

Consider, for example, the factorial function. This is defined by: the factorial of 0 is 1; otherwise if n>0 then the factorial of n is n*(n-1)*(n-2)*…*2*1. Thus, given an integer n>0, the factorial of n may be written as n*((n-1)*(n-2)*…*2*1) which is n multiplied by the factorial of (n-1). This leads to the following recursive definition

1. factorial(0) = 1
2. if n>0 then factorial(n) = n*factorial(n-1)

The first part of the definition does not involve recursion and may be called the *base case*. The second part is the *recursive case*.

This may be written, in SML, in at least two ways. The first way, in which there is a condition that checks whether the argument is zero, is:

```
fun factorial(n) = if n=0 then 1 else
     n*factorial(n-1);
```

Alternatively, we can give two separate reduction rules, producing the definition:

```
fun factorial(0) = 1 |
     factorial(n) = n*factorial(n-1);
```

It is important to note here that the pattern `n`, used in the second rule, includes the pattern `0` from the first rule. Thus, if the order of these rules was to be reversed, the recursive rule would always be used and the base case 0 would never be applied. Clearly this would be incorrect and thus the order of these rules should not be reversed.

Consider the expression:

```
factorial(2);
```

Here the actual parameter is `2` and the expression matches the second rule. Therefore this is rewritten to:

```
2*factorial(2-1)
```

which in turn is rewritten to:

```
2*factorial(1)
```

We can again apply the second rule for factorial rewriting this to:

```
2*(1*factorial(1-1))
```

which is rewritten to:

```
2*(1*factorial(0))
```

Now we apply the first rule to rewrite `factorial(0)` to `1` and get:

```
2*(1*1)
```

This is rewritten by two more steps to the normal form `2`, which is the result. The process involved in this evaluation should be clear: the argument is reduced by each application of a reduction rule for `factorial`, the process stopping once the base case `0` has been reached.

It is important to note the use of brackets around a term that has been produced by reducing a function application. Thus, for example, `2*factorial(1)` reduces to `2*(1*factorial(1-1))` not `2*1*factorial(1-1)`. The inclusion of these brackets makes explicit the fact that `1*factorial(1-1)` must be fully reduced **before** it is combined with the value `2`.

Interestingly, the definitions of factorial given above do not explicitly consider the case where the argument is negative. However, the function may be applied to negative numbers. Consider, for example, `factorial(~1)`. This is reduced in the following way:

```
factorial(~1) →
~1*factorial(~1-1) →
~1*factorial(~2) →
~1*(~2*factorial(~2-1)) → ...
```

You can see that, as before, the value of the actual argument reduces. This time, however, this reduction cannot lead to the base case (0) and thus the reduction does not terminate. Later we shall see how exceptions may be introduced to avoid this sort of problem.

Suppose, now, that we wish to define a function that, when given a non-negative integer n, evaluates to the sum of all the integers up to and including n. Then we might define this by:

1. sum(0) = 0
2. if n>0 sum(n) = n+(n-1)+…+2+1

The second rule might be written recursively by noting that (n-1)+…+2+1 is the same as sum(n-1). Thus, we get the following rules:

1. sum(0) = 0
2. if n>0 sum(n) = n+sum(n-1)

This leads to the following SML function definition:

```
fun sum(0) = 0 |
      sum(n) = n + sum(n-1);
```

Again it is possible to give the step-by-step reduction of this function applied to a value. The following is such a sequence of reduction steps:

```
sum(3) →
3+(sum(3-1)) →
3+(sum(2)) →
3+(2+(sum(2-1))) →
3+(2+(sum(1))) →
3+(2+(1+sum(1-1)))) →
3+(2+(1+(sum(0)))) →
3+(2+(1+0)) →
3+(2+1) →
3+3 →
6
```

**Activities**

The following questions ask you to define functions in SML. However, you should define these functions before entering them into SML. Once you have defined them you should enter them and then test thoroughly.

1. Define a function `p2` that takes an integer `n` and returns `2` raised to the power of `n`. Thus, for example, `p2(2)` evaluates to 4 and `p2(3)` evaluates to `8`.

2. Write a function `sum_cubes` that takes a positive integer `n` and returns the value of the sum of the cubes of the integers from `0` to `n`. Thus, for example, `sum_cubes(0)` would evaluate to `0` and `sum_cubes(3)` would evaluate to `36` (=1+8+27).

3. Define a function `power` that takes two integers `m` and `n` (`n` is *non-negative*) and returns `m` raised to the power of `n`. Thus, for example, `power(3,2)` would evaluate to `9`.

4. Having defined power, give the step-by-step evaluation of the following expressions:
   ```
   power(1,2)
   power(4,3)
   power(2,-1)
   ```

### *Mutual recursion*

We have seen how we can define a function in terms of itself and how to define a function in terms of other functions. Suppose, however, that we wish to define two functions f1 and f2, and to define f1 in terms of f2, and f2 in terms of f1. The techniques

introduced so far do not allow this: if we first define f1, then we get an error message, since f1 is defined in terms of f2, and this has yet to be defined. We get similar problems if we define f2 before f1. Instead we need to use mutual recursion: two functions are simultaneously recursively defined. In SML this is achieved by writing one function definition, finishing it with the keyword `and`, writing the other function definition (without the keyword `fun`) and finally finishing the definition with a semi-colon.

Consider, for example, the problem of defining the following functions in SML:

1. a function `even` that takes a non-negative integer and returns `true` if and only if that integer is even;
2. a function `odd` that takes a non-negative integer and returns `true` if and only if that integer is odd.

One way of defining these is to say that given integer n:

1. n is even if n-1 is odd;
2. n is odd if n-1 is even.

Thus, we can define `even` in terms of `odd`, and `odd` in terms of `even`. This may be defined using:

```
fun even(0) = true |
    even(n) = odd(n-1)
    and
    odd(0) = false |
    odd(n) = even(n-1);
```

While there are more efficient ways of defining the functions `odd` and `even`, there are situations in which mutual recursion is extremely useful. For example, as noted in Hansen and Rischel [1999], a file system can be described using a mutually recursive datatype:

1. a *file* is either a catalogue with a name and contents or a file with a name,
2. *contents* is a list of files.

Mutual recursion happens since the types that represent files and contents are defined in terms of one another. Since the datatype is mutually recursive, functions that operate on a file system are likely to be mutually recursive.

### *Making a function infix*

When you apply a function that you have defined, normally the expression is of the form of the function name followed by its arguments. This order is prefix and contrasts with infix operators such as + and * that lie between their arguments.

It is possible to tell SML to treat one of your functions as infix. This might be useful where the nature of the function makes it more natural for it to lie between its arguments. In order to make a function infix it is sufficient to type in `infix` followed by the function name. For example, the following defines a function `add` and makes it infix:

```
fun add(x,y) = x+y;
infix add;
```

You should now apply `add` as an infix function:

```
1 add 2;
val it = 3: int
```

Suppose we have an infix function `f` and an expression of the form `x f y f z` for arguments `x`, `y`, and `z`. Then this might be interpreted in either one of two ways: `(x f y) f z` or `x f (y f z)`. The first case is *associate to the left*: the left-most instance of the operator is evaluated first. The second case is *associate to the right*: the right-most instance of the operator is evaluated first. Normally we use association to the left and this is what infix provides. If, however, you want to define an infix function with association to the right you should instead use `infixr`. For example, the following defines an infix function `sub` that associates to the right:

```
fun sub(x,y) = x-y;
infixr sub;
```

Then we get the following behaviour:

```
3 - 2 - 1;
val it = 0 : int

3 sub 2 sub 1;
val it = 2 : int
```

A further complication occurs where a user-defined infix function is combined with other infix functions. You might recall that in SML, this problem is resolved for standard functions through the use of precedence. Similarly, every user-defined infix function has a precedence. Unless you give the precedence of the infix operator you are defining, it has precedence 0. Thus, for example, since `sub` has precedence 0 the following happens:

```
1 sub 1 - 1;
val it = 1 : int

1 - 1 sub 1;
val it = ~1 : int
```

If you wish to give your infix operator a precedence other than 0, you state this when you tell SML that the function is to be infix - the precedence lies between the keyword `infix` and the function name. For example, the following defines a new infix function `sub2` with precedence 8:

```
fun sub2(x,y) = x-y;
infix 8 sub2;
```

Then we get the following behaviour:

```
1 sub2 1 - 1;
val it = ~1 : int

1 - 1 sub2 1;
val it = 1 : int
```

**Activities**

1. Define your own versions of logical conjunction (and) and disjunction (or) operators, calling them `new_and` and `new_or` respectively, and make these infix.

2. Experiment with these new operators, evaluating expressions including the following:

a) `true new_and false`
b) `true new_or (false new_or true)`

3. Now define an infix operator `imp` defined by: `x imp y` evaluates to `true` if, and only if, either `y` is `true` or `x` is `false`.

4. Define infix functions `divl` and `divr` that each take integers `x` and `y` and returns `x div y` but differ through `divl` associating to the left, and `divr` associating to the right.

5. Determine the results of evaluating the following expressions. When you are confident that you know the results, and can explain them, check your answers by typing the expressions into SML

a) `29 divl 4 divl 2` and `29 divr 4 divr 2`
b) `10 divl 2 divl 5` and `10 divr 2 divr 5`

### *The `let` statement*

Sometimes we wish to evaluate a sub-expression a number of times within an expression. Instead of having separate evaluations, we can evaluate the sub-expression, give the result a name, and refer to the name within our function. This is done by using the `let` statement, whose syntax is:

```
let Def in Expression end
```

Here `Def` is a sequence of definitions of names, each in the form of `val name = exp`, and `Expression` is the expression to be evaluated. Where there is more than one definition, these definitions are separated by the keyword `and`. The definitions in `Def` are used within `Expression` and essentially define the values of temporary variables to be used within `Expression`. The scope of these temporary variables is exactly `Expression`. Consider, for example, the following function definition:

```
fun f1(n) = let val x=n+1 and y=n+2 in x+y end;
```

Suppose the expression `f1(4)` is to be evaluated. Then this is initially reduced to:

```
let val x=4+1 and y=4+2 in x+y end
```

Essentially this expression is `x+y` in the environment in which `x` takes on the value `5` and `y` takes on the value `6`. Thus this expression reduces to `11`.

The use of a `let` statement can aid the understandability of code. It can also improve efficiency where a sub-expression is used more than once within an expression. A temporary variable can hold the value produced by reducing the sub-expression and this variable can be referred to within the expression. If this is done, the sub-expression is only reduced once.

In the following example (from Paulson [1996]) the function `findroot` applies the Newton-Raphson method to find approximate square roots:

```
fun findroot(a,x,acc)=
     let val nextx = (a/x+x)/2.0
     in if abs(x-nextx) < acc*x
          then nextx else findroot(a,nextx,acc)
     end;
```

Here `nextx` represents the next approximation and is used several times in the function definition.

**Activity**

Using `let`, and the function `power` produced earlier, define a function that takes an integer argument `x` and returns the fourth power of `x`, if the fourth power of `x` is less than `1000` and otherwise returns half of the fourth power of `x`.

## *Exceptions*

Exceptions usually represent conditions that represent errors of some type. One possible source is your entering an expression that should lead to an error. For example, you might type in the following:

```
1 div 0;
```

SML will respond with an error message that mentions the existence of an 'uncaught exception'. It should also tell you something about the nature of the exception, probably stating that it is division by zero. It has told you that the exception is uncaught: this means that SML has not been told what to do when this exception occurs.

In the next section you will meet lists. Some functions are not defined on the empty list, an example being a function that returns the first element of a list: the empty list has no first element. If we apply such a function to the empty list we get another exception.

We can define our own exceptions and then define functions that are capable of producing (or raising) these exceptions. The general form of the definition is:

```
exception Exception_name;
```

Suppose, for example, we are defining functions (such as `factorial`) that are not defined on negative integers. Then we might create a special exception to allow us to report the application of such a function to a negative number. This is achieved using:

```
exception negative;
```

Then your functions can now raise this exception using an expression of the form `raise negative`. The following does this for `factorial`.

```
fun fact n = if n<0 then raise negative
     else if n = 0 then 1 else n*fact(n-1);
```

This leads to the following behaviour:

```
fact(4);
val it = 24 : int

fact(0);
val it = 1 : int

fact(~3);
uncaught exception negative
```

Importantly, exceptions move out through expressions until they are either caught (or handled) or they return as a result. They cannot 'disappear'. Thus, we get the following behaviour:

```
fact(~3)+1;
uncaught exception negative
```

As well as raising exceptions, we can handle them. Exception handling involves part of a function specifying what to do if a certain exception is raised. We achieve this by following an expression by the word `handle` and then one or more patterns. These patterns are often of the form of exception names.

Based on our earlier definition of `fact`, we can produce the following:

```
fun fact1 n = fact n handle negative => ~100;
```

If `fact1` is applied to an integer it applies `fact` to that integer. If `fact` returns a value, this is the result of `fact1`. However, if fact returns the exception negative, `fact1` returns the value `~100`.

An exception might take an argument (or parameter). This allows an exception that is raised to carry further information through parameters. If the argument has type `t` the syntax for defining this exception is:

```
exception Exception_name of t;
```

We might, for example, wish to be able to associate a value with an exception like negative. The following achieves this:

```
exception neg2 of int;
```

Then we can now define a version of factorial that not only can return an exception that indicates it has been given a negative value, it also can state what this value was.

```
fun factv n = if n<0 then raise neg2(n)
     else if n = 0 then 1 else n*factv(n-1);
```

We might then produce the following version of a factorial function that uses this information:

```
fun fact1 n = factv(n) handle neg2(n) => n;
```

The examples we have given here probably seem rather artificial. This is partly because we are only dealing with small programs. It is also because we are not dealing with complex datastructures. In particular, if you define your own datastructures you are likely to have partial functions (functions that are not defined for all values) and are likely to want exceptions for these.

**Activity**
Define a function `divide` that takes two integers `x` and `y` and returns `x div y` if `y` is non-zero and otherwise returns a new exception `zero`. Now use this function `divide` and, by catching the exception, define a function `calculate` that takes integer z and:

- if z+2 is not 0 then it returns a pair whose first element is the string `"ok"` and second element is `100 div (z+2)`.
- if z+2 is 0 then it returns a pair whose first element is the string 'div by zero' and second element is `0`.

## *Summary*

We have seen how functions can be defined, relatively simply and concisely, in SML. In SML functions are usually recursive: the function is defined in terms of itself. This can be extended to allow mutual recursion: two or more functions are defined in terms of one another.

Functions that take two arguments can be infix and here you can choose whether the operator associates to the left or the right. It is also possible to associate with an infix function a priority in order to determine the order of evaluation of expressions containing this function.

In order to aid the definition and use of functions, two new notions have been introduced: the `let` statement and exceptions. The `let` statement allows an expression to be evaluated and given a temporary name. This name can then be used within some other expression. The `let` statement can be used to improve efficiency. Exceptions may be introduced to allow a function to raise certain error conditions. It is also possible to handle exceptions when they occur.

### *Learning outcomes*

After studying this chapter and completing the exercises and reading you should be able to:

- comprehend and write recursive and non-recursive functions in SML
- give the step-by-step evaluation of a function application
- use and explain mutual recursion
- define and read infix functions
- comprehend and use the let statement
- explain the role of exceptions in programming
- read and write functions containing exceptions
- handle exceptions.

# Chapter 3: Lists, tuples and records

## *Learning objectives*

After reading this chapter you should be able to:
- explain SML code that includes tuples, records, and lists
- write SML code that uses or manipulates tuples, records, and lists
- define a list in terms of `::` and `nil`;

## *Reading*

The material in this chapter is covered in Chapters 3 and 5 of Hansen and Rischel [1999].

Hansen M. R. and Rischel H. *Programming Using SML*. (Harlow: Addison-Wesley, 1999) [ISBN: 0-201-39820-6].

## *Introduction*

This chapter will consider the definition and use of three general datatypes in SML: tuples, records and lists. By the end of the chapter you should be able to interpret the use of these datatypes, appreciate when they might be used and write SML functions that manipulate or produce elements of these types.

## *Tuples*

Tuples should be familiar. A tuple is simply a finite sequence of values, surrounded by brackets, with the values being separated by commas. You will already have seen one type of tuple, the 2-tuple, which is generally called a pair. In general, a tuple containing n elements is called an n-tuple. Thus, for example, a tuple containing 4 elements is a 4-tuple.

There is no restriction on the types of the elements in a tuple. Thus, for example, `(1,2)` and `(#"a",3.6,33,4)` are tuples. In general, a tuple is of the form $(v_1,v_2,\ldots,v_n)$ where each one of $v_1,\ldots,v_n$ are SML terms.

Every tuple has a type that is determined by the types of the elements contained within the tuple. In general, a tuple $(v_1,v_2,\ldots,v_n)$ in which $v_i$ has type $t_i$ $(1\leq i\leq n)$ has type $t_1*t_2*\ldots*t_n$. Thus, for example, the tuple `(1,2)` has type int*int and the tuple `(#"a",3.6,33,4)` has type char*real*int*int.

It is important to note here that you cannot 'expand out' type definitions. Thus, for example (int*int)*(int*int) is not the same as int*int*int*int. To see this we can compare two elements, ((1,2),(3,4)) and (1,2,3,4) of these two types. While the elements contain the same values 1,2,3 and 4, they have different structures and so are not the same.

A tuple can contain any finite number of elements *except* one: there is no such thing as a tuple with one element. The tuple with no elements can be input as `()` and has a special type: `unit`. Thus:

```
();
val it = (): unit
```

Functions can take or return tuples. Where before, functions seem to have taken more than one argument, they have in fact normally taken only one argument which is a tuple containing two or more values. Thus, for example, the following function takes a pair of integers and returns a pair of integers:

```
fun add_diff(x,y) = (x+y,x-y);
val add_diff = fn: int*int -> int*int
```

There are functions that allow you to pull elements out of a tuple. These functions are of the form of the `#` symbol followed by an integer. The integer specifies the position of the value to be returned. Thus, for example, `#1(3,true,false)` evaluates to `3` and `#2(3,true,false)` evaluates to `true`.

**Activities**

1. What are the types of the following tuples?
a) `(4.35,#"d","ss")`
b) `(1,1,1,1)`
c) `()`

2. Write an SML function `swap` that takes a pair and returns the pair with the order of the elements reversed. What is the type of `swap`?

3. Define a function `order` that takes a pair of integers and evaluates to an ordered pair containing these values. Thus, for example `order(1,2)` will evaluate to `(1,2)` and `order(2,1)` will evaluate to `(1,2)`. What is the type of `order`?

## *Records*

A record is a collection of values in which each value is associated with some unique name. The names thus describe the different components of the record and the names are used to access values from a record or to define a record.

In order to define a record it is sufficient to list the names and associated values, the names and values being separated by the equality sign and the list being surrounded by set brackets. For example, the following defines a record that contains three values, one associated with `firstname`, one associated with `secondname` and one associated with `age`:

```
val rec1 = {age = 24, firstname = "joe",
      secondname = "bloggs"};

val rec1 = { age = 24, firstname = "joe",
      secondname = "bloggs"}: {age:int,
      firstname: string, secondname: string}
```

You can see from this that the type is given by listing the names, with the types of the associated values. An important property of records is that the order in which the values are given is irrelevant: the record is defined by the mappings from names to values not the order in which these mappings are listed. Thus, for example, the records `{age = 24, name = "joe"}` and `{name = "joe", age = 24}` are equivalent since they define the same mapping:

```
{age = 24, name = "joe"} = {name = "joe", age =
      24};
val it = true: bool
```

It is possible to split up a record by comparing it to a record containing variables. The variables then take on the values from the record. For example, the following extracts the values from the record `rec1`:

```
val {age=x, firstname=y, secondname=z} = rec1;
```

This results in:

```
val x = 24: int
val y = "joe": string
val z = "bloggs": string
```

It is also possible to determine the value associated with a particular name in a record using the selector symbol (`#`) followed by the name. Thus, for example, the following determines the value associated with `age` in `rec1`:

```
#age rec1;
val it = 24: int
```

The similarity between this and the notation used to extract values from tuples is no co-incidence: a tuple is a special type of record in which the names are consecutive integers starting at 1. Thus, for example, the following defines a 4-tuple:

```
{1 = 3.2, 2=12, 3="cat", 4=true};
val it = (3.2,12,"cat,true):
     real*int*string*bool
```

It is possible to define a record type using the construct `type`. The following, for example, defines a type `person`:

```
type person = {age: int, firstname:
     string, secondname: string};
```

In fact, the construct `type` can be used to define other types. The following, for example, defines the type of pairs of integers:

```
type int_pair = int*int;
```

When defining functions to be applied to records, it is vital to define record types. This is because SML must be given enough information to be able to determine the type of the function. Where a function is defined using the selector symbol and the record type is not stated, SML cannot determine the complete type of the record to which the selector is to be applied. For example, the following leads to an error:

```
fun age(x) = #age x;
```

Instead it is possible to write the following:

```
fun age(x:person) = #age x;
val age = fn: person -> int;
```

**Activities**

1. Define a record type `car` that holds the following pieces of information:
a) the 'model';
b) the company that manufactures the car;
c) the engine size; and
d) the number of doors.

2. Define the following functions:
a) a function `doors` that takes a record of type `car` and returns the number of doors;
b) a function `greater` that takes two records `r1` and `r2` of type `car` and evaluates to `true` if and only if the engine size of `r1` is greater than that of `r2`;
c) a function `makes` that takes a string `x` and a record `r` of type `car` and returns true if and only if `x` is the name of the company that makes `r`.

### *Lists*

A list is a finite sequence of values. The list is one of the most important structures in functional programming and thus is built into Standard ML. SML does, however, place one important restriction on lists: the elements of a list must all have the same type.

In order to input a list of values, it is sufficient to write the sequence of elements, separating the elements by commas, and surrounding the sequence using the `[` and `]` symbols. For example, `[1,3,2]` is a list of integers that contains the values `1`, `3`, and `2`. The order of elements is important in lists and thus `[1,3,2]` and `[1,2,3]` are different lists:

```
[1,3,2] = [1,2,3];
val it = false: bool
```

While it is possible to input a list in the manner described above, this representation is not sufficient to define arbitrary functions. This is because, in general, the number of elements in a list is not fixed. Consider, for example, the problem of defining a function that takes a list of integers and returns the list with each value increased by 1. The following defines this for lists of length 1:

```
fun incl1([x]) = [x+1];
```

However, this will not work on an integer list whose length is not 1. If we wish to define a general function that will take an integer list of any length, we need to apply some other approach.

The type of lists may be defined recursively using the fact that every list is one of the following:
1. the empty list;
2. a list with a head (the first element) and a tail (the remaining list once the head is removed).

The following gives some lists and their corresponding heads and tails.

| **list** | **Head** | **tail** |
|---|---|---|
| [1] | 1 | [] |
| [1,2,3] | 1 | [2,3] |
| [] | not defined | not defined |

**Above: the head and tail of lists**

The underlying definition of lists, in SML, is based on the rule: a list is either empty or it has a head and a tail. The constructors (special operators used to build and define the type) used are `nil` for the empty list and `::` for the operator (often called *cons*) that adds an element to the front of a list. Thus, the empty list is represented as `nil` (we can also

use `[]`) and a list with head `h` and tail `t` is represented as `h::t`. From the table you can see that the shortest lists that can be represented using head and tail are those of length 1. These are the lists with empty tails.

Any list can be represented in terms of the list constructors. For example, we can represent the list `[1,2]` as `1::(2::nil)`. In order to see this you can first of all consider the list `[1,2]` which has head `1` and tail `[2]`. Thus `[1,2]` can be written as `1::[2]`. Now consider the list `[2]`. This has head `2` and tail `nil` and thus may be written as `2::nil`. Thus `[1,2]` is `1::[2]` which in turn is `1::(2::nil)`. The following table gives some lists and their representation in terms of list constructors.

| **List** | **representation** |
|---|---|
| ["ab","a"] | "ab"::("a"::nil) |
| [1,2,3] | 1::(2::(3::nil)) |
| [[1]] | (1::nil)::nil |
| [[1],[2]] | (1::nil)::((2::nil)::nil) |
| [[[1]]] | ((1::nil)::nil)::nil |

**Above: lists defined in terms of nil and ::**

In order to determine the representation of a list, in terms of `::` and `nil`, it is sufficient to split the list into its head and tail, determine the representation of the head and tail, and then combine these using `::`. Consider, for example, the list `[[1],[2]]`. This list has head `[1]` and tail `[[2]]` and thus we need to determine the representation of these. The head `[1]` is simply `1::nil` and thus the list is equivalent to `(1::nil)::[[2]]`. The tail `[[2]]` itself has head `[2]` and tail `nil` and, in turn, `[2]` has head `2` and tail `nil`. Thus `[[2]]` is `(2::nil)::nil`. Thus `[[1],[2]]` is `(1::nil)::((2::nil)::nil)`.

The type of a list depends upon the type of the elements within the list. For a list whose elements have type $t_1$ the list has type $t_1$ `list`. Thus, for example, a list of integers has type `int list` while a list of lists of integers has type `int list list`. Similarly, we can have lists of strings or even lists of functions (all with the same type). The rule, that defines the type of a list in terms of the types of its elements, should explain why all of the elements of a list have to have the same type: if this was not the case there would be more than one choice for the type of a list!

Again, it is important to note that a list may contain elements that have an internal structure rather than just contain values, and this structure is important. Thus, for example, `[1]` and `[[1]]` are different elements and even have different types: `int list` and `int list list` respectively.

Most list functions are defined in terms of the constructors `::` and `nil`. Consider, for example, the problem of determining whether a list is empty. The following definition, of the function `empty_list`, suffices:

```
fun empty_list(nil) = true |
     empty_list(x::y) = false;
```

It is worth noting that in the second rule, the values of the variables x and y are not important: the output does not depend upon them. In order to state this, it is sufficient to use the special variable name _ for each of them giving:

```
fun empty_list(nil) = true |
     empty_list(_::_) = false;
```

This second definition explicitly states that, given a non-empty list, the output does not depend upon the value of the head and tail. In fact, some SML systems will respond to the first definition with a warning stating that the values of `x` and `y` are not used.

While the function `empty_list` is not recursive, most definitions of list functions are. This is a natural consequence of the recursive nature of the definition of a list. Naturally, there are usually two cases: the base case (usually `nil`) and the recursive case (usually a non-empty list `h::t`).

Consider the list function `member` that takes an element and a list and determines whether the element is in the list. The definition will be recursive since we need to be able to apply it to arbitrary length lists. We can identify the following cases:

1. The list is empty: the element is not in the list.
2. The list has head `h` and tail `t`: element `x` is in the list if either `x` is `h` or `x` is in `t`.

The first part can be seen as the base case, the second the recursive case. The recursive case in turn contains two sufficient conditions (one of which is like a base case). The following suffices:

```
fun member(x,nil) = false |
     member(x,h::t) = if (h=x) then true else
     member(x,t);
```

Rather than use an `if` statement, it is possible to more concisely define `member` using `orelse` as follows:

```
fun member(x,nil) = false |
     member(x,h::t) = (h=x) orelse member(x,t);
```

In fact these are equivalent since `orelse` and `andalso` actually stand for the following conditions:

```
e1 andalso e2       ≡    if el then e2 else false
e1 oresle e2        ≡    if el then true else e2
```

The version you use should thus depend upon the one with which you are happiest. It is also worth briefly mentioning the response SML will give to either of these definitions. This will be something like:

```
val member = fn: ''a * ''a list -> bool
```

The meaning of this is essentially that `member` is a function that takes an element, a list of elements of the same type and returns a boolean. Here `member` is polymorphic and `''a` is a type variable: when the function `member` is applied this type variable becomes instantiated. For example, when `member` is applied with arguments `1` and `[2]` `''a` takes on the value `int` and `member` takes on the type `int* int list -> bool`. You will learn more about polymorphism in the next chapter.

Now consider the way in which (the first version of) `member`, when applied to arguments, is evaluated. Suppose, initially, that member is evaluated with arguments `1` and `[2,3,1,4]`. Then the following process is applied:

```
member(1,[2,3,1,4]) →
if 1=2 then true else member(1,[3,1,4]) →
if false then true else member(1,[3,1,4]) →
member(1,[3,1,4]) →
if 1=3 then true else member(1,[1,4]) →
if false then true else member(1,[1,4]) →
member(1,[1,4]) →
if 1=1 then true else member(1,[4]) →
if true then true else member(1,[4]) →
true
```

Essentially, the evaluation has worked through the list until the element has been found. If the element is not in the list, the process continues until the empty list has been met and then the first rule is used. The following illustrates this:

```
member(1,[2,3]) →
if 1=2 then true else member(1,[3]) →
if false then true else member(1,[3]) →
member(1,[3]) →
if 1=3 then true else member(1,nil) →
if false then true else member(1,nil) →
member(1,nil) →
false
```

### *Writing code: advice*

It is important to note that before writing an SML function definition, it is usually best to determine the relationships between the elements being considered. Thus, when deciding how to define the membership function, we initially wrote down a recursive definition in natural language, and only then did we produce some SML code.

We might wish to add up the elements of an integer list. Here again we have a recursive function with two cases: one case for the empty list and one for a list with a head and a tail. We might observe that sum([t1,t2,…,tn]) = t1+t2+…+tn = t1+(t2+…+tn) = t1+sum([t2,…,tn]). This might form the basis of the recursive case, leading to the following natural language rules:

sum applied to the empty list is 0
sum applied to the list h::t is h added on to sum applied to t

This leads to the following definition:

```
fun sum(nil) = 0 |
     sum(h::t) = h+sum(t);
```

Suppose now that this function is applied to the list [2,5,3]. The evaluation proceeds as follows:

```
sum([2,5,3]) →
2+sum([5,3]) →
2+(5+sum([3])) →
2+(5+(3+sum(nil))) →
2+(5+(3+0)) →
2+(5+3) →
2+8 →
10
```

It is important to note the use of brackets to give the correct order of evaluation: without these brackets we might think that the numbers are added from the left.

Now consider the problem of defining a function to append two lists: take two lists and evaluate to these two lists combined. Now there are two lists to be considered and we might use recursion on either list or even both lists. In this case the solution is to apply recursion to the first list. This is because we can define append(l1,l2) in the following manner:

1. if l1 is the empty list then return l2
2. if l1 has head h and tail t, determine the result of appending t and l2 and then put h on the front of this.

This can be expressed in a manner closer to SML code in the following way

1. append(nil,l2) is l2
2. append(h::t,l2) is append(t,l2) with h 'put on the front'.

Finally, this is equivalent to the following SML code:

```
fun append(nil,x) = x |
     append(h::t,x) = h::append(t,x);
```

It is worth noting that SML has an in-built append function represented by infix operator `@`. Consider now the process of evaluating `append`:

```
append([1,2],[3,4]) →
1::append([2],[3,4]) →
1::(2::append(nil,[3,4])) →
1::(2::[3,4]) = [1,2,3,4]
```

Essentially, the second rule is applied repeatedly until the first argument is the empty list. At this point recursion ends and the first rule is applied.

While the recursive list functions we have considered so far have all had two cases, one the empty list and the other `h::t`, this need not be the case. Consider, for example, a function `len` that takes a list and evaluates to the string `"zero"` if the list is empty, `"one"` if the list contains exactly one element and `"many"` if it contains more than one element. This leads to the following definition:

```
fun len(nil) = "zero" |
     len([h]) = "one" |
     len(h1::(h2::t)) = "many";
```

**Activities**

1. Represent the following lists in terms of the constructors `nil` and `::`.

a) [2.3,54.2]
b) [["c"],[]]
c) [[[[]]]]

2. Define a list function `sl` that takes a list with two or more elements and returns the second element of the list.

3. Define a function `el` that takes an integer `n>0` and a list `x`, of length at least `n`, and returns the nth element of `x`.

4. Define a function `add1` that takes a list x of integers and returns the sum of the absolute values of the integers. Thus, for example, `add1([1,-1,-3])` should evaluate to `5`. **Hint**: if `x` is an integer then `abs(x)` is the absolute value of `x`.

5. Define the following SML functions, in each case giving the **type** of the function:

a) A function `biggest` that takes two lists `x` and `y` (of the same length) of integers and returns a list of this length with the property that the value in the ith position of the output is the maximum of the values in the ith positions of `x` and `y`. Thus, for example, `biggest([1,2,4],[6,0,2])` should evaluate to `[6,2,4]`.

b) A function `oddL` that takes an integer list `x` and evaluates to true if and only if all the elements in `x` are odd numbers. Thus, for example, `oddL([1,2])` returns `false` while `oddL(1,3])` returns `true`.

6. Give the step-by-step evaluation of the following terms, using the function definitions provided earlier.

a) `sum([1,2,~3])`

b) `append(["c","a"],["t"])`

c) `append([1,2,3],[])`

### *Summary*

This chapter has considered three structures within SML: tuple, records and lists. Each of these structures allows you to define your own types. The elements of these types are essentially structures containing elements whose type you specify. Tuples and records are non-recursive types: when a type is defined using these its size is fixed. They are thus useful for holding information when the amount of information is known in advance. In contrast, lists types allow arbitrary length lists. Lists are thus defined recursively, using the constructors `nil` and `::`.

Since lists are a recursive datatype, functions applied to lists are usually defined recursively. Normally list functions are defined in terms of two rules: a base case that deals with the empty list and a recursive case that deals with non-empty lists.

### *Learning outcomes*

After studying this chapter and completing the exercises and reading you should be able to:

- explain SML code that includes tuples, records, and lists
- write SML code that uses or manipulates tuples, records, and lists
- define a list in terms of `::` and `nil`;