2

Names, Functions and Types

Most functional languages are interactive. If you enter an expression, the computer immediately evaluates it and displays the result. Interaction is fun; it gives immediate feedback; it lets you develop programs in easily managed pieces.

We can enter an expression followed by a semicolon . . .

```
2+2;
```

. . . and ML responds

```
> 4 : int
```

Here we see some conventions that will be followed throughout the book. Most ML systems print a prompt character when waiting for input; here, the input is shown in `typewriter characters`. The response is shown, in slanted characters,

```
> on a line like this.
```

At its simplest, ML is just a calculator. It has integers, as shown above, and real numbers. ML can do simple arithmetic . . .

```
3.2 - 2.3;
> 0.9 : real
```

. . . and square roots:

```
Math.sqrt 2.0;
> 1.414213562 : real
```

Again, anything typed to ML must end with a semicolon (;). ML has printed the value and type. Note that *real* is the type of real numbers, while *int* is the type of integers.

Interactive program development is more difficult with procedural languages because they are too verbose. A self-contained program is too long to type as a single input.

Chapter outline

This chapter introduces Standard ML and functional programming. The basic concepts include declarations, simple data types, record types, recursive functions and polymorphism. Although this material is presented using Standard ML, it illustrates general principles.

The chapter contains the following sections:

Value declarations. Value and function declarations are presented using elementary examples.

Numbers, character strings and truth values. The built-in types *int*, *real*, *char*, *string* and *bool* support arithmetic, textual and logical operations.

Pairs, tuples and records. Ordered pairs and tuples allow functions to have multiple arguments and results.

The evaluation of expressions. The difference between strict evaluation and lazy evaluation is not just a matter of efficiency, but concerns the very meaning of expressions.

Writing recursive functions. Several worked examples illustrate the use of recursion.

Local declarations. Using `let` or `local`, names can be declared with a restricted scope.

Introduction to modules. Signatures and structures are introduced by developing a generic treatment of arithmetic operations.

Polymorphic type checking. The principles of polymorphism are introduced, including type inference and polymorphic functions.

Value declarations

A ***declaration*** gives something a name. ML has many kinds of things that can be named: values, types, signatures, structures and functors. Most names in a program stand for values, like numbers, strings — and functions. Although functions are values in ML, they have a special declaration syntax.

2.1 *Naming constants*

Any value of importance can be named, whether its importance is universal (like the constant π) or transient (the result of a previous computation). As a trivial example, suppose we want to compute the number of seconds in an hour. We begin by letting the name *seconds* stand for 60.

```
val seconds = 60;
```

The value declaration begins with ML keyword `val` and ends with a semicolon.

Names in this book usually appear in *italics*. ML repeats the name, with its value and type:

```
> val seconds = 60 : int
```

Let us declare constants for minutes per hour and hours per day:

```
val minutes = 60;
> val minutes = 60 : int
val hours = 24;
> val hours = 24 : int
```

These names are now valid in expressions:

```
seconds*minutes*hours;
> 86400 : int
```

If you enter an expression at top level like this, ML stores the value under the name *it*. By referring to *it* you can use the value in a further calculation:

```
it div 24;
> 3600 : int
```

The name *it* always has the value of the last expression typed at top level. Any previous value of *it* is lost. To save the value of *it*, declare a permanent name:

```
val secsinhour = it;
> val secsinhour = 3600 : int
```

Incidentally, names may contain underscores to make them easier to read:

```
val secs_in_hour = seconds*minutes;
> val secs_in_hour = 3600 : int
```

To demonstrate real numbers, we compute the area of a circle of radius r by the formula $area = \pi r^2$:

```
val pi = 3.14159;
> val pi = 3.14159 : real
val r = 2.0;
> val r = 2.0 : real
val area = pi * r * r;
> val area = 12.56636 : real
```

2.2 *Declaring functions*

The formula for the area of a circle can be made into an ML function like this:

```
fun area (r) = pi*r*r;
```

The keyword `fun` starts the function declaration, while *area* is the function name, *r* is the ***formal parameter***, and $pi*r*r$ is the ***body***. The body refers to r and to the constant *pi* declared above.

Because functions are values in ML, a function declaration is a form of value declaration, and so ML prints the value and type:

```
> val area = fn : real -> real
```

The type, which in standard mathematical notation is $real \to real$, says that *area* takes a real number as argument and returns another real number. The value of a function is printed as `fn`. In ML, as in most functional languages, functions are abstract values: their internal structure is hidden.

Let us call the function, repeating the area calculation performed above:

```
area(2.0);
> 12.56636 : real
```

Let us try it with a different argument. Observe that the parentheses around the argument are optional:

```
area 1.0;
> 3.14159 : real
```

The parentheses are also optional in function declarations. This definition of *area* is equivalent to the former one.

```
fun area r = pi*r*r;
```

The evaluation of function applications is discussed in more detail below.

Comments. Programmers often imagine that their creations are too transparent to require further description. This logical clarity will not be evident to others unless the program is properly commented. A comment can describe the purpose of a declaration, give a literature reference, or explain an obscure matter. Needless to say, comments must be correct and up-to-date.

A comment in Standard ML begins with `(*` and ends with `*)`, and may extend over several lines. Comments can even be nested. They can be inserted almost anywhere:

```
fun area r =     (*area of circle with radius r*)
      pi*r*r;
```

Functional programmers should not feel absolved from writing comments. People once claimed that Pascal was self-documenting.

Redeclaring a name. Value names are called ***variables***. Unlike variables in imperative languages, they cannot be updated. But a name can be reused for another purpose. If a name is declared again then the new meaning is adopted afterwards, but does not affect existing uses of the name. Let us redeclare the constant pi:

```
val pi = 0.0;
> val pi = 0.0 : real
```

We can see that *area* still takes the original value of pi:

```
area(1.0);
> 3.14159 : real
```

At this point in the session, several variables have values. These include $seconds$, $minutes$, $area$ and pi, as well as the built-in operations provided by the library. The set of bindings visible at any point is called the ***environment***. The function $area$ refers to an earlier environment in which pi denotes 3.14159. Thanks to the permanence of names (called ***static binding***), redeclaring a function cannot damage the system, the library or your program.

Correcting your program. Because of static binding, redeclaring a function called by your program may have no visible effect. When modifying a program, be sure to recompile the entire file. Large programs should be divided into modules; Chapter 7 will explain this in detail. After the modified module has been recompiled, the program merely has to be relinked.

2.3 *Identifiers in Standard ML*

An ***alphabetic name*** must begin with a letter, which may be followed by any number of letters, digits, underscores (_), or primes (`'`), usually called single quotes. For instance:

```
x    UB40    Hamlet_Prince_of_Denmark    h''3_H
```

The case of letters matters, so `q` differs from `Q`. Prime characters are allowed because ML was designed by mathematicians, who like variables called x, x', x''. When choosing names, be certain to avoid ML's keywords:

```
abstype and andalso as case datatype do
else end eqtype exception fn fun functor
handle if in include infix infixr let local
nonfix of op open orelse raise rec
sharing sig signature struct structure
then type val where while with withtype
```

Watch especially for the short ones: `as`, `fn`, `if`, `in`, `of`, `op`.

ML also permits ***symbolic names***. These consist of the characters

```
! % & $ # + - * / : < = > ? @ \ ~ ` ^ |
```

Names made up of these characters can be as long as you like:

```
---->    $^$^$^$    !!?@**??!!    :-|==>->#
```

Certain strings of special characters are reserved for ML's syntax and should not be used as symbolic names:

```
:  |  =  =>  ->  #  :>
```

A symbolic name is allowed wherever an alphabetic name is:

```
val +-+-+ = 1415;
> val +-+-+ = 1415 : int
```

Names are more formally known as ***identifiers***. An identifier can simultaneously denote a value, a type, a structure, a signature, a functor and a record field.

Exercise 2.1 On your computer, learn how to start an ML session and how to terminate it. Then learn how to make the ML compiler read declarations from a file — a typical command is *use* `"myfile"`.

Numbers, character strings and truth values

The simplest ML values are integer and real numbers, strings and characters, and the booleans or truth values. This section introduces these types with their constants and principal operations.

2.4 *Arithmetic*

ML distinguishes between integers (type *int*) and real numbers (type *real*). Integer arithmetic is exact (with unlimited precision in some ML systems) while real arithmetic is only as accurate as the computer's floating-point hardware.

Integers. An integer constant is a sequence of digits, possibly beginning with a minus sign (~). For instance:

```
0    ~23    01234    ~85601435654678
```

Integer operations include addition (+), subtraction (−), multiplication (∗), division (*div*) and remainder (*mod*). These are infix operators with conventional precedences: thus in

```
(((m*n)*k) - (m div j)) + j
```

all the parentheses can be omitted without harm.

Real numbers. A real constant contains a decimal point or E notation, or both. For instance:

```
0.01    2.718281828    ~1.2E12    7E~5
```

The ending En means 'times the nth power of 10.' A negative exponent begins with the unary minus sign (~). Thus 123.4E~2 denotes 1.234.

Negative real numbers begin with unary minus (~). Infix operators for reals include addition (+), subtraction (−), multiplication (*) and division (/). Function application binds more tightly than infix operators. For instance, *area a + b* is equivalent to (*area a*) + *b*, not *area* (*a* + *b*).

Unary plus and minus. The unary minus sign is a tilde (~). Do not confuse it with the subtraction sign (−)! ML has no unary plus sign. Neither + nor − may appear in the exponent of a real number.

Type constraints. ML can deduce the types in most expressions from the types of the functions and constants in it. But certain built-in functions are ***overloaded***, having more than one meaning. For example, + and * are defined for both integers and reals. The type of an overloaded function must be determined from the context; occasionally types must be stated explicitly.

For instance, ML cannot tell whether this squaring function is intended for integers or reals, and therefore rejects it.

```
fun square x = x*x;
> Error- Unable to resolve overloading for *
```

Suppose the function is intended for real numbers. We can insert the type *real* in a number of places.

We can specify the type of the argument:

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

We can specify the type of the result:

```
fun square x : real = x*x;
> val square = fn : real -> real
```

Equivalently, we can specify the type of the body:

```
fun square x = x*x : real;
> val square = fn : real -> real
```

Type constraints can also appear within the body, indeed almost anywhere.

Default overloading. The standard library introduces the notion of a default overloading; the compiler may resolve the ambiguity in *square* by choosing type *int*. Using a type constraint in such cases is still advisable, for clarity. The motivation for default overloadings is to allow different precisions of numbers to coexist. For example, unless the precision of `1.23` is determined by its context, it will be assumed to have the default precision for real numbers. As of this writing there is no experience of using different precisions, but care is plainly necessary.

Arithmetic and the standard library. The standard library includes numerous functions for integers and reals, of various precisions. Structure *Int* contains such functions as *abs* (absolute value), *min*, *max* and *sign*. Here are some examples:

```
Int.abs ~4;
> 4 : int
Int.min(7, Int.sign 12);
> 1 : int
```

Structure *Real* contains analogous functions such as *abs* and *sign*, as well as functions to convert between integers and reals. Calling *real*(*i*) converts *i* to the equivalent real number. Calling *round*(*r*) converts *r* to the nearest integer. Other real-to-integer conversions include *floor*, *ceil* and *trunc*. Conversion functions are necessary whenever integers and reals appear in the same expression.

Structure *Math* contains higher mathematical functions on real numbers, such as *sqrt*, *sin*, *cos*, *atan* (inverse tangent), *exp* and *ln* (natural logarithm). Each takes one real argument and returns a real result.

Exercise 2.2 A Lisp hacker says: ‘Since the integers are a subset of the real numbers, the distinction between them is wholly artificial — foisted on us by hardware designers. ML should simply provide numbers, as Lisp does, and automatically use integers or reals as appropriate.’ Do you agree? What considerations are there?

Exercise 2.3 Which of these function definitions require type constraints?

```
fun double(n) = 2*n;
fun f u = Math.sin(u)/u;
fun g k = ~ k * k;
```

2.5 *Strings and characters*

Messages and other text are strings of characters. They have type *string*. String constants are written in double quotes:

```
"How now! a rat? Dead, for a ducat, dead!";
> "How now! a rat? Dead, for a ducat, dead!" : string
```

The concatenation operator (^) joins two strings end-to-end:

```
"Fair " ^ "Ophelia";
> "Fair Ophelia" : string
```

The built-in function *size* returns the number of characters in a string. Here *it* refers to "Fair Ophelia":

```
size (it);
> 12 : int
```

The space character counts, of course. The empty string contains no characters; *size* ("") is 0.

Here is a function that makes noble titles:

```
fun title(name) = "The Duke of " ^ name;
> val title = fn : string -> string
title "York";
> "The Duke of York" : string
```

Special characters. ***Escape sequences***, which begin with a backslash (\), insert certain special characters into a string. Here are some of them:

- \n inserts a newline character (line break).
- \t inserts a tabulation character.
- \" inserts a double quote.
- \\ inserts a backslash.
- \ followed by a newline and other white-space characters, followed by another \ inserts nothing, but continues a string across the line break.

Here is a string containing newline characters:

```
"This above all:\nto thine own self be true\n";
```

The type char. Just as the number 3 differs from the set {3}, a character differs from a one-character string. Characters have type *char*. The constants have the form #*s*, where *s* is a string constant consisting of a single character. Here is a letter, a space and a special character:

```
#"a"    #" "    #"\n"
```

The functions *ord* and *chr* convert between characters and character codes. Most implementations use the ASCII character set; if k is in the range $0 \leq$

$k \leq 255$ then $chr(k)$ returns the character with code k. Conversely, $ord(c)$ is the integer code of the character c. We can use these to convert a number between 0 and 9 to a character between `#"0"` and `#"9"`:

```
fun digit i = chr(i + ord #"0");
> val digit = fn : int -> char
```

The functions str and $String.sub$ convert between characters and strings. If c is a character then $str(c)$ is the corresponding string. Conversely, if s is a string then $String.sub(s, n)$ returns the nth character in s, counting from zero. Let us try these, first expressing the function $digit$ differently:

```
fun digit i = String.sub("0123456789", i);
> val digit = fn : int -> char
str (digit 5);
> "5" : string
```

The second definition of $digit$ is preferable to the first, as it does not rely on character codes.

Strings, characters and the standard library. Structure $String$ contains numerous operations on strings. Structure $Char$ provides functions such as $isDigit$, $isAlpha$, etc., to recognize certain classes of character. A ***substring*** is a contiguous subsequence of characters from a string; structure $Substring$ provides operations for extracting and manipulating them.

The ML *Definition* only has type $string$ (Milner *et al.*, 1990). The standard library introduces the type $char$. It also modifies the types of built-in functions such as ord and chr, which previously operated on single-character strings.

Exercise 2.4 For each version of $digit$, what do you expect in response to the calls $digit$ `~1` and $digit$ `10`? Try to predict the response before experimenting on the computer.

2.6 *Truth values and conditional expressions*

To define a function by cases — where the result depends on the outcome of a test — we employ a conditional expression.[1] The test is an expression E of type $bool$, whose values are $true$ and $false$. The outcome of the test chooses one of two expressions E_1 or E_2. The value of the conditional expression

`if` E `then` E_1 `else` E_2

[1] Because a Standard ML expression can update the state, conditional expressions can also act like the `if` commands of procedural languages.

is that of E_1 if E equals $true$, and that of E_2 if E equals $false$. The `else` part is mandatory.

The simplest tests are the relations:

- less than (<)
- greater than (>)
- less than or equals (<=)
- greater than or equals (>=)

These are defined on integers and reals; they also test alphabetical ordering on strings and characters. Thus the relations are overloaded and may require type constraints. Equality (=) and its negation (<>) can be tested for most types.

For example, the function *sign* computes the sign (1, 0, or -1) of an integer. It has two conditional expressions and a comment.

```
fun sign(n) =
         if n>0 then 1
    else if n=0 then 0
    else (*n<0*)     ~1;
> val sign = fn : int ->int
```

Tests are combined by ML's boolean operations:

- logical or (called `orelse`)
- logical and (called `andalso`)
- logical negation (the function *not*)

Functions that return a boolean value are known as ***predicates***. Here is a predicate to test whether its argument, a character, is a lower-case letter:

```
fun isLower c = #"a" <= c andalso c <= #"z";
> val isLower = fn : char -> bool
```

When a conditional expression is evaluated, either the `then` or the `else` expression is evaluated, never both. The boolean operators `andalso` and `orelse` behave differently from ordinary functions: the second operand is evaluated only if necessary. Their names reflect this sequential behaviour.

Exercise 2.5 Let d be an integer and m a string. Write an ML boolean expression that is true just when d and m form a valid date: say `25` and `"October"`. Assume it is not a leap year.

Pairs, tuples and records

In mathematics, a collection of values is often viewed as a single value. A vector in two dimensions is an ordered pair of real numbers. A statement about two vectors $\vec{v}_1$ and $\vec{v}_2$ can be taken as a statement about four real numbers, and those real numbers can themselves be broken down into smaller pieces, but thinking at a high level is easier. Writing $\vec{v}_1 + \vec{v}_2$ for their vector sum saves us from writing $(x_1 + x_2, y_1 + y_2)$.

Dates are a more commonplace example. A date like 25 October 1415 consists of three values. Taken as a unit, it is a triple of the form $(day, month, year)$. This elementary concept has taken remarkably long to appear in programming languages, and only a few handle it properly.

Standard ML provides ordered pairs, triples, quadruples and so forth. For $n \geq 2$, the ordered collection of n values is called an n-tuple, or just a ***tuple***. The tuple whose components are $x_1, x_2, \ldots, x_n$ is written $(x_1, x_2, \ldots, x_n)$. Such a value is created by an expression of the form $(E_1, E_2, \ldots, E_n)$. With functions, tuples give the effect of multiple arguments and results.

The components of an ML tuple may themselves be tuples or any other value. For example, a period of time can be represented by a pair of dates, regardless of how dates are represented. It also follows that nested pairs can represent n-tuples. (In Classic ML, the original dialect, $(x_1, \ldots, x_{n-1}, x_n)$ was merely an abbreviation for $(x_1, \ldots, (x_{n-1}, x_n) \ldots)$.)

An ML ***record*** has components identified by name, not by position. A record with 20 components occupies a lot of space on the printed page, but is easier to manage than a 20-tuple.

2.7 *Vectors: an example of pairing*

Let us develop the example of vectors. To try the syntax for pairs, enter the vector (2.5, −1.2):

```
(2.5, ~1.2);
> (2.5, ~1.2) : real * real
```

The vector's type, which in mathematical notation is $real \times real$, is the type of a pair of real numbers. Vectors are ML values and can be given names. We declare the zero vector and two others, called a and b.

```
val zerovec = (0.0, 0.0);
> val zerovec = (0.0, 0.0) : real * real
val a = (1.5, 6.8);
> val a = (1.5, 6.8) : real * real
val b = (3.6, 0.9);
```

```
> val b = (3.6, 0.9) : real * real
```

Many functions on vectors operate on the components. The length of (x, y) is $\sqrt{x^2 + y^2}$, while the negation of (x, y) is $(-x, -y)$. To code these functions in ML, simply write the argument as a pattern:

```
fun lengthvec (x,y) = Math.sqrt(x*x + y*y);
> val lengthvec = fn : real * real -> real
```

The function *lengthvec* takes the pair of values of x and y. It has type $real \times real \to real$: its argument is a pair of real numbers and its result is another real number.[2] Here, a is a pair of real numbers.

```
lengthvec a;
> 6.963476143 : real
lengthvec (1.0, 1.0);
> 1.414213562 : real
```

Function *negvec* negates a vector with respect to the point (0, 0).

```
fun negvec (x,y) : real*real = (~x, ~y);
> val negvec = fn : real * real -> real * real
```

This function has type $real \times real \to real \times real$: given a pair of real numbers it returns another pair. The type constraint $real \times real$ is necessary because minus (~) is overloaded.

We negate some vectors, giving a name to the negation of b:

```
negvec (1.0, 1.0);
> (~1.0, ~1.0) : real * real
val bn = negvec(b);
> val bn = (~3.6, ~0.9) : real * real
```

Vectors can be arguments and results of functions and can be given names. In short, they have all the rights of ML's built-in values, like the integers. We can even declare a type of vectors:

```
type vec = real*real;
> type vec
```

Now *vec* abbreviates $real \times real$. It is only an abbreviation though: every pair of real numbers has type *vec*, regardless of whether it is intended to represent a vector. We shall employ *vec* in type constraints.

[2] function *Math.sqrt*, which is defined only for real numbers, constrains the overloaded operators to type *real*.

2.8 Functions with multiple arguments and results

Here is a function that computes the average of a pair of real numbers.

```
fun average(x,y) = (x+y)/2.0;
> val average = fn : (real * real) -> real
```

This would be an odd thing to do to a vector, but *average* works for any two numbers:

```
average(3.1,3.3);
> 3.2 : real
```

A function on pairs is, in effect, a function of two arguments: $lengthvec(x, y)$ and $average(x, y)$ operate on the real numbers x and y. Whether we view (x, y) as a vector is up to us. Similarly *negvec* takes a pair of arguments — and returns a pair of results.

Strictly speaking, every ML function has one argument and one result. With tuples, functions can effectively have any number of arguments and results. Currying, discussed in Chapter 5, also gives the effect of multiple arguments.

Since the components of a tuple can themselves be tuples, two vectors can be paired:

```
((2.0, 3.5), zerovec);
> ((2.0, 3.5), (0.0, 0.0)) : (real*real) * (real*real)
```

The sum of vectors (x_1, y_1) and (x_2, y_2) is $(x_1 + x_2, y_1 + y_2)$. In ML, this function takes a pair of vectors. Its argument pattern is a pair of pairs:

```
fun addvec ((x1,y1), (x2,y2)) : vec = (x1+x2, y1+y2);
> val addvec = fn : (real*real) * (real*real) -> vec
```

Type *vec* appears for the first time, constraining addition to operate on real numbers. ML gives *addvec* the type

$$((real \times real) \times (real \times real)) \to vec$$

which is equivalent to the more concise $(vec \times vec) \to vec$. The ML system may not abbreviate every $real \times real$ as *vec*.

Look again at the argument pattern of *addvec*. We may equivalently view this function as taking

- one argument: a pair of pairs of real numbers
- two arguments: each a pair of real numbers
- four arguments: all real numbers, oddly grouped

Here we add the vectors (8.9,4.4) and b, then add the result to another vector. Note that vec is the result type of the function.

```
addvec((8.9, 4.4), b);
> (12.5, 5.3) : vec
addvec(it, (0.1, 0.2));
> (12.6, 5.5) : vec
```

Vector subtraction involves subtraction of the components, but can be expressed by vector operations:

```
fun subvec(v1,v2) = addvec(v1, negvec v2);
> val subvec = fn : (real*real) * (real*real) -> vec
```

The variables $v1$ and $v2$ range over pairs of reals.

```
subvec(a,b);
> (~2.1, 5.9) : vec
```

The distance between two vectors is the length of the difference:

```
fun distance(v1,v2) = lengthvec(subvec(v1,v2));
> val distance = fn : (real*real) * (real*real) -> real
```

Since $distance$ never refers separately to $v1$ or $v2$, it can be simplified:

```
fun distance pairv = lengthvec(subvec pairv);
```

The variable $pairv$ ranges over pairs of vectors. This version may look odd, but is equivalent to its predecessor. How far is it from a to b?

```
distance(a,b);
> 6.262587325 : real
```

A final example will show that the components of a pair can have different types: here, a real number and a vector. Scaling a vector means multiplying both components by a constant.

```
fun scalevec (r, (x,y)) : vec = (r*x, r*y);
> val scalevec = fn : real * (real*real) -> vec
```

The type constraint vec ensures that the multiplications apply to reals. The function $scalevec$ takes a real number and a vector, and returns a vector.

```
scalevec(2.0, a);
> (3.0, 13.6) : vec
scalevec(2.0, it);
> (6.0, 27.2) : vec
```

Selecting the components of a tuple. A function defined on a pattern, say (x, y), refers to the components of its argument through the pattern variables x and y. A `val` declaration may also match a value against a pattern: each variable in the pattern refers to the corresponding component.

Here we treat $scalevec$ as a function returning two results, which we name xc and yc.

```
val (xc,yc) = scalevec(4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real
```

The pattern in a `val` declaration can be as complicated as the argument pattern of a function definition. In this contrived example, a pair of pairs is split into four parts, which are all given names.

```
val ((x1,y1), (x2,y2)) = (addvec(a,b), subvec(a,b));
> val x1 = 5.1 : real
> val y1 = 7.7 : real
> val x2 = ~2.1 : real
> val y2 = 5.9 : real
```

The 0-tuple and the type unit. Previously we have considered n-tuples for $n \geq 2$. There is also a 0-tuple, written () and pronounced 'unity,' which has no components. It serves as a placeholder in situations where no data needs to be conveyed. The 0-tuple is the sole value of type $unit$.

Type $unit$ is often used with procedural programming in ML. A procedure is typically a 'function' whose result type is $unit$. The procedure is called for its effect — not for its value, which is always (). For instance, some ML systems provide a function use of type $string \to unit$. Calling use `"myfile"` has the effect of reading the definitions on the file `"myfile"` into ML.

A function whose argument type is $unit$ passes no information to its body when called. Calling the function simply causes its body to be evaluated. In Chapter 5, such functions are used to delay evaluation for programming with infinite lists.

Exercise 2.6 Write a function to determine whether one time of day, in the form ($hours$, $minutes$, AM or PM), comes before another. As an example, `(11, 59, "AM")` comes before `(1, 15, "PM")`.

Exercise 2.7 Old English money had 12 pence in a shilling and 20 shillings in a pound. Write functions to add and subtract two amounts, working with triples ($pounds$, $shillings$, $pence$).

2.9 Records

A record is a tuple whose components — called ***fields*** — have labels. While each component of an n-tuple is identified by its position from 1 to n, the fields of a record may appear in any order. Transposing the components of a tuple is a common error. If employees are taken as triples $(name, age, salary)$ then there is a big difference between `("Jones", 25, 15300)` and `("Jones", 15300, 25)`. But the records

```
{name="Jones", age=25, salary=15300}
```

and

```
{name="Jones", salary=15300, age=25}
```

are equal. A record is enclosed in braces `{...}`; each field has the form $label = expression$.

Records are appropriate when there are many components. Let us record five fundamental facts about some Kings of England, and note ML's response:

```
val henryV =
  {name    = "Henry V",
   born    = 1387,
   crowned = 1413,
   died    = 1422,
   quote   = "Bid them achieve me and then sell my bones"};
> val henryV =
>   {born = 1387,
>    died = 1422,
>    name = "Henry V",
>    quote = "Bid them achieve me and then sell my bones",
>    crowned = 1413}
> : {born: int,
>    died: int,
>    name: string,
>    quote: string,
>    crowned: int}
```

ML has rearranged the fields into a standard order, ignoring the order in which they were given. The record type lists each field as $label : type$, within braces.

Here are two more Kings:

```
val henryVI =
  {name    = "Henry VI",
   born    = 1421,
   crowned = 1422,
   died    = 1471,
   quote   = "Weep, wretched man, \
\ I'll aid thee tear for tear"};
```

```
val richardIII =
  {name    = "Richard III",
   born    = 1452,
   crowned = 1483,
   died    = 1485,
   quote   = "Plots have I laid..."};
```

The *quote* of *henryVI* extends across two lines, using the backslash, newline, backslash escape sequence.

Record patterns. A record pattern with fields *label* = *variable* gives each variable the value of the corresponding label. If we do not need all the fields, we can write three dots (`...`) in place of the others. Here we get two fields from Henry V's famous record, calling them *nameV* and *bornV*:

```
val {name=nameV, born=bornV, ...} = henryV;
> val nameV = "Henry V" : string
> val bornV = 1387 : int
```

Often we want to open up a record, making its fields directly visible. We can specify each field in the pattern as *label* = *label*, making the variable and the label identical. Such a specification can be shortened to simply *label*. We open up Richard III:

```
val {name,born,died,quote,crowned} = richardIII;
> val name = "Richard III" : string
> val born = 1452 : int
> val died = 1485 : int
> val quote = "Plots have I laid..." : string
> val crowned = 1483 : int
```

To omit some fields, write (`...`) as before. Now *quote* stands for the quote of Richard III. Obviously this makes sense for only one King at a time.

Record field selections. The selection #*label* gets the value of the given *label* from a record.

```
#quote richardIII;
> "Plots have I laid..." : string
#died henryV - #born henryV;
> 35 : int
```

Different record types can have labels in common. Both employees and Kings have a *name*, whether `"Jones"` or `"Henry V"`. The three Kings given above have the same record type because they have the same number of fields with the same labels and types.

Here is another example of different record types with some labels in common: the n-tuple $(x_1, x_2, \ldots, x_n)$ is just an abbreviation for a record with numbered fields:

$$\{1 = x_1, 2 = x_2, \ldots, n = x_n\}$$

Yes, a label can be a positive integer! This obscure fact about Standard ML is worth knowing for one reason: the selector `#`k gets the value of component k of an n-tuple. So `#1` selects the first component and `#2` selects the second. If there is a third component then `#3` selects it, and so forth:

```
#2 ("a","b",3,false);
> "b" : string
```

Partial record specifications. A field selection that omits some of the fields does not completely specify the record type; a function may only be defined over a complete record type. For instance, a function cannot be defined for all records that have fields *born* and *died*, without specifying the full set of field names (typically using a type constraint). This restriction makes ML records efficient but inflexible. It applies equally to record patterns and field selections of the form `#`*label*. Ohori (1995) has defined and implemented flexible records for a variant of ML.

Declaring a record type. Let us declare the record type of Kings. This abbreviation will be useful for type constraints in functions.

```
type king = {name    : string,
             born    : int,
             crowned : int,
             died    : int,
             quote   : string};
> type king
```

We now can declare a function on type *king* to return the King's lifetime:

```
fun lifetime(k: king) = #died k - #born k;
> val lifetime = fn : king -> int
```

Using a pattern, *lifetime* can be declared like this:

```
fun lifetime({born,died,...}: king) = died - born;
```

Either way the type constraint is mandatory. Otherwise ML will print a message like 'A fixed record type is needed here.'

```
lifetime henryV;
> 35 : int
lifetime richardIII;
> 33 : int
```

Exercise 2.8 Does the following function definition require a type constraint? What is its type?

```
fun lifetime({name, born, crowned, died, quote}) = died - born;
```

Exercise 2.9 Discuss the differences, if any, between the selector #*born* and the function

```
fun born_at({born}) = born;
```

2.10 *Infix operators*

An ***infix operator*** is a function that is written between its two arguments. We take infix notation for granted in mathematics. Imagine doing without it. Instead of 2+2=4 we should have to write =(+(2,2),4). Most functional languages let programmers declare their own infix operators.

Let us declare an infix operator *xor* for 'exclusive or.' First we issue an ML `infix` directive:

```
infix xor;
```

We now must write *p xor q* rather than *xor*(*p*, *q*):

```
fun (p xor q) = (p orelse q) andalso not (p andalso q);
> val xor = fn : (bool * bool) -> bool
```

The function *xor* takes a pair of booleans and returns a boolean.

```
true xor false xor true;
> false : bool
```

The infix status of a name concerns only its syntax, not its value, if any. Usually a name is made infix before it has any value at all.

Precedence of infixes. Most people take $m \times n + i/j$ to mean $(m \times n) + (i/j)$, giving $\times$ and / higher precedence than $+$. Similarly $i - j - k$ means $(i - j) - k$, since the operator $-$ associates to the left. An ML infix directive may state a precedence from 0 to 9. The default precedence is 0, which is the lowest. The directive `infix` causes association to the left, while `infixr` causes association to the right.

To demonstrate infixes, the following functions construct strings enclosed in parentheses. Operator *plus* has precedence 6 (the precedence of + in ML) and constructs a string containing a + sign.

```
infix 6 plus;
fun (a plus b) = "(" ^ a ^ "+" ^ b ^ ")";
```

```
> val plus = fn : string * string -> string
```

Observe that *plus* associates to the left:

```
"1" plus "2" plus "3";
> "((1+2)+3)" : string
```

Similarly, *times* has precedence 7 (like * in ML) and constructs a string containing a * sign.

```
infix 7 times;
fun (a times b) = "(" ^ a ^ "*" ^ b ^ ")";
> val times = fn : string * string -> string
"m" times "n" times "3" plus "i" plus "j" times "k";
> "(((m*n)*3)+i)+(j*k))" : string
```

The operator *pow* has higher precedence than times and associates to the right, which is traditional for raising to a power. It produces a # sign. (ML has no operator for powers.)

```
infixr 8 pow;
fun (a pow b) = "(" ^ a ^ "#" ^ b ^ ")";
> val pow = fn : string * string -> string
"m" times "i" pow "j" pow "2" times "n";
> "((m*(i#(j#2)))*n)" : string
```

Many infix operators have symbolic names. Let ++ be the operator for vector addition:

```
infix ++;
fun ((x1,y1) ++ (x2,y2)) : vec = (x1+x2, y1+y2);
> val ++ = fn : (real*real) * (real*real) -> vec
```

It works exactly like *addvec*, but with infix notation:

```
b ++ (0.1,0.2) ++ (20.0, 30.0);
> (23.7, 31.1) : vec
```

Keep symbolic names separate. Symbolic names can cause confusion if you run them together. Below, ML reads the characters +~ as one symbolic name, then complains that this name has no value:

```
1+~3;
> Unknown name +~
```

Symbolic names must be separated by spaces or other characters:

```
1+ ~3;
> ~2 : int
```

Taking infixes as functions. Occasionally an infix has to be treated like an ordinary function. In ML the keyword `op` overrides infix status: if $\oplus$ is an infix operator then op$\oplus$ is the corresponding function, which can be applied to a pair in the usual way.

```
> op++ ((2.5,0.0), (0.1,2.5));
(2.6, 2.5) : real * real
op^ ("Mont","joy");
> "Montjoy" : string
```

Infix status can be revoked. If $\oplus$ is an infix operator then the directive nonfix$\oplus$ makes it revert to ordinary function notation. A subsequent infix directive can make $\oplus$ an infix operator again.

Here we deprive ML's multiplication operator of its infix status. The attempt to use it produces an error message, since we may not apply 3 as a function. But $\star$ can be applied as a function:

```
nonfix *;
3*2;
> Error: Type conflict...
*(3,2);
> 6 : int
```

The `nonfix` directive is intended for interactive development of syntax, for trying different precedences and association. Changing the infix status of established operators leads to madness.

The evaluation of expressions

An imperative program specifies commands to update the machine state. During execution, the state changes millions of times per second. Its structure changes too: local variables are created and destroyed. Even if the program has a mathematical meaning independent of hardware details, that meaning is beyond the comprehension of the programmer. Axiomatic and denotational semantic definitions make sense only to a handful of experts. Programmers trying to correct their programs rely on debugging tools and intuition.

Functional programming aims to give each program a straightforward mathematical meaning. It simplifies our mental image of execution, for there are no state changes. Execution is the reduction of an expression to its value, replacing equals by equals. Most function definitions can be understood within elementary mathematics.

When a function is applied, as in $f(E)$, the argument E must be supplied to the body of f. If the expression contains several function calls, one must be

chosen according to some evaluation rule. The evaluation rule in ML is ***call-by-value*** (or ***strict*** evaluation), while most purely functional languages adopt ***call-by-need*** (or ***lazy*** evaluation).

Each evaluation rule has its partisans. To compare the rules we shall consider two trivial functions. The squaring function sqr uses its argument twice:

```
fun sqr(x) : int = x*x;
> val sqr = fn : int -> int
```

The constant function $zero$ ignores its argument and returns 0:

```
fun zero(x : int) = 0;
> val zero = fn : int -> int
```

When a function is called, the argument is substituted for the function's formal parameter in the body. The evaluation rules differ over when, and how many times, the argument is evaluated. The formal parameter indicates where in the body to substitute the argument. The name of the formal parameter has no other significance, and no significance outside of the function definition.

2.11 *Evaluation in ML: call-by-value*

Let us assume that expressions consist of constants, variables, function calls and conditional expressions (`if-then-else`). Constants have explicit values; variables have bindings in the environment. So evaluation has only to deal with function calls and conditionals. ML's evaluation rule is based on an obvious idea.

> To compute the value of $f(E)$, first compute the value of the expression E.

This value is substituted into the body of f, which then can be evaluated. Pattern-matching is a minor complication. If f is declared by, say

```
fun f (x,y,z) = body
```

then substitute the corresponding parts of E's value for the pattern variables x, y and z. (A practical implementation performs no substitutions, but instead binds the formal parameters in the local environment.)

Consider how ML evaluates $sqr(sqr(sqr(2)))$. Of the three function calls, only the innermost call has a value for the argument. So $sqr(sqr(sqr(2)))$ reduces to $sqr(sqr(2 \times 2))$. The multiplication must now be evaluated, yielding $sqr(sqr(4))$. Evaluating the inner call yields $sqr(4 \times 4)$, and so forth. Reduc-

tions are written $sqr(sqr(4)) \Rightarrow sqr(4 \times 4)$. The full evaluation looks like this:

$$\begin{aligned} sqr(sqr(sqr(2))) &\Rightarrow sqr(sqr(2 \times 2)) \\ &\Rightarrow sqr(sqr(4)) \\ &\Rightarrow sqr(4 \times 4) \\ &\Rightarrow sqr(16) \\ &\Rightarrow 16 \times 16 \\ &\Rightarrow 256 \end{aligned}$$

Now consider $zero(sqr(sqr(sqr(2))))$. The argument of $zero$ is the expression evaluated above. It is evaluated but the value is ignored:

$$\begin{aligned} zero(sqr(sqr(sqr(2)))) &\Rightarrow zero(sqr(sqr(2 \times 2))) \\ &\vdots \\ &\Rightarrow zero(256) \\ &\Rightarrow 0 \end{aligned}$$

Such waste! Functions like $zero$ are uncommon, but frequently a function's result does not depend on all of its arguments.

ML's evaluation rule is known as ***call-by-value*** because a function is always given its argument's value. It is not hard to see that call-by-value corresponds to the usual way we should perform a calculation on paper. Almost all programming languages adopt it. But perhaps we should look for an evaluation rule that reduces $zero(sqr(sqr(sqr(2))))$ to 0 in one step. Before such issues can be examined, we must have a look at recursion.

2.12 *Recursive functions under call-by-value*

The factorial function is a standard example of recursion. It includes a base case, $n = 0$, where evaluation stops.

```
fun fact n =
      if n=0 then 1 else n * fact(n-1);
> val fact = fn : int -> int
fact 7;
> 5040 : int
fact 35;
> 10333147966386144929666651337523200000000 : int
```

ML evaluates $fact(4)$ as follows. The argument, 4, is substituted for n in the body, yielding

```
if 4 = 0 then 1 else 4 × fact(4 − 1)
```

Figure 2.1 *Evaluation of* $fact(4)$

$$
\begin{aligned}
fact(4) &\Rightarrow 4 \times fact(4-1)\\
&\Rightarrow 4 \times fact(3)\\
&\Rightarrow 4 \times (3 \times fact(3-1))\\
&\Rightarrow 4 \times (3 \times fact(2))\\
&\Rightarrow 4 \times (3 \times (2 \times fact(2-1)))\\
&\Rightarrow 4 \times (3 \times (2 \times fact(1)))\\
&\Rightarrow 4 \times (3 \times (2 \times (1 \times fact(1-1))))\\
&\Rightarrow 4 \times (3 \times (2 \times (1 \times fact(0))))\\
&\Rightarrow 4 \times (3 \times (2 \times (1 \times 1)))\\
&\Rightarrow 4 \times (3 \times (2 \times 1))\\
&\Rightarrow 4 \times (3 \times 2)\\
&\Rightarrow 4 \times 6\\
&\Rightarrow 24
\end{aligned}
$$

Figure 2.2 *Evaluation of* $facti(4, 1)$

$$
\begin{aligned}
facti(4, 1) &\Rightarrow facti(4-1, 4 \times 1)\\
&\Rightarrow facti(3, 4)\\
&\Rightarrow facti(3-1, 3 \times 4)\\
&\Rightarrow facti(2, 12)\\
&\Rightarrow facti(2-1, 2 \times 12)\\
&\Rightarrow facti(1, 24)\\
&\Rightarrow facti(1-1, 1 \times 24)\\
&\Rightarrow facti(0, 24)\\
&\Rightarrow 24
\end{aligned}
$$

Since $4 = 0$ is false, the conditional reduces to $4 \times fact(4-1)$. Then $4-1$ is selected, and the entire expression reduces to $4 \times fact(3)$. Figure 2.1 summarizes the evaluation. The conditionals are not shown: they behave similarly apart from $n = 0$, when the conditional returns 1.

The evaluation of $fact(4)$ exactly follows the mathematical definition of factorial: $0! = 1$, and $n! = n \times (n-1)!$ if $n > 0$. Could the execution of a recursive procedure be shown as succinctly?

Iterative functions. Something is odd about the computation of $fact(4)$. As the recursion progresses, more and more numbers are waiting to be multiplied. The multiplications cannot take place until the recursion terminates with $fact(0)$. At that point $4 \times (3 \times (2 \times (1 \times 1)))$ must be evaluated. This paper calculation shows that $fact$ is wasting space.

A more efficient version can be found by thinking about how we should compute factorials. By the associative law, each multiplication can be done at once:

$$4 \times (3 \times fact(2)) = (4 \times 3) \times fact(2) = 12 \times fact(2)$$

The computer will not apply such laws unless we force it to. The function $facti$ keeps a running product in p, which initially should be 1:

```
fun facti (n,p) =
      if n=0 then p  else  facti(n-1, n*p);
> val facti = fn : int * int -> int
```

Compare the evaluation for $facti(4, 1)$, shown in Figure 2.2, with that of $fact(4)$. The intermediate expressions stay small; each multiplication can be done at once; storage requirements remain constant. The evaluation is ***iterative*** — also termed ***tail recursive***. In Section 6.3 we shall prove that $facti$ gives correct results by establishing the law $facti(n, p) = n! \times p$.

Good compilers detect iterative forms of recursion and execute them efficiently. The result of the recursive call $facti(n-1, n \times p)$ undergoes no further computation, but is immediately returned as the value of $facti(n, p)$. Such a ***tail call*** can be executed by assigning the arguments n and p their new values and then jumping back into the function, avoiding the cost of a proper function invocation. The recursive call in $fact$ is not a tail call because its value undergoes further computation, namely multiplication by n.

Many functions can be made iterative by adding an argument, like p in $facti$. Sometimes the iterative function runs much faster. Sometimes, making a function iterative is the only way to avoid running out of store. However, adding

an extra argument to every recursive function is a bad habit. It leads to ugly, convoluted code that might run slower than it should.

The special rôle of conditional expressions. The conditional expression permits definition by cases. Recall how the factorial function is defined:

$$0! = 1$$
$$n! = n \times (n-1)! \qquad \text{for } n > 0$$

These equations determine $n!$ for all integers $n \geq 0$. Omitting the condition $n > 0$ from the second equation would lead to absurdity:

$$1 = 0! = 0 \times (-1)! = 0$$

Similarly, in the conditional expression

```
if E then E1 else E2,
```

ML evaluates E_1 only if $E = true$, and evaluates E_2 only if $E = false$.

Due to call-by-value, there is no ML function $cond$ such that $cond(E, E_1, E_2)$ is evaluated like a conditional expression. Let us try to declare one and use it to code the factorial function:

```
fun cond(p,x,y) : int = if p then x else y;
> val cond = fn : bool * int * int -> int
fun badf n = cond(n=0, 1, n*badf(n-1));
> val badf = fn : int -> int
```

This may look plausible, but every call to $badf$ runs forever. Observe the evaluation of $badf(0)$:

$$\begin{aligned} badf(0) &\Rightarrow cond(true, 1, 0 \times badf(-1)) \\ &\Rightarrow cond(true, 1, 0 \times cond(false, 1, -1 \times badf(-2))) \\ &\vdots \end{aligned}$$

Although $cond$ never requires the values of all three of its arguments, the call-by-value rule evaluates them all. The recursion cannot terminate.

Conditional and/or. ML's boolean infix operators `andalso` and `orelse` are not functions, but stand for conditional expressions.

The expression E_1 `andalso` E_2 abbreviates

```
if E1 then E2 else false.
```

The expression E_1 `orelse` E_2 abbreviates

if E_1 then *true* else E_2.

These operators compute the boolean and/or, but evaluate E_2 only if necessary. If they were functions, the call-by-value rule would evaluate both arguments. All other ML infixes are really functions.

The sequential evaluation of `andalso` and `orelse` makes them ideal for expressing recursive predicates (boolean-valued functions). The function *powoftwo* tests whether a number is a power of two:

```
fun even n  =  (n mod 2 = 0);
> val even = fn : int -> bool
fun powoftwo n =  (n=1)  orelse
                  (even(n)  andalso powoftwo(n div 2));
> val powoftwo = fn : int -> bool
```

You might expect *powoftwo* to be defined by conditional expressions, and so it is, through `orelse` and `andalso`. Evaluation terminates once the outcome is decided:

$$
\begin{aligned}
powoftwo(6) &\Rightarrow (6 = 1) \texttt{ orelse } (even(6) \texttt{ andalso } \cdots) \\
&\Rightarrow even(6) \texttt{ andalso } powoftwo(6 \, div \, 2) \\
&\Rightarrow powoftwo(3) \\
&\Rightarrow (3 = 1) \texttt{ orelse } (even(3) \texttt{ andalso } \cdots) \\
&\Rightarrow even(3) \texttt{ andalso } powoftwo(3 \, div \, 2) \\
&\Rightarrow false
\end{aligned}
$$

Exercise 2.10 Write the reduction steps for $powoftwo(8)$.

Exercise 2.11 Is $powoftwo$ an iterative function?

2.13 *Call-by-need, or lazy evaluation*

The call-by-value rule has accumulated a catalogue of complaints. It evaluates E superfluously in $zero(E)$. And it evaluates $E1$ or $E2$ superfluously in $cond(E, E1, E2)$. Conditional expressions and similar operations cannot be functions. ML provides `andalso` and `orelse`, but we have no means of defining similar things.

Shall we give functions their arguments as expressions, not as values? The general idea is this:

> To compute the value of $f(E)$, substitute E immediately into the body of f. Then compute the value of the resulting expression.

This is the ***call-by-name*** rule. It reduces $zero(sqr(sqr(sqr(2))))$ at once to 0. But it does badly by $sqr(sqr(sqr(2)))$. It duplicates the argument, $sqr(sqr(2))$. The result of this 'reduction' is

$$sqr(sqr(2)) \times sqr(sqr(2)).$$

This happens because $sqr(x) = x \times x$.

Multiplication, like other arithmetic operations, needs special treatment. It must be applied to values, not expressions: it is an example of a ***strict*** function. To evaluate $E_1 \times E_2$, the expressions E_1 and E_2 must be evaluated first.

Let us carry on with the evaluation. As the outermost function is $\times$, which is strict, the rule selects the leftmost call to sqr. Its argument is also duplicated:

$$(sqr(2) \times sqr(2)) \times sqr(sqr(2))$$

A full evaluation goes something like this.

$$\begin{aligned} sqr(sqr(sqr(2))) &\Rightarrow sqr(sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow (sqr(2) \times sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow ((2 \times 2) \times sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow (4 \times sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow (4 \times (2 \times 2)) \times sqr(sqr(2)) \\ &\ \vdots \end{aligned}$$

Does it ever reach the answer? Eventually. But call-by-name cannot be the evaluation rule we want.

The ***call-by-need*** rule (lazy evaluation) is like call-by-name, but ensures that each argument is evaluated at most once. Rather than substituting an expression into the function's body, the occurrences of the argument are linked by pointers. If the argument is ever evaluated, the value will be shared with its other occurrences. The pointer structure forms a directed graph of functions and arguments. As a part of the graph is evaluated, it is updated by the resulting value. This is called ***graph reduction***.

Figure 2.3 presents a graph reduction. Every step replaces an occurrence of $sqr(E)$ by $E \times E$, where the two Es are shared. There is no wasteful duplication: only three multiplications are performed. We seem to have the best of both worlds, for $zero(E)$ reduces immediately to 0. But the graph manipulations are expensive.

Figure 2.3 *Graph reduction of* $sqr(sqr(sqr(2)))$

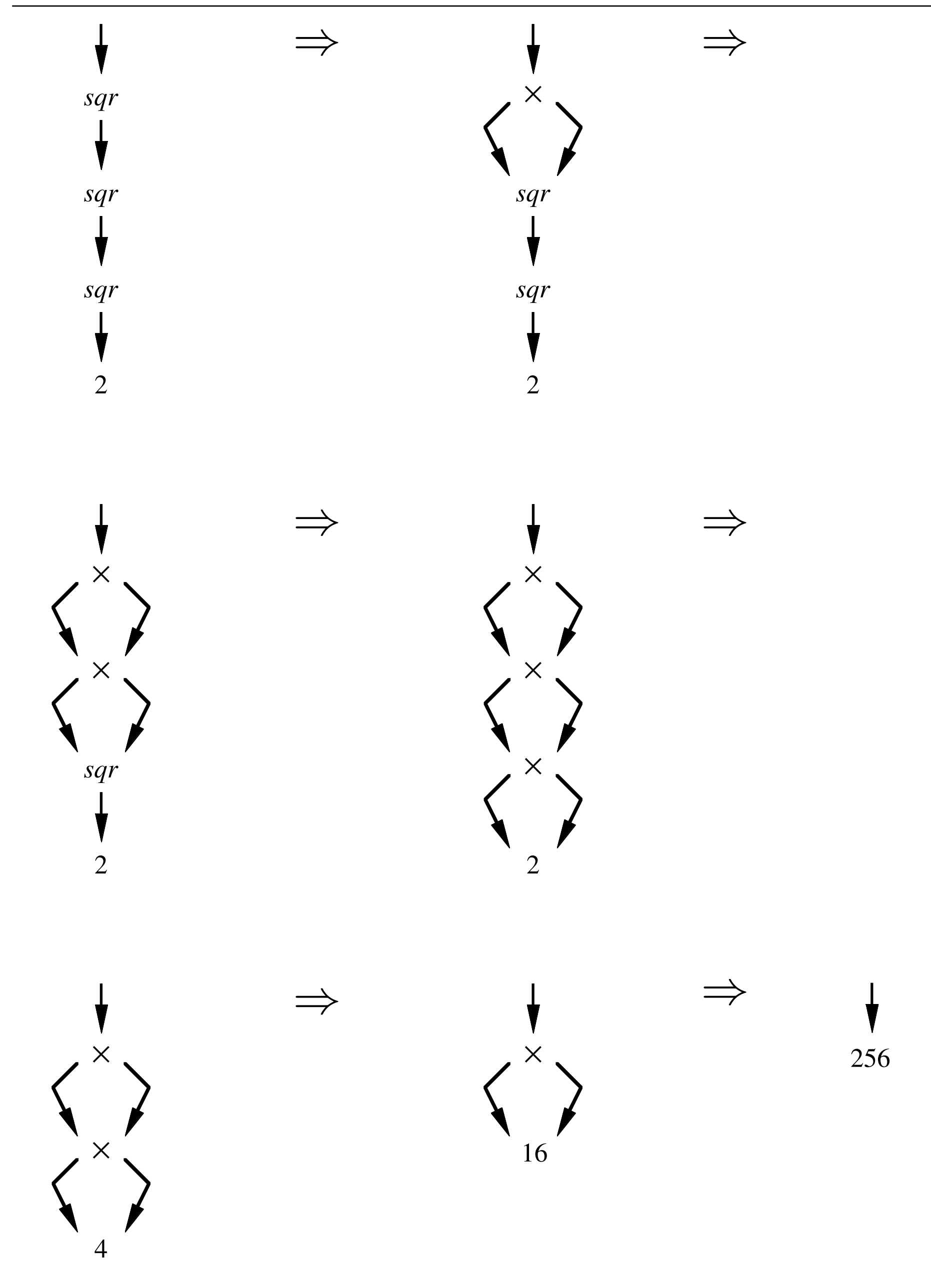

Figure 2.4 *A space leak with lazy evaluation*

$$
\begin{aligned}
facti(4, 1) &\Rightarrow facti(4-1, 4\times 1)\\
&\Rightarrow facti(3-1, 3\times(4\times 1))\\
&\Rightarrow facti(2-1, 2\times(3\times(4\times 1)))\\
&\Rightarrow facti(1-1, 1\times(2\times(3\times(4\times 1))))\\
&\Rightarrow 1\times(2\times(3\times(4\times 1)))\\
&\ \ \vdots\\
&\Rightarrow 24
\end{aligned}
$$

Lazy evaluation of $cond(E, E_1, E_2)$ behaves like a conditional expression provided that its argument, the tuple (E, E_1, E_2), is itself evaluated lazily. The details of this are quite subtle: tuple formation must be viewed as a function. The idea that a data structure like (E, E_1, E_2) can be partially evaluated — either E_1 or E_2 but not both — leads to infinite lists.

A comparison of strict and lazy evaluation. Call-by-need does the least possible evaluation. It may seem like the route to efficiency. But it requires much book-keeping. Realistic implementations became possible only after David Turner (1979) applied graph reduction to ***combinators***. He exploited obscure facts about the λ-calculus to develop new compilation techniques, which researchers continue to improve. Every new technology has its evangelists: some people are claiming that lazy evaluation is the way, the truth and the light. Why does Standard ML not adopt it?

Lazy evaluation says that $zero(E) = 0$ even if E fails to terminate. This flies in the face of mathematical tradition: an expression is meaningful only if all its parts are. Alonzo Church, the inventor of the λ-calculus, preferred a variant (the λI-calculus) banning constant functions like $zero$.

Infinite data structures complicate mathematical reasoning. To fully understand lazy evaluation, it is necessary to know some domain theory, as well as the theory of the λ-calculus. The output of a program is not simply a value, but a partially evaluated expression. These concepts are not easy to learn, and many of them are mechanistic. If we can only think in terms of the evaluation mechanism, we are no better off than the procedural programmers.

Efficiency is problematical too. Sometimes lazy evaluation saves enormous

amounts of space; sometimes it wastes space. Recall that $facti$ is more efficient than $fact$ under strict evaluation, performing each multiplication at once. Lazy evaluation of $facti(n, p)$ evaluates n immediately (for the test $n = 0$), but not p. The multiplications accumulate; we have a space leak (Figure 2.4).

Most lazy programming languages are purely functional. Can lazy evaluation be combined with commands, such as are used in ML to perform input/output? Subexpressions would be evaluated at unpredictable times; it would be impossible to write reliable programs. Much research has been directed at combining functional and imperative programming (Peyton Jones and Wadler, 1993).

Writing recursive functions

Since recursion is so fundamental to functional programming, let us take the time to examine a few recursive functions. There is no magic formula for program design, but perhaps it is possible to learn by example. One recursive function we have already seen implements Euclid's Algorithm:

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

The Greatest Common Divisor of two integers is by definition the greatest integer that divides both. Euclid's Algorithm is correct because the divisors of m and n are the same as those of m and $n - m$, and, by repeated subtraction, the same as the divisors of m and n mod m. Regarding its efficiency, consider

$$gcd(5499{,}6812) \Rightarrow gcd(1313, 5499) \Rightarrow gcd(247, 1313) \\ \Rightarrow gcd(78, 247) \Rightarrow gcd(13, 78) \Rightarrow gcd(0, 13) \Rightarrow 13.$$

Euclid's Algorithm dates from antiquity. We seldom can draw on 2000 years of expertise, but we should aim for equally elegant and efficient solutions.

Recursion involves reducing a problem to smaller subproblems. The key to efficiency is to select the right subproblems. There must not be too many of them, and the rest of the computation should be reasonably simple.

2.14 *Raising to an integer power*

The obvious way to compute x^k is to multiply repeatedly by x. Using recursion, the problem x^k is reduced to the subproblem x^{k-1}. But x^{10} need not involve 10 multiplications. We can compute x^5 and then square it. Since

$x^5 = x \times x^4$, we can compute x^4 by squaring also:

$$x^{10} = (x^5)^2 = (x \times x^4)^2 = (x \times (x^2)^2)^2$$

By exploiting the law $x^{2n} = (x^n)^2$ we have improved vastly over repeated multiplication. But the computation is still messy; using instead $x^{2n} = (x^2)^n$ eliminates the nested squaring:

$$2^{10} = 4^5 = 4 \times 16^2 = 4 \times 256^1 = 1024$$

By this approach, *power* computes x^k for real x and integer $k > 0$:

```
fun power(x,k) : real =
      if k=1 then x
      else if k mod 2 = 0 then     power(x*x, k div 2)
                          else x * power(x*x, k div 2);
> val power = fn : real * int -> real
```

Note how *mod* tests whether the exponent is even. Integer division (*div*) truncates its result to an integer if k is odd. The function *power* embodies the equations (for $n > 0$)

$$\begin{aligned} x^1 &= x \\ x^{2n} &= (x^2)^n \\ x^{2n+1} &= x \times (x^2)^n. \end{aligned}$$

We can test *power* using the built-in exponentiation function *Math.pow*:

```
power(2.0,10);
> 1024.0 : real
power(1.01, 925);
> 9937.353723 : real
Math.pow(1.01, 925.0);
> 9937.353723 : real
```

Reducing x^{2n} to $(x^2)^n$ instead of $(x^n)^2$ makes *power* iterative in its first recursive call. The second call (for odd exponents) can be made iterative only by introducing an argument to hold the result, which is a needless complication.

Exercise 2.12 Write the computation steps for $power(2.0, 29)$.

Exercise 2.13 How many multiplications does $power(x, k)$ need in the worst case?

Exercise 2.14 Why not take $k = 0$ for the base case instead of $k = 1$?

2.15 *Fibonacci numbers*

The Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., is popular with mathematical hobbyists because it enjoys many fascinating properties. The sequence (F_n) is defined by

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1}. && \text{for } n \geq 2 \end{aligned}$$

The corresponding recursive function is a standard benchmark for measuring the efficiency of compiled code! It is far too slow for any other use because it computes subproblems repeatedly. For example, since

$$F_8 = F_6 + F_7 = F_6 + (F_5 + F_6),$$

it computes F_6 twice.

Each Fibonacci number is the sum of the previous two:

$$0+1=1 \quad 1+1=2 \quad 1+2=3 \quad 2+3=5 \quad 3+5=8 \cdots$$

So we should compute with pairs of numbers. Function $nextfib$ takes (F_{n-1}, F_n) and returns the next pair (F_n, F_{n+1}).

```
fun nextfib(prev, curr :int) = (curr, prev+curr);
> val nextfib = fn : int * int -> int * int
```

The special name it, by referring to the previous pair, helps us demonstrate the function:

```
nextfib (0,1);
> (1, 1) : int * int
nextfib it;
> (1, 2) : int * int
nextfib it;
> (2, 3) : int * int
nextfib it;
> (3, 5) : int * int
```

Recursion applies $nextfib$ the requisite number of times:

```
fun fibpair (n) =
      if n=1 then (0,1) else nextfib(fibpair(n-1));
> val fibpair = fn : int -> int * int
```

It quickly computes (F_{29}, F_{30}), which previously would have required nearly three million function calls:

```
fibpair 30;
> (514229, 832040) : int * int
```

Let us consider in detail why *fibpair* is correct. Clearly $fibpair(1) = (F_0, F_1)$. And if, for $n \geq 1$, we have

$$fibpair(n) = (F_{n-1}, F_n),$$

then

$$fibpair(n+1) = (F_n, F_{n-1} + F_n) = (F_n, F_{n+1}).$$

We have just seen a proof of the formula $fibpair(n) = (F_{n-1}, F_n)$ by **mathematical induction**. We shall see many more examples of such proofs in Chapter 6. Proving properties of functional programs is often straightforward; this is one of the main advantages of functional languages.

The function *fibpair* uses a correct and fairly efficient algorithm for computing Fibonacci numbers, and it illustrates computing with pairs. But its pattern of recursion wastes space: *fibpair* builds the nest of calls

$$nextfib(nextfib(\cdots nextfib(0, 1)\cdots)).$$

To make the algorithm iterative, let us turn the computation inside out:

```
fun itfib (n, prev, curr) : int =
    if n=1 then curr       (*does not work for n=0*)
    else itfib (n-1, curr, prev+curr);
> val itfib = fn : int * int * int -> int
```

The function *fib* calls *itfib* with correct initial arguments:

```
fun fib (n) = itfib(n,0,1);
> val fib = fn : int -> int
fib 30;
> 832040 : int
fib 100;
> 354224848179261915075 : int
```

For Fibonacci numbers, iteration is clearer than recursion:

$$itfib(7, 0, 1) \Rightarrow itfib(6, 1, 1) \Rightarrow \cdots itfib(1, 8, 13) \Rightarrow 13$$

In Section 6.3 we shall show that *itfib* is correct by proving the rather unusual law $itfib(n, F_k, F_{k+1}) = F_{k+n}$.

Exercise 2.15 How is the repeated computation in the recursive definition of F_n related to the call-by-name rule? Could lazy evaluation execute this definition efficiently?

Exercise 2.16 Show that the number of steps needed to compute F_n by its recursive definition is exponential in n. How many steps does *fib* perform? Assume that call-by-value is used.

Exercise 2.17 What is the value of $itfib(n, F_{k-1}, F_k)$?

2.16 *Integer square roots*

The integer square root of n is the integer k such that

$$k^2 \leq n < (k+1)^2.$$

To compute this by recursion, we must choose a subproblem: an integer smaller than n. Division by 2 is often helpful, but how can we obtain $\sqrt{2x}$ from $\sqrt{x}$? Observe that $\sqrt{4x} = 2\sqrt{x}$ (for real x); division by 4 may lead to a simple algorithm.

Suppose $n > 0$. Since n may not be exactly divisible by 4, write $n = 4m + r$, where $r = 0, 1, 2,$ or 3. Since $m < n$ we can recursively find the integer square root of m:

$$i^2 \leq m < (i+1)^2.$$

Since m and i are integers, $m + 1 \leq (i+1)^2$. Multiplication by 4 implies $4i^2 \leq 4m$ and $4(m+1) \leq 4(i+1)^2$. Therefore

$$(2i)^2 \leq 4m \leq n < 4m + 4 \leq (2i+2)^2.$$

The square root of n is $2i$ or $2i+1$. There is only to test whether $(2i+1)^2 \leq n$, determining whether a 1 should be added.

```
fun increase(k,n) = if (k+1)*(k+1) > n then k else k+1;
> val increase = fn : int * int -> int
```

The recursion terminates when $n = 0$. Repeated integer division will reduce any number to 0 eventually:

```
fun introot n =
    if n=0 then 0 else increase(2 * introot(n div 4), n);
> val introot = fn : int -> int
```

There are faster methods of computing square roots, but ours is respectably fast and is a simple demonstration of recursion.

```
introot 123456789;
> 11111 : int
it*it;
> 123454321 : int
introot 2000000000000000000000000000000;
```

```
> 1414213562373095 : int
it*it;
> 1999999999999999861967979879025 : int
```

Exercise 2.18 Code this integer square root algorithm using iteration in a procedural programming language.

Exercise 2.19 Declare an ML function for computing the Greatest Common Divisor, based on these equations (m and n range over positive integers):

$$\begin{aligned} GCD(2m, 2n) &= 2 \times GCD(m, n) \\ GCD(2m, 2n+1) &= GCD(m, 2n+1) \\ GCD(2m+1, 2n+1) &= GCD(n-m, 2m+1) \qquad m < n \\ GCD(m, m) &= m. \end{aligned}$$

How does this compare with Euclid's Algorithm?

Local declarations

Reducing the fraction n/d to least terms, where n and d have no common factor, involves dividing both numbers by their GCD.

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

The wasteful re-computation of $gcd(n, d)$ can be prevented by first defining an auxiliary function:

```
fun divideboth (n, d, com: int) = (n div com, d div com);
fun fraction (n,d) = divideboth (n, d, gcd(n,d));
```

But this is a contorted way of giving $gcd(n, d)$ the name com. ML allows the declaration of names within an expression:

```
fun fraction (n,d) =
    let val com = gcd(n,d)
    in  (n div com, d div com)  end;
> val fraction = fn : int * int -> int * int
```

We have used a `let` expression, which has the general form

```
let D in E end
```

During evaluation, the declaration D is evaluated first: expressions within the declaration are evaluated, and their results given names. The environment thus

created is visible only inside the `let` expression. Then the expression E is evaluated, and its value returned.

Typically D is a compound declaration, which consists of a list of declarations:

$$D_1;\, D_2;\, \ldots;\, D_n$$

The effect of each declaration is visible in subsequent ones. The semicolons are optional and many programmers omit them.

2.17 *Example: real square roots*

The Newton-Raphson method finds roots of a function: in other words, it solves equations of the form $f(x) = 0$. Given a good initial approximation, it converges rapidly. It is highly effective for computing square roots, solving the equation $a - x^2 = 0$. To compute $\sqrt{a}$, choose any positive x_0, 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.

The function *findroot* performs this computation, where x approximates the square root of a and acc is the desired accuracy (relative to x). Since the next approximation is used several times, it is given the name $nextx$ using `let`.

```
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;
> val findroot = fn : (real * real * real) -> real
```

The function $sqroot$ calls *findroot* with suitable starting values.

```
fun sqroot a = findroot (a, 1.0, 1.0E~10);
> val sqroot = fn : real -> real
sqroot 2.0;
> 1.414213562 : real
it*it;
> 2.0 : real
```

Nested function declarations. Our square root function is still not ideal. The arguments a and acc are passed unchanged in every recursive call of *findroot*. They can be made global to *findroot* for efficiency and clarity.

A further `let` declaration nests *findroot* within $sqroot$. The accuracy acc is declared first, to be visible in *findroot*; the argument a is also visible.

```
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;
> val sqroot = fn : real -> real
```

As we see from ML's response, *findroot* is not visible outside *sqroot*.

Most kinds of declaration are permitted within `let`. Values, functions, types and exceptions may be declared.

When not to use `let`. Consider taking the minimum of $f(x)$ and $g(x)$. You could name these quantities using `let`:

```
let val a = f x
    val b = g x
in
    if a<b then a else b
end
```

Better, declare a function for the minimum of two real numbers:

```
fun min(a,b) : real =  if a<b then a else b;
```

Now `min(f x, g x)` is clear because *min* computes something familiar. Take every opportunity to declare meaningful functions, even if they are only needed once.

2.18 *Hiding declarations using* `local`

A `local` declaration resembles a `let` expression:

```
local D1 in D2 end
```

This declaration behaves like the list of declarations D_1; D_2 except that D_1 is visible only within D_2, not outside. Since a list of declarations is regarded as one declaration, both D_1 and D_2 can declare any number of names.

While `let` is frequently used, `local` is not. Its sole purpose is to hide a declaration. Recall *itfib* and *fib*, which compute Fibonacci numbers. The function *itfib* should be called only from *fib*:

```
local
    fun itfib (n, prev, curr) : int =
          if n=1 then curr
          else itfib (n-1, curr, prev+curr)
in
```

```
    fun fib (n) = itfib(n,0,1)
end;
> val fib = fn : int -> int
```

Here the `local` declaration makes *itfib* private to *fib*.

Exercise 2.20 Above we have used `local` to hide the function *itfib*. Why not simply nest the declaration of *itfib* within *fib*? Compare with the treatment of *findroot* and *sqroot*.

Exercise 2.21 Using `let`, we can eliminate the expensive squaring operation in our integer square root function. Code a variant of *introot* that maps n to its integer square root k, paired with the difference $n - k^2$. Only simple multiplications and divisions are needed; an optimizing compiler could replace them by bit operations.

2.19 *Simultaneous declarations*

A simultaneous declaration defines several names at once. Normally the declarations are independent. But `fun` declarations allow recursion, so a simultaneous declaration can introduce mutually recursive functions.

A `val` declaration of the form

```
val Id1 = E1 and ··· and Idn = En
```

evaluates the expressions $E_1, \ldots, E_n$ and then declares the identifiers $Id_1, \ldots, Id_n$ to have the corresponding values. Since the declarations do not take effect until all the expressions are evaluated, their order is immaterial.

Here we declare names for π, e and the logarithm of 2.

```
val pi   = 4.0 * Math.atan 1.0
and e    = Math.exp 1.0
and log2 = Math.ln 2.0;
> pi = 3.141592654 : real
> e = 2.718281828 : real
> log2 = 0.693147806 : real
```

A single input declares three names. The simultaneous declaration emphasizes that they are independent.

Now let us declare the chimes of Big Ben:

```
val one = "BONG ";
> val one = "BONG " : string
val three = one^one^one;
> val three = "BONG BONG BONG " : string
val five = three^one^one;
```

```
> val five = "BONG BONG BONG BONG BONG " : string
```

There must be three separate declarations, and in this order.

A simultaneous declaration can also swap the values of names:

```
val one = three and three = one;
> val one = "BONG BONG BONG " : string
> val three = "BONG " : string
```

This is, of course, a silly thing to do! But it illustrates that the declarations occur at the same time. Consecutive declarations would give *one* and *three* identical bindings.

Mutually recursive functions. Several functions are ***mutually recursive*** if they are declared recursively in terms of each other. A recursive descent parser is a typical case. This sort of parser has one function for each element of the grammar, and most grammars are mutually recursive: an ML declaration can contain expressions, while an expression can contain declarations. Functions to traverse the resulting parse tree will also be mutually recursive.

Parsing and trees are discussed later in this book. For a simpler example, consider summing the series

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots + \frac{1}{4k+1} - \frac{1}{4k+3} \cdots$$

By mutual recursion, the final term of the summation can be either positive or negative:

```
fun pos d = neg(d-2.0) + 1.0/d
and neg d = if d>0.0 then pos(d-2.0) - 1.0/d
                     else 0.0;
> val pos = fn : real -> real
> val neg = fn : real -> real
```

Two functions are declared. The series converges leisurely:

```
4.0 * pos(201.0);
> 3.151493401
4.0 * neg(8003.0);
> 3.141342779
```

Mutually recursive functions can often be combined into one function with the help of an additional argument:

```
fun sum(d,one) =
```

```
    if d>0.0 then sum(d-2.0, ~one) + one/d else 0.0;
```

Now sum(d,1.0) returns the same value as pos(d), and sum(d,~1.0) returns the same value as
neg(d).

Emulating goto statements. Functional programming and procedural programming are more alike than you may imagine. Any combination of goto and assignment statements — the worst of procedural code — can be translated into a set of mutually recursive functions. Here is a simple case:

```
var x := 0;  y := 0;  z := 0;
F:  x := x+1; goto G
G:  if y<z then goto F  else  (y := x+y;  goto H)
H:  if z>0 then (z := z-x;  goto F)  else  stop
```

For each of the labels, F, G and H, declare mutually recursive functions. The argument of each function is a tuple holding all of the variables.

```
fun F(x,y,z)  = G(x+1,y,z)
and G(x,y,z)  = if y<z then F(x,y,z)  else  H(x,x+y,z)
and H(x,y,z)  = if z>0 then F(x,y,z-x)  else  (x,y,z);
> val F = fn : int * int * int -> int * int * int
> val G = fn : int * int * int -> int * int * int
> val H = fn : int * int * int -> int * int * int
```

Calling $f(0, 0, 0)$ gives x, y and z their initial values for execution, and returns the result of the procedural code.

```
f(0,0,0);
> (1, 1, 0) : int * int * int
```

Functional programs are referentially transparent, yet can be totally opaque. If your code starts to look like this, beware!

Exercise 2.22 What is the effect of this declaration?

```
val (pi,log2) = (log2,pi);
```

Exercise 2.23 Consider the sequence (P_n) defined for $n \geq 1$ by

$$P_n = 1 + \sum_{k=1}^{n-1} P_k.$$

(In particular, $P_1 = 1$.) Express this computation as an ML function. How efficient is it? Is there a faster way of computing P_n?

Introduction to modules

An engineer understands a device in terms of its component parts, and those, similarly, in terms of their subcomponents. A bicycle has wheels; a wheel has a hub; a hub has bearings, and so forth. It takes several stages before we reach the level of individual pieces of metal and plastic. In this way one can understand the entire bike at an abstract level, or parts of it in detail. The engineer can improve the design by modifying one part, often without thinking about the other parts.

Programs (which are more complicated than bicycles!) should also be seen as consisting of components. Traditionally, a subprogram is a procedure or function, but these are too small — it is like regarding the bicycle as composed of thousands of metal shapes. Many recent languages regard programs as consisting of ***modules***, each of which defines its own data structures and associated operations. The interface to each module is specified separately from the module itself. Different modules can therefore be coded by different members of a project team; the compiler can check that each module meets its interface specification.

Consider our vector example. The function $addvec$ is useless in isolation; it must be used together with other vector operations, all sharing the same representation of vectors. We can guess that the other operations are related because their names all end with vec, but nothing enforces this naming convention. They should be combined together to form a program module.

An ML ***structure*** combines related types, values and other structures, with a uniform naming discipline. An ML ***signature*** specifies a class of structures by listing the name and type (or other attributes) of each component.

Standard ML's signatures and structures have analogues in other languages, such as Modula-2's definition and implementation modules (Wirth, 1985). ML also provides ***functors*** — structures taking other structures as parameters — but we shall defer these until Chapter 7.

2.20 *The complex numbers*

Many types of mathematical object can be added, subtracted, multiplied and divided. Besides the familiar integer and real numbers, there are the rational numbers, matrices, polynomials, etc. Our example below will be the complex numbers, which are important in scientific mathematics. We shall gather up their arithmetic operations using a structure $Complex$, then declare a signature for $Complex$ that also matches any structure that defines the same arithmetic operations. This will provide a basis for generic arithmetic.

We start with a quick introduction to the complex numbers. A ***complex num-***

ber has the form $x + iy$, where x and y are real numbers and i is a constant postulated to satisfy $i^2 = -1$. Thus, x and y determine the complex number.

The complex number zero is $0 + i0$. The sum of two complex numbers consists of the sums of the x and y parts; the difference is similar. The definitions of product and reciprocal look complicated, but are easy to justify using algebraic laws and the axiom $i^2 = -1$:

$$
\begin{aligned}
(x + iy) + (x' + iy') &= (x + x') + i(y + y') \\
(x + iy) - (x' + iy') &= (x - x') + i(y - y') \\
(x + iy) \times (x' + iy') &= (xx' - yy') + i(xy' + x'y) \\
1/(x + iy) &= (x - iy)/(x^2 + y^2)
\end{aligned}
$$

In the reciprocal above, the y component is $-y/(x^2 + y^2)$. We can now define the complex quotient z/z' as $z \times (1/z')$.

By analogy with our vector example, we could implement the complex numbers by definitions such as

```
type complex = real*real;
val complexzero = (0.0, 0.0);
⋮
```

but it is better to use a structure.

Further reading. Penrose (1989) explains the complex number system in more detail, with plenty of motivation and examples. He discusses the connections between the complex numbers and fractals, including a definition of the Mandelbrot set. Later in the book, the complex numbers play a central rôle in his discussion of quantum mechanics. Penrose gives the complex numbers a metaphysical significance; that might be taken with a pinch of salt! Feynman *et al.* (1963) give a more technical but marvellously enjoyable description of the complex numbers in Chapter 22.

2.21 *Structures*

Declarations can be grouped to form a structure by enclosing them in the keywords `struct` and `end`. The result can be bound to an ML identifier using a `structure` declaration:

```
structure Complex =
  struct
  type t    = real*real;
  val zero  = (0.0, 0.0);
  fun sum   ((x,y), (x',y')) = (x+x', y+y') : t;
  fun diff  ((x,y), (x',y')) = (x-x', y-y') : t;
  fun prod  ((x,y), (x',y')) = (x*x' - y*y', x*y' + x'*y) : t;
  fun recip (x,y)            =
                               let val t = x*x + y*y
                               in (x/t, ~y/t) end
  fun quo   (z,z')           = prod(z, recip z');
  end;
```

Where structure $Complex$ is visible, its components are known by compound names such as $Complex\,.\,zero$ and $Complex\,.\,sum$. Inside the structure body, the components are known by their ordinary identifiers, such as $zero$ and sum; note the use of $recip$ in the declaration of quo. The type of complex numbers is called $Complex\,.\,t$. When the purpose of a structure is to define a type, that type is commonly called t.

We may safely use short names. They cannot clash with names occurring in other structures. The standard library exploits this heavily, for example to distinguish the absolute value functions *Int*.*abs* and *Real*.*abs*.

Let us experiment with our new structure. We declare two ML identifiers, i and a; a mathematician would normally write their values as i and 0.3, respectively.

```
val i = (0.0, 1.0);
> val i = (0.0, 1.0) : real * real
val a = (0.3, 0.0);
> val a = (0.3, 0.0) : real * real
```

In two steps we form the sum $a + i + 0.7$, which equals $1 + i$. Finally we square that number to obtain $2i$:

```
val b = Complex.sum(a,i);
> val b = (0.3, 1.0) : Complex.t
Complex.sum(b, (0.7, 0.0));
> (1.0, 1.0) : Complex.t
Complex.prod(it,it);
> (0.0, 2.0) : Complex.t
```

Observe that $Complex\,.\,t$ is the same type as $real \times real$; what is more confusing, it is the same type as vec above. Chapter 7 describes how to declare an ***abstract type***, whose internal representation is hidden.

Structures look a bit like records, but there are major differences. A record's components can only be values (including, perhaps, other records). A structure's

components may include types and exceptions (as well as other structures). But you cannot compute with structures: they can only be created when the program modules are being linked together. Structures should be seen as encapsulated environments.

2.22 *Signatures*

A signature is a description of each component of a structure. ML responds to our declaration of the structure *Complex* by printing its view of the corresponding signature:

```
structure Complex = ...;
> structure Complex :
>   sig
>   type t
>   val diff  : (real * real) * (real * real) -> t
>   val prod  : (real * real) * (real * real) -> t
>   val quo   : (real * real) * (real * real) -> t
>   val recip : real * real -> real * real
>   val sum   : (real * real) * (real * real) -> t
>   val zero  : real * real
>   end
```

The keywords `sig` and `end` enclose the signature body. It shows the types of all the components that are values, and mentions the type t. (Some compilers display `eqtype t` instead of `type t`, informing us that t is a so-called equality type.)

The signature inferred by the ML compiler is frequently not the best one for our purposes. The structure may contain definitions that ought to be kept private. By declaring our own signature and omitting the private names, we can hide them from users of the structure. We might, for instance, hide the name *recip*.

The signature printed above expresses the type of complex numbers sometimes as t and sometimes as $real \times real$. If we use t everywhere, then we obtain a general signature that specifies a type t equipped with operators *sum*, *prod*, etc.:

```
signature ARITH =
   sig
   type t
   val zero : t
   val sum  : t * t -> t
   val diff : t * t -> t
   val prod : t * t -> t
   val quo  : t * t -> t
   end;
```

The declaration gives the name *ARITH* to the signature enclosed within the brackets `sig` and `end`. We can declare other structures and make them conform to signature *ARITH*. Here is the skeleton of a structure for the rational numbers:

```
structure Rational : ARITH =
  struct
  type t    = int*int;
  val zero = (0, 1);
  ⋮
  end;
```

A signature specifies the information that ML needs to integrate program units safely. It cannot specify what the components actually do. A well-documented signature includes comments describing the purpose of each component. Comments describing a component's implementation belong in the structure, not in the signature. Signatures can be combined in various ways to form new signatures; structures can similarly be combined.

ML functors can express generic modules: for example, ones that take any structure conforming to signature *ARITH*. The standard library offers extensive possibilities for this. An ML system may provide floating point numbers in various precisions, as structures matching signature *FLOAT*. A numerical algorithm can be coded as a functor. Applying the functor to a floating point structure specializes the algorithm to the desired precision. ML thus has some of the power of object-oriented languages such as C++ — though in a more restrictive form, since structures are not computable values.

Exercise 2.24 Declare a structure $Real$, matching signature *ARITH*, such that $Real.t$ is the type $real$ and the components $zero$, sum, $prod$, etc., denote the corresponding operations on type $real$.

Exercise 2.25 Complete the declaration of structure $Rational$ above, basing your definitions on the laws $n/d+n'/d' = (nd'+n'd)/dd'$, $(n/d)\times(n'/d') = nn'/dd'$, and $1/(n/d) = d/n$. Use the function gcd to maintain the fractions in lowest terms, and ensure that the denominator is always positive.

Polymorphic type checking

Until recently, the debate on type checking has been deadlocked, with two rigid positions:

- Weakly typed languages like Lisp and Prolog give programmers the freedom they need when writing large programs.
- Strongly typed languages like Pascal give programmers security by restricting their freedom to make mistakes.

Polymorphic type checking offers a new position: the security of strong type checking, as well as great flexibility. Programs are not cluttered with type specifications since most type information is deduced automatically.

A type denotes a collection of values. A function's argument type specifies which values are acceptable as arguments. The result type specifies which values could be returned as results. Thus, *div* demands a pair of integers as argument; its result can only be an integer. If the divisor is zero then there will be no result at all: an error will be signalled instead. Even in this exceptional situation, the function *div* is faithful to its type.

ML can also assign a type to the identity function, which returns its argument unchanged. Because the identity function can be applied to an argument of any type, it is ***polymorphic***. Generally speaking, an object is polymorphic if it can be regarded as having multiple types. ML polymorphism is based on ***type schemes***, which are like patterns or templates for types. For instance, the identity function has the type scheme $\alpha \to \alpha$.

2.23 *Type inference*

Given little or no explicit type information, ML can infer all the types involved with a function declaration. Type inference follows a natural but rigorous procedure. ML notes the types of any constants, and applies type checking rules for each form of expression. Each variable must have the same type everywhere in the declaration. The type of each overloaded operator (like +) must be determined from the context.

Here is the type checking rule for the conditional expression. If E has type *bool* and E_1 and E_2 have the same type, say τ, then

```
if E then E1 else E2
```

also has type τ. Otherwise, the expression is ill-typed.

Let us examine, step by step, the type checking of *facti*:

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

The constants 0 and 1 have type *int*. Therefore n=0 and n-1 involve integers, so n has type *int*. Now $n*p$ must be integer multiplication, so p has type *int*.

Since p is returned as the result of $facti$, its result type is int and its argument type is $int \times int$. This fits with the recursive call. Having made all these checks, ML can respond

```
> val facti = fn : int * int -> int
```

If the types are not consistent, the compiler rejects the declaration.

Exercise 2.26 Describe the steps in the type checking of $itfib$.

Exercise 2.27 Type check the following function declaration:

```
fun f (k,m) = if k=0 then 1 else f(k-1);
```

2.24 *Polymorphic function declarations*

If type inference leaves some types completely unconstrained then the declaration is polymorphic — literally, 'having many forms.' Most polymorphic functions involve pairs, lists and other data structures. They usually do something simple, like pairing a value with itself:

```
fun pairself x = (x,x);
> val pairself = fn : 'a -> 'a * 'a
```

This type is polymorphic because it contains a ***type variable***, namely `'a`. In ML, type variables begin with a prime (single quote) character.

```
'b      'c      'we_band_of_brothers      '3
```

Let us write α, β, γ for the ML type variables `'a`, `'b`, `'c`, because type variables are traditionally Greek letters. Write $x : \tau$ to mean 'x has type τ,' for instance $pairself : \alpha \to (\alpha \times \alpha)$. Incidentally, $\times$ has higher precedence than $\to$; the type of $pairself$ can be written $\alpha \to \alpha \times \alpha$.

A polymorphic type is a type scheme. Substituting types for type variables forms an ***instance*** of the scheme. A value whose type is polymorphic has infinitely many types. When $pairself$ is applied to a real number, it effectively has type $real \to real \times real$.

```
pairself 4.0;
> (4.0, 4.0) : real * real
```

Applied to an integer, $pairself$ effectively has type $int \to int \times int$.

```
pairself 7;
> (7, 7) : int * int
```

Here $pairself$ is applied to a pair; the result is called pp.

```
val pp = pairself ("Help!",999);
> val pp = (("Help!", 999), ("Help!", 999))
> : (string * int) * (string * int)
```

Projection functions return a component of a pair. The function *fst* returns the first component; *snd* returns the second:

```
fun fst (x,y) = x;
> val fst = fn : 'a * 'b -> 'a
fun snd (x,y) = y;
> val snd = fn : 'a * 'b -> 'b
```

Before considering their polymorphic types, we apply them to *pp*:

```
fst pp;
> ("Help!", 999) : string * int
snd(fst pp);
> 999 : int
```

The type of *fst* is $\alpha \times \beta \to \alpha$, with two type variables. The argument pair may involve any two types τ_1 and τ_2 (not necessarily different); the result has type τ_1.

Polymorphic functions can express other functions. The function that takes $((x, y), w)$ to x could be coded directly, but two applications of *fst* also work:

```
fun fstfst z = fst(fst z);
> val fstfst = fn : ('a * 'b) * 'c -> 'a
fstfst pp;
> "Help!" : string
```

The type $(\alpha \times \beta) \times \gamma \to \alpha$ is what we should expect for *fstfst*. Note that a polymorphic function can have different types within the same expression. The inner *fst* has type $(\alpha \times \beta) \times \gamma \to \alpha \times \beta$; the outer *fst* has type $\alpha \times \beta \to \alpha$.

Now for something obscure: what does this function do?

```
fun silly x = fstfst(pairself(pairself x));
> val silly = fn : 'a -> 'a
```

Not very much:

```
silly "Hold off your hands.";
> "Hold off your hands." : string
```

Its type, $\alpha \to \alpha$, suggests that *silly* is the identity function. This function can be expressed rather more directly:

```
fun I x = x;
> val I = fn : 'a -> 'a
```

Further issues. Milner (1978) gives an algorithm for polymorphic type checking and proves that a type-correct program cannot suffer a run-time type error.

Damas and Milner (1982) prove that the types inferred by this algorithm are ***principal***: they are as polymorphic as possible. Cardelli and Wegner (1985) survey several approaches to polymorphism. For Standard ML, things are quite complicated.

Equality testing is polymorphic in a limited sense: it is defined for most, not all, types. Standard ML provides a class of ***equality type variables*** to range over this restricted collection of types. See Section 3.14.

Recall that certain built-in functions are overloaded: addition (+) is defined for integers and reals, for instance. Overloading sits uneasily with polymorphism. It complicates the type checking algorithm and frequently forces programmers to write type constraints. Fortunately there are only a few overloaded functions. Programmers cannot introduce further overloading.

Summary of main points

- A variable stands for a value; it can be redeclared but not updated.
- Basic values have type int, $real$, $char$, $string$ or $bool$.
- Values of any types can be combined to form tuples and records.
- Numerical operations can be expressed as recursive functions.
- An iterative function employs recursion in a limited fashion, where recursive calls are essentially jumps.
- Structures and signatures serve to organize large programs.
- A polymorphic type is a scheme containing type variables.