



MODERN PROGRAMMING LANGUAGES

A PRACTICAL INTRODUCTION

SECOND EDITION

Adam Brooks Webber

Franklin, Beedle & Associates Inc.
22462 SW Washington St.
Sherwood, Oregon 97140
503/625-4445
www.fbeedle.com



Chapter 5

A First Look at ML

5.1 Introduction

This chapter is an introduction to Standard ML. The ML language family has a number of dialects. The Standard ML dialect is the one used in this book; from here on it will just be referred to as ML. ML is one of the more popular functional languages, and some large commercial projects have been developed in ML. But let's be honest about this: you will probably never see an employer advertising for someone with ML skills. The point of learning ML is not to beef up your résumé, but to expand your programming language consciousness. By learning ML you will gain a new perspective—it is very different from the usual crowd of popular imperative languages. And who knows? You may find that ML is just the right language for you, and you may even choose to use it in commercial projects of your own.

This is a hands-on chapter. There are many short examples of ML. You may find it helpful to type in the examples as you go. You should do as many of the exercises as you can. By the end of the chapter, you should be able to

write simple expressions and function definitions in ML and use several ML types, including tuples and lists.

5.2 Getting Started with an ML Language System

You will need an ML language system to try the examples and solve the exercises. The examples in this book were produced using SML/NJ (Standard ML of New Jersey), which is an excellent, free, open-source ML language system. If you are reading this book as part of an organized course, your teacher may give you instructions for running ML on your local systems. If not, or if you want your own copy, you can easily download and install SML/NJ on your own Unix or Windows system. The Web site for this book has up-to-date links for downloading SML/NJ.

Like most functional-language systems, SML/NJ operates in an interactive mode: it prompts you to type in an expression, you type one in, it evaluates your expression, it prints out the value, and then the whole cycle repeats. When we first run SML/NJ on our system it prints this:

```
Standard ML of New Jersey1
-
```

The dash is its prompt; it is now waiting for input. Below, we have added to the session by typing an ML expression, followed by a semicolon, followed by the Enter key. (To be easier to read, the input is shown in boldface to distinguish it from ML's output. And the Enter key is not shown—that is assumed at the end of every input line.)

```
Standard ML of New Jersey
- 1 + 2 * 3;
val it = 7 : int
-
```

ML has evaluated the expression, printed the result, and prompted for another expression.

ML's response is probably more verbose than you expected. The value of $1+2*3$ should be 7, of course, but ML printed more than that. For one thing, it printed the type (`int`) as well. As we will see, ML tries to infer a type for every expression. Of course, anyone could figure out what the type of $1+2*3$ must be, but ML's type

1. It also prints the version number and build-date information on this line. This may be different on your system. The examples in this book were tested with version 110.70.

system is very expressive and its type inference is unusually powerful. You might have wondered why it prints `val it = 7` instead of just `7`. Part of the explanation is that ML maintains a variable named `it` whose value is always the value of the last expression that was typed in. The rest of the explanation will have to wait until we see a few more things about ML.

The semicolon at the end of the line of input is very important. It is very easy to forget when you are first beginning to experiment with ML. In case you should forget it, here is what will happen:

```
- 1 + 2 * 3
=
```

ML assumes that you are not yet finished with the expression you want it to evaluate. (Expressions can take up more than one line.) It prompts for further input with the character `=`. Eventually, when the input ends with the semicolon ML is waiting for, ML will evaluate the whole thing:

```
- 1 + 2 * 3
= ;
val it = 7 : int
=
```

5.3 Constants

Let's start by looking at the simplest of expressions—constants. The example below shows ML evaluating some numeric constants. These constants illustrate the two numeric types, `int` and `real`.

```
- 1234;
val it = 1234 : int
- 123.4;
val it = 123.4 : real
```

The syntax for integer and real constants is conventional, with one important wrinkle: ML uses the tilde symbol (`~`) for the negation operator. So the number `-1` is written as `~1` in ML.

The next example shows the ML constants `true` and `false`, which are the two values of ML's `bool` type.

```
- true;
val it = true : bool
- false;
val it = false : bool
```

There is not much else to say about them, but this is a good time to point out that ML is case sensitive. For the boolean true value, you must write `true`, not `TRUE` or `True` or anything else.

Now for some strings and characters (the ML types `string` and `char`):

```
- "fred";
val it = "fred" : string
- "H";
val it = "H" : string
- #"H";
val it = #"H" : char
```

String constants, like `"fred"`, are enclosed in double quotation marks (that is, with one double-quote character before and after—not two single-quote characters!). To enable unusual characters inside a string, ML supports the same kind of escape sequences that Java and C do; for example, `\t` for a tab, `\n` for a linefeed, or `\"` to put in a quote mark without ending the string. As you can see from the example, there is a difference between a one-character string constant and a character constant. To get a character constant, put the `#` symbol before the quoted character.

5.4 Operators

The next example shows ML's basic arithmetic operators.

```
- ~ 1 + 2 - 3 * 4 div 5 mod 6;
val it = ~1 : int
- ~ 1.0 + 2.0 - 3.0 * 4.0 / 5.0;
val it = ~1.4 : real
```

The integer binary operators for addition (+), subtraction (-), and multiplication (*) are standard. For integer division, ML provides two operators, written as `div` and `mod`. The `div` operator computes the integer quotient (ignoring the remainder), while `mod` returns the remainder after integer division.² As with constants, ML uses the tilde symbol for the negation operator. It takes a little practice to get used to this, since most languages use the minus-sign symbol both as a unary operator (for negation) and as a binary operator (for subtraction).

For real numbers, ML uses the same operators, +, -, *, and ~. There is also a real-division operator (/). For strings, there is a concatenation operator (^):

2. The concept of integer division with a remainder appears simple enough at first glance, but it is actually subject to a variety of interpretations. What happens with negative operands? Is 5 divided by -2 equal to -2, remainder 1, or to -3, remainder -1? In Java this works one way, and in ML it works the other way. Some languages (like Ada and Prolog) have different remainder-like operators so that they can do it both ways; other languages (like C++) just leave it up to each implementation to decide.

```
- "bibity" ^ "bobity" ^ "boo";
val it = "bibitybobityboo" : string
```

Then there are the ordering comparison operators: less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). They can be applied to pairs of strings, characters, integers, or real numbers.

```
- 2 < 3;
val it = true : bool
- 1.0 <= 1.0;
val it = true : bool
- #"d" > #"c";
val it = true : bool
- "abce" >= "abd";
val it = false : bool
```

Applied to strings, the comparisons test alphabetical order. Since "abce" would come before "abd" in the dictionary, the expression "abce" >= "abd" is false.

The two more fundamental comparison operators are the equality test (=) and the inequality test (<>). Some, but not all, of ML's types can be tested for equality or inequality using those operators; these are called the *equality types*. All the types we have seen so far are equality types except the real numbers. Real numbers cannot be tested for equality in ML. The reason is that most real arithmetic in computers is rounded to the limited precision of the computer hardware. Because of this rounding, two computations that should produce equal values mathematically often produce slightly different values on the computer. This means that it is usually a mistake (in any programming language) to compare two real numbers to see if they are exactly equal. This mistake is more difficult to make in ML, since ML does not allow you to compare real values for equality directly. (If you are really sure you want to, you can still accomplish it indirectly by combining a <= test with a >= test.)

For boolean values, ML has operators for logical or (*orelse*), logical and (*andalso*), and logical complement (*not*). For example:

```
- 1 < 2 orelse 3 > 4;
val it = true : bool
- 1 < 2 andalso not (3 < 4);
val it = false : bool
```

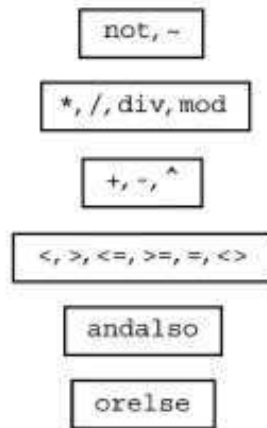
The *orelse* and *andalso* operators do not evaluate the second operand if the first one is enough to decide the result. If the first operand of an *orelse* is true, the whole result is true, so ML does not bother to evaluate the second operand. This is more than just an optimization. It is easy to write a program to test whether both

operands are evaluated or not; just make the second operand something whose evaluation would cause an error and then see if the error occurs.

```
- true orelse 1 div 0 = 0;
val it = true : bool
```

Evaluating the expression `1 div 0` should cause an error, but no error occurs because the expression is not evaluated. Operators like this are called *short-circuiting* operators. (To be perfectly accurate, `orelse` and `andalso` are not really operators in ML. They are just keywords. All true ML operators evaluate all their operands. But to keep things simple, we will continue to call them operators.)

The operators seen so far are all left-associative and fall into these six precedence levels:



ML has additional operators and additional precedence levels, and it allows programs to define new operators and specify their precedence. However, this book will not require any further operators.

5.5 Conditional Expressions

If the language you know best is an imperative language, you are probably already familiar with if-then and if-then-else statements. However, you may not have used an if-then-else *expression* like ML's conditional:

```
- if 1 < 2 then #"x" else #"y";
val it = #"x" : char
- if 1 > 2 then 34 else 56;
val it = 56 : int
- (if 1 < 2 then 34 else 56) + 1;
val it = 35 : int
```

A conditional expression has this syntax:

```
<conditional-expression> ::=
    if <expression> then <expression> else <expression>
```

The *<expression>* in the *if* part must have the type `bool`, and the *<expression>* in the *then* part must have the same type as the *<expression>* in the *else* part. If the *<expression>* in the *if* part is true, the *<expression>* in the *then* part is evaluated and gives the value for the whole *<conditional-expression>*. Otherwise, the *<expression>* in the *else* part is evaluated and gives the value for the whole *<conditional-expression>*. Like the `orelse` and `andalso` operators described above, the conditional expression is short-circuiting. The only part evaluated is the one actually needed.

5.6 Type Conversion and Function Application

Here is an example of a type error in ML:

```
- 1 * 2;
val it = 2 : int
- 1.0 * 2.0;
val it = 2.0 : real
- 1.0 * 2;
Error: operator and operand don't agree [literal]
operator domain: real * real
operand:          real * int
in expression:
  1.0 * 2
```

The first two expressions evaluated correctly. We have already seen that the `*` operator, and others like `+` and `<`, work on different types of pairs. When the same operator works differently on different types of operands, it is said to be *overloaded*. Chapter 8 will discuss overloading further. The `+` operator has one definition that applies to a pair of `int` values and another that applies to a pair of `real` values. But it does not have a definition that applies if the first operand is a `real` and the second is an `int`, so the third expression in the example above causes a type error.

In many languages, including Java, a mixed-type expression like `1.0*2` would be handled without error by converting the integer operand to a real number before multiplying. ML does not work this way. It has predefined functions that a program can use to convert values from one type to another, but it never does such conversions automatically.

Here are some of ML's predefined conversion functions:

Function	Parameter Type	Result Type	Notes
<code>real</code>	<code>int</code>	<code>real</code>	Converts integer to real.
<code>floor</code>	<code>real</code>	<code>int</code>	Rounds down.
<code>ceil</code>	<code>real</code>	<code>int</code>	Rounds up.
<code>round</code>	<code>real</code>	<code>int</code>	Rounds to the nearest integer.
<code>trunc</code>	<code>real</code>	<code>int</code>	Truncates after the decimal point, effectively rounding toward zero.
<code>ord</code>	<code>char</code>	<code>int</code>	Finds the ASCII code for the given character.
<code>chr</code>	<code>int</code>	<code>char</code>	Finds the character with the given ASCII code.
<code>str</code>	<code>char</code>	<code>string</code>	Converts a character to a one-character string.

(Note that `real` is used in ML both as the name of a predefined function and as the name of a type.) The next example shows some of these conversion functions at work.

```
- real(123);
val it = 123.0 : real
- floor(3.6);
val it = 3 : int
- floor 3.6;
val it = 3 : int
- str #"a";
val it = "a" : string
```

Did you notice that we stopped using parentheses around the function parameter in the middle of that example? To call a function in ML, you just write the function's name followed by its parameter. You can write parentheses around either the name or the parameter or both, but the preferred ML style is to avoid them. This is an important and rather unusual thing about ML syntax. The expressions `f(1)`, `(f)1`, `(f)(1)`, `(f 1)`, and `f 1` all have the same value—the value returned by function `f` when it is called with the parameter `1`.

You might have to use parentheses in a function application if ML's precedence and associativity for function application are not what you want. Function application has very high precedence, higher than anything else we have seen. For example, the expression `f a+1` is evaluated by applying `f` to `a`, then adding `1` to

the result. If you want to apply f to the value $a+1$, you have to indicate that using parentheses: $f(a+1)$. Also, function application is left-associative (for reasons that will become clear in Chapter 9). So if you want to compute $f(g(1))$, you have to write it that way, or at least write $f(g\ 1)$. The expression $f\ g\ 1$ won't do the same thing.

5.7 Variable Definition

The `val` keyword is used to define a new variable and bind it to a value.

```
- val x = 1 + 2 * 3;
val x = 7 : int
- x;
val it = 7 : int
- val y = if x = 7 then 1.0 else 2.0;
val y = 1.0 : real
```

Variable names defined with `val` should consist of a letter, followed by zero or more additional letters, digits, and/or underscores. As observed before, ML is case sensitive, so the variable `x` and the variable `X` are two different things.

You *can* use `val` to redefine an existing variable, giving it a new value and even a new type:

```
- val fred = 23;
val fred = 23 : int
- fred;
val it = 23 : int
- val fred = true;
val fred = true : bool
- fred;
val it = true : bool
```

It is not particularly useful to redefine variables like this, but it is mentioned here because `val` definitions do look a little like the assignment statements used in imperative languages. Do not be deceived. When you give a new definition of a variable, it does not assign a new value to the variable. It does not alter or overwrite the previous definition. It only adds a new definition on top of the previous one. When we get around to writing larger programs, this distinction becomes very important. Any part of the program that was using the old definition before you redefined it is still using the old definition afterwards. A new definition with `val` does not have side effects on other parts of the program.

We can now give the full answer to a question we skirted previously: when you type an expression into ML, why does it respond with a line beginning with

`val it = ?` The reason is that the ML language system actually expects keyboard input to be a series of definitions, such as `val` definitions. If you just type an expression *exp*, rather than a definition, as we have in most of the examples of this chapter, ML treats it as if you had typed `val it = exp`. This makes a new instance of a variable named `it` and binds it to the value of your expression. Thus, the variable `it` always has the value of the last expression typed.

5.8 Garbage Collection

Sometimes, for no apparent reason, the SML/NJ language system prints a line that looks like this:

```
GC #0.0.0.0.1.3: (0 ms)
```

If you have been trying the examples in this chapter, you may have seen a line like this in the middle of ML's normal output. This message is what SML/NJ says when it is performing *garbage collection*, reclaiming pieces of memory that are no longer being used. Chapter 14 discusses garbage collection further. Depending on your installation, you may or may not see these messages. If you do see them, you can just ignore them.

5.9 Tuples and Lists

Most languages allow functions to be called with a list of parameters, such as `f(1, 2, 3)`. Putting `(1, 2, 3)` together in parentheses groups the parameters together into an ordered parameter list that is passed to the function. An ordered collection of values of different types is sometimes called a *tuple*. ML supports tuples in a more general way than most languages. It allows tuples as expressions anywhere, not just for parameter lists.

```
- val barney = (1 + 2, 3.0 * 4.0, "brown");
val barney = (3,12.0,"brown") : int * real * string
- val point1 = ("red", (300, 200));
val point1 = ("red", (300,200)) : string * (int * int)
```

A tuple in ML is formed just by putting two or more expressions, separated by commas, inside parentheses. As the second expression above shows, tuples can even contain other tuples.

In the example above, the type of `barney` is reported by ML to be `int * real * string`. Obviously, the symbol `*` is not being used as a multiplication operator in this case. Instead, it is being used as a *type constructor*. Given

any two ML types a and b , $a * b$ is the ML type for tuples of two things, the first of type a and the second of type b . Parentheses are significant in tuple types; `string * (int * int)` is not the same as `(string * int) * int`, and neither is the same as `string * int * int`.

To extract the i th element of a tuple v in ML, write the expression `# i v`. Tuple positions are numbered from left to right, starting with 1.

```
- #2 barney;
val it = 12.0 : real
- #1 (#2 point1);
val it = 300 : int
```

One final observation about tuples: they can have any length greater than one, but there is no such thing as a tuple of one. If you write a single expression inside parentheses, like `(1+2)`, the parentheses just serve to group the operations in the usual way. No tuple is constructed.

In addition to tuples, ML has lists. One important difference between a tuple and a list is that all the elements of a list must be of the same type. Lists are formed using square brackets instead of parentheses. They can contain any number of elements.

```
- [1, 2, 3];
val it = [1,2,3] : int list
- [1.0, 2.0];
val it = [1.0, 2.0] : real list
- [true];
val it = [true] : bool list
```

In the example above, the type of `[1, 2, 3]` is reported to be `int list`. Here, `list` is another type constructor. Given any ML type a , the type a list applies to lists of things of type a . Lists can contain any type of element, even tuples and other lists, as long as every element of the list has the same type.

```
- [(1, 2), (1, 3)];
val it = [(1,2), (1,3)] : (int * int) list
- [[1, 2, 3], [1, 2]];
val it = [[1,2,3], [1,2]] : int list list
```

Is the difference between a tuple and a list clear? Consider the next example, which shows a tuple of three integers and a list of three integers.

```
- val x = (1, 2, 3);
val x = (1,2,3) : int * int * int
- val y = [1, 2, 3];
val y = [1,2,3] : int list
```

The variable `x` is a tuple of three integers, and the variable `y` is a list of three inte-

gers. Note their different types as determined by ML. Although *x* and *y* look similar, the kinds of things you can do with them in ML are very different. In particular, a function that can take *x* as its parameter will apply only to tuples of three integers (values of type `int * int * int`). On the other hand, a function that can take *y* as its parameter will apply to all lists of integers—`int list` is the type of any list of integers, no matter what its length. The right choice depends on the problem being solved.

The empty list in ML can be written either as `nil` or just as `[]`. The empty list in ML has some slight peculiarities. Unlike with all other list constants, ML cannot tell the exact type of `[]`. Is it the empty list of integers? The empty list of strings?

```
- [];
val it = [] : 'a list
- nil;
val it = [] : 'a list
```

ML gives the type for the empty list as `'a list`. Names beginning with an apostrophe, like `'a` in this example, are *type variables*. A type variable stands for a type that is unknown. The type `'a list` might be translated into English as “a list of elements, type unknown.” A useful predefined function called `null` tests whether a list is empty:

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

It is also possible to test whether a list is empty by comparing it for equality with the empty list, as in the expression `x = []`. But the function `null` is preferred for this, for reasons that will be described in the next section.

The `@` operator in ML is used to concatenate two lists, which of course must have the same type:

```
- [1, 2, 3] @ [4, 5, 6];
val it = [1,2,3,4,5,6] : int list
```

The `@` operator does for lists what the `^` operator does for strings. Both the parameters of `@` must be lists. The expression `1@[2, 3]` is incorrect. You would either have to write `[1]@[2, 3]` or use a different operator, the *cons* operator, which is written as `::` (a double colon). Informally, you can think of the *cons* operator as gluing a new element onto the front of a list; for example, `1 :: [2, 3]` evaluates to the list `[1, 2, 3]`.

```
- val x = #"c" :: [];
```

```

val x = ["c"] : char list
- val y = ["b"] :: x;
val y = ["b","c"] : char list
- val z = ["a"] :: y;
val z = ["a","b","c"] : char list

```

At first glance, it may seem that the @ operator would be a lot more useful than the cons operator. In fact, the cons operator is used far more often. This is for two reasons. First, it can be used naturally in recursive functions that construct lists one element at a time. Second, it is more efficient (as will be shown in Chapter 21). There are quite a few languages that provide an operator like :: to construct lists. Such an operator was first introduced in the Lisp language, where it was called *cons* (an abbreviation for *construct*). That name has become generic, and now any operator like ML's :: is called a cons operator.³

Unlike most operators in ML, the cons operator is right-associative. This turns out to be the most natural associativity for this operation. For example, you would expect `1::2::3::[]` to evaluate to the list `[1,2,3]`, and it does. If :: were left-associative, `1::2::3::[]` would be an error, since the leftmost pair `1::2` does not even have a list as its second operand.

The two important functions for extracting parts of a list are `hd` and `tl` (which are abbreviations for *head* and *tail*).

```

- val z = 1 :: 2 :: 3 :: [];
val z = [1,2,3] : int list
- hd z;
val it = 1 : int
- tl z;
val it = [2,3] : int list
- tl(tl z);
val it = [3] : int list
- tl(tl(tl z));
val it = [] : int list

```

As you can see, the `hd` function returns the first element of the list, and the `tl` function returns the rest of the list after the first element. It is an error to try to compute the `hd` or `tl` of an empty list.

Although a string in ML is not the same as a list of characters, they obviously have a lot in common. The `explode` function converts a value of type `string` into a `char list`, and the `implode` function does the opposite conversion.

3. With continued exposure, many people find themselves using the word *cons* as a verb—to *cons* something onto a list means to attach it at the front and to *cons up* a list means to build the list by *consing* things onto it one at a time.

```

- explode "hello";
val it = [#"h",#"e",#"l",#"l",#"o"] : char list
- implode [#"h", #"i"];
val it = "hi" : string

```

5.10 Function Definitions

Up to this point the ML language system has looked like a calculator; you type things and it evaluates them. The next piece of ML is the first step toward making it less like a calculator and more like a programming language. To define new functions in ML you give a `fun` definition, like this:

```

- fun firstChar s = hd (explode s);
val firstChar = fn : string -> char
- firstChar "abc";
val it = #"a" : char

```

This function, `firstChar`, takes a string parameter and returns its first character. The syntax of the `fun` definition is quite simple:

```

<fun-def> ::=
    fun <function-name> <parameter> = <expression> ;

```

The `<function-name>` is the name of the function being defined. It can be any legal ML name. The simplest `<parameter>` is just a variable name, as in the `firstChar` example. The `<expression>` is any ML expression. The value of the `<expression>` is the value the function returns. (This is a subset of the legal syntax for function definitions in ML. Chapter 7 discusses this further.)

Notice that ML figured out the type of `firstChar` without having to be told, just as it has done all along for expressions. That type, `string -> char`, describes functions that take a `string` parameter and return a `char` result. ML knows that the parameter `s` must be a string because the `explode` function was applied to it, and it knows that the result must be a character because that is what `hd` returns when applied to a list of characters.

The `->` symbol is another type constructor, like `*` and `list`. Given any two types `a` and `b`, `a -> b` is the type for functions that take a parameter of type `a` (the domain type) and return a result of type `b` (the range type).

To write a function that takes more than one input value, you can use a tuple parameter. For example, here is a function that returns the integer quotient of two integer values:


```
- fun quot(a, b) = a div b;
val quot = fn : int * int -> int
- quot (6, 2);
val it = 3 : int
```

It looks like the same kind of parameter list found in many other languages, but remember that ML handles tuple values in a much more general way. Consider this example:

```
- val pair = (6, 2);
val pair = (6,2) : int * int
- quot pair;
val it = 3 : int
```

Here we have defined the variable `pair` to be the tuple `(6, 2)`, then called our function `quot` passing that tuple. This shows that there is nothing special about parameter lists on a function call. They are just tuples. Every ML function takes exactly one parameter—that parameter may be a tuple, but whether you build the tuple when you call the function, as in `quot (6, 2)`, or whether you pass a tuple you have already constructed, as in `quot pair`, makes no difference to ML.

You have already seen enough ML to get a lot of work done with functions. Here, for example, is a function to compute the factorial of a non-negative integer:

```
- fun fact n =
=   if n = 0 then 1
=   else n * fact(n - 1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int
```

Notice that the definition is spread out over more than one line. Like most modern languages, ML does not care where line breaks occur. They are there only to make the function definition more readable. We could have put the whole thing on one line.

The previous example was a recursive function definition. The `fact` function has a base case (`if n = 0 then 1`) that says what value to return for the smallest legal input. It has a recursive case (`else n * fact(n - 1)`) in which the function calls itself, but with a value that is closer to the base case. Recursion is used much more heavily in ML and the other functional languages than in most imperative languages. Imperative languages make heavy use of iteration: while loops, for loops, and the like. Functional languages make heavy use of recursion. It is possible to write iterative functions in ML, but it is rarely done. We will not use ML's iterative constructs at all in this book.

This next function adds up all the elements of a list:


```

- fun listsum x =
=   if null x then 0
=   else hd x + listsum(tl x);
val listsum = fn : int list -> int
- listsum [1, 2, 3, 4, 5];
val it = 15 : int

```

The `listsum` function definition illustrates a common pattern for recursive functions in ML. The base case applies when the list is `nil`, and the recursive call passes the `tl` of the list. In this way, `listsum` is called with `x`, then recursively with `tl x`, then with `tl (tl x)`, and so on all the way down to `nil`. That gives `listsum` a chance to look at each element of the list (using `hd x`). This is a pattern to consider whenever you are writing a function that has to do something for each element of a list.

A useful predefined function in ML is the `length` function, which computes the length of a list. This next example shows an implementation of it:

```

- fun length x =
=   if null x then 0
=   else 1 + length (tl x);
val length = fn : 'a list -> int
- length [true, false, true];
val it = 3 : int
- length [4.0, 3.0, 2.0, 1.0];
val it = 4 : int

```

An interesting thing about this function definition is the type ML decided for it: `'a list -> int`. As has already been shown, `'a` is a type variable. The input to `length` is a list of elements of unknown type. This is an example of a *polymorphic* function—it allows parameters of different types. We will not have to write a specialized length-computing function for every type of list, one for lists of booleans, another for lists of reals, and so on. This one `length` function will work on all types of lists. ML functions often end up being polymorphic. There is no special trick to making them that way. This example did not need to use any special syntax. ML just found the type in the usual way.

Now we can answer a question brought up in the previous section: why you should use the test `null x` instead of `x = []`. Look at what happens if `length` is defined using the test `x = []`:

```

- fun badlength x =
=   if x = [] then 0
=   else 1 + badlength (tl x);
val badlength = fn : 'a list -> int
- badlength [true, false, true];

```

```
val it = 3 : int
- badlength [4.0, 3.0, 2.0, 1.0];
Error: operator and operand don't agree
[equality type required]
```

ML gives `badlength` the type `'a list -> int`. There is a minor difference between this and the type of `length`—a critical extra apostrophe. Type variables beginning with a double apostrophe, like `'a`, are restricted to equality-testable types. The function `badlength` works on most types of lists, but not on lists of reals, since reals cannot be tested for equality. The source of the problem is the test `x = []`. Because of this test, ML adopts the restriction that `x`'s elements must be equality testable. That is why you should use `null x` instead of `x = []`; it avoids this unnecessary type restriction.

Let's see one more example of a recursive function in ML. This one reverses a list.

```
- fun reverse L =
  = if null L then nil
  = else reverse(tl L) @ [hd L];
val reverse = fn : 'a list -> 'a list
- reverse [1, 2, 3];
val it = [3,2,1] : int list
```

In English, this function definition might be said in this way: "The reverse of an empty list is an empty list, and the reverse of any other list is the list you get by appending the first element onto the end of the reverse of the rest of the list." It's no easier to understand in English, is it?

5.11 ML Types and Type Annotations

So far we have seen the ML types `int`, `real`, `bool`, `char`, and `string`. We have also seen three type constructors: `*` for making tuple types, `list` for making list types, and `->` for making function types.

When the three type constructors are combined in a more complicated type, `list` has highest precedence and `->` has lowest precedence. For example, the type `int * int list` is the same as `int * (int list)`—the type of pairs of which the first item is an integer and the second a list of integers. The type for a list of pairs of integers would have to be written as `(int * int) list`, using parentheses to overcome the higher precedence of the `list` type constructor.

ML has discovered and written out all the types, so it might seem like a waste of time to learn how to write them yourself. Actually, it is important to know how

to do it, because you do occasionally have to write ML types in an ML program. Sometimes ML's type inference needs a little help and *type annotations* are necessary. Consider this function:

```
- fun prod(a, b) = a * b;
  val prod = fn : int * int -> int
```

How did ML decide on the type `int * int -> int` for this function? Why not `real * real -> real`? Wouldn't a function to multiply two real numbers be written exactly the same way?

ML has no information about the types of `a` and `b` in `prod` other than the `*` operator that is applied to them. The `*` operator could apply to integers or to real numbers. When there are no other clues, ML uses the *default type* for `*`, which is `int * int -> int`. (The same thing would apply to the operators `+` and `-`.) If you want to define `prod` so that it applies to real numbers, you have to give ML a more definite clue: a type annotation. Here is one way to do it:

```
- fun prod(a:real, b:real) : real = a * b;
  val prod = fn : real * real -> real
```

A type annotation is just a colon followed by an ML type. The example above has three type annotations that establish the types of `a`, `b`, and the returned value.

Unlike most languages, ML allows type annotations *after any variable or expression*. For instance, we could have given ML any one of these alternate clues:

```
fun prod(a, b) : real = a * b;
fun prod(a : real, b) = a * b;
fun prod(a, b : real) = a * b;
fun prod(a, b) = (a : real) * b;
fun prod(a, b) = a * b : real;
fun prod(a, b) = (a * b) : real;
fun prod((a, b) : real * real) = a * b;
```

These all work and accomplish the same thing. One hint, anywhere, is enough to help ML decide on the type. But, although ML treats these all the same, the original example is probably the best since it is the most readable. In fact, enhancing readability is probably the major reason for using type annotations in ML. ML can usually figure out types without help, but the human reader will appreciate all the help he or she can get! This book uses type annotations sparingly. That suffices only because the examples are small and are described in the text. A maturer ML programming style for larger ML projects would use type annotations more heavily. Many ML programmers give type annotations with every `fun` definition, as in the example above. Some styles of ML programming go even further, giving type annotations with variable definitions throughout the code.

5.12 Conclusion

This chapter discussed the language ML. The following parts of the language were introduced:

- The ML types `int`, `real`, `bool`, `char`, and `string` and how to write constants of each type.
- The ML operators `~`, `+`, `-`, `*`, `div`, `mod`, `/`, `^`, `::`, `@`, `<`, `>`, `<=`, `>=`, `=`, `<>`, `not`, `andalso`, and `orelse`.
- The conditional expression.
- Function application.
- The predefined functions `real`, `floor`, `ceil`, `round`, `trunc`, `chr`, `ord`, `str`, `hd`, `tl`, `explode`, `implode`, and `null`.
- Defining new variable bindings using `val`.
- Tuple construction using `(x, y, ..., z)` and selection using `#n`.
- List construction using `[x, y, ..., z]`.
- The type constructors `*`, `list`, and `->`.
- Function definition using `fun`, including tuples as parameters, polymorphic functions, and recursive functions.
- Type annotations.

This is enough ML to complete the exercises that follow.

Exercises

Throughout this chapter, we have used the SML/NJ language system in an interactive mode. For longer examples, it makes more sense to store your function definitions in a file. Once you have created a file containing a definition or definitions, you can load it into an ML session by using the predefined `use` function. For example, if you have created a file named `assign1.sml` in the current directory, you can run your ML language system and type `use "assign1.sml"`; after the prompt. The ML language system will read the contents of the file just as if you had typed it one line at a time. After `use` finishes, you can continue typing interactive ML expressions, for example, to test the functions defined in your file.

Exercise 1 Write a function `cube` of type `int -> int` that returns the cube of its parameter.

Exercise 2 Write a function `cuber` of type `real -> real` that returns the cube of its parameter.

Exercise 3 Write a function `fourth` of type `'a list -> 'a` that returns the fourth element of a list. Your function need not behave well on lists with less than four elements.

Exercise 4 Write a function `min3` of type `int * int * int -> int` that returns the smallest of three integers.

Exercise 5 Write a function `red3` of type `'a * 'b * 'c -> 'a * 'c` that converts a tuple with three elements into one with two by eliminating the second element.

Exercise 6 Write a function `thirds` of type `string -> char` that returns the third character of a string. Your function need not behave well on strings with lengths less than 3.

Exercise 7 Write a function `cycle1` of type `'a list -> 'a list` whose output list is the same as the input list, but with the first element of the list moved to the end. For example, `cycle1 [1,2,3,4]` should return `[2,3,4,1]`.

Exercise 8 Write a function `sort3` of type `real * real * real -> real list` that returns a list of three real numbers, in sorted order with the smallest first.

Exercise 9 Write a function `del3` of type `'a list -> 'a list` whose output list is the same as the input list, but with the third element deleted. Your function need not behave well on lists with lengths less than 3.

Exercise 10 Write a function `sqsum` of type `int -> int` that takes a non-negative integer n and returns the sum of the squares of all the integers 0 through n . Your function need not behave well on inputs less than zero.

Exercise 11 Write a function `cycle` of type `'a list * int -> 'a list` that takes a list and an integer n as input and returns the same list, but with the first element cycled to the end of the list n times. (Make use of your `cycle1` function from a previous exercise.) For example, `cycle ([1,2,3,4,5,6], 2)` should return the list `[3,4,5,6,1,2]`.

Exercise 12 Write a function `pow` of type `real * int -> real` that raises a real number to an integer power. Your function need not behave well if the integer power is negative.

Exercise 13 Write a function `max` of type `int list -> int` that returns the largest element of a list of integers. Your function need not behave well if the list is empty. *Hint:* Write a helper function `maxhelper` that takes as a second parameter the largest element seen so far. Then you can complete the exercise by defining

```
fun max x = maxhelper (tl x, hd x);
```

Exercise 14 Write a function `isPrime` of type `int -> bool` that returns true if and only if its integer parameter is a prime number. Your function need not behave well if the parameter is negative.

Exercise 15 Write a function `select` of this type:

```
'a list * ('a -> bool) -> 'a list
```

that takes a list and a function f as parameters. Your function should apply f to each element of the list and should return a new list containing only those elements of the original list for which f returned true. (The elements of the new list may be given in any order.) For example, evaluating `select ([1,2,3,4,5,6,7,8,9,10], isPrime)` should result in a list like `[7,5,3,2]`. This is an example of a *higher-order* function, since it takes another function as a parameter. We will see much more about higher-order functions in Chapter 9.

Further Reading

This is a great book to help you learn more about ML:

Ullman, Jeffrey D. *Elements of ML Programming*. Upper Saddle River, NJ: Prentice Hall, 1998.

It covers the basics seen in this chapter, the more advanced things that will be seen in later chapters, and the even more advanced things there will not be time to discuss.