Chapter 9
A Third Look at ML

9.1 Introduction

This chapter continues the introduction to ML. It will start with some more advanced pattern-matching features. In Chapter 7 you learned about the pattern-matching style of function definitions:

```
fun f 0 = "zero"
  | f _ = "non-zero";
```

The same syntax for pattern matching can be used in several other places in ML programs, such as the `case` expression:

```
case n of
  0 => "zero" |
  _ => "non-zero"
```

After covering that, this chapter will turn to higher-order functions. A higher-order function is one that takes other functions as parameters or produces them as returned values. Such functions are used much more often in functional languages than in traditional imperative languages. By the end of this chapter you will see why.

9.2 More Pattern Matching

A *rule* is a piece of ML syntax that looks like this:

```
<rule> ::= <pattern> => <expression>
```

A *match* consists of one or more rules separated by the | token, like this:

```
<match> ::= <rule> | <rule> '|' <match>
```

As always, ML doesn't care how you break a match across lines, but it is easier for people to read if each rule in the match is on a line by itself. The pattern of a rule can, as always, define variables that are bound by pattern matching. The scope of those definitions runs from the point of definition to the end of the rule. A rule is a kind of block in ML. Each rule in a match must have the same type of expression on the right-hand side. A match is not an ML expression by itself, but it forms part of several different kinds of ML expressions. You have already seen something much like this for pattern-matching function definitions.

One important kind of ML expression that uses a match is the case expression. Its syntax is simple:

```
<case-exp> ::= case <expression> of <match>
```

For example:

```
- case 1 + 1 of
=    3 => "three" |
=    2 => "two" |
=    _ => "hmm";
val it = "two" : string
```

The value of `case` *<expression>* `of` *<match>* is the value of the expression in the first rule of the *<match>* whose pattern matches the value of the *<expression>*. In the example above, the case expression has the value 2, which matches the pattern `2`, so the value of the entire case expression is the string `"two"`.

That example does not use the full pattern-matching power of ML's case expression. Many languages have some kind of case construct that matches the value of an expression against compile-time constants, with a default entry in case there is no match. But few languages have a case expression that allows general pattern matching. For example, the following expression produces the third element of `x` (an `int list`) if the list has three or more elements, or the second element if the list has only two elements, or the first element if the list has only one element, or the constant 0 if the list is empty:

```
case x of
  _ :: _ :: c :: _ => c |
  _ :: b :: _ => b |
  a :: _ => a |
  nil => 0
```

The case expression can easily do everything the conditional expression can do. Any expression of the form `if` exp_1 `then` exp_2 `else` exp_3 can be rewritten as a two-rule case expression like this:

```
case exp₁ of
  true => exp₂ |
  false => exp₃
```

From the perspective of the ML system, it makes no difference whether you write a conditional expression or the equivalent case expression. Of course, from the perspective of the human reader, the conditional expression is easier to read and understand, so you should use the case expression only when you need its extra flexibility.

9.3 Function Values and Anonymous Functions

When the ML language system starts up, many variables are already defined and bound. This includes all the predefined functions like `ord` and operators like `~`:

```
- ord;
val it = fn : char -> int
- ~;
val it = fn : int -> int
```

Function names, like `ord`, and operator names, like `~`, are variables just like any others in ML. They are variables that happen to be bound, initially, to functions. That is what ML is reporting in the example above; `ord` is a variable whose value is a function of type `char -> int`, and `~` is a variable whose value is a function of type `int -> int`.

The definition `val x = 7` binds `x` to the integer 7. In the same way, this next example binds `x` to the function denoted by the `~` operator.

```
- val x = ~;
val x = fn : int -> int
- x 3;
val it = ~3 : int
```

As you can see, x now denotes the same function that ~ does. It was applied to the operand 3 to negate it.

This can be a hard concept to follow at first, especially if you are accustomed to a conventional imperative language. In most imperative languages, a function definition gives a unique, permanent name to the function. In ML, a function is not tied to a particular function name. We may speak informally of "the function f" in some ML program, but it would be more accurate to speak of "the function currently bound to the name f." Functions themselves do not *have* names, though one or more names may be bound to a particular function.

Then how can a function be created without a name? The fun syntax creates a new function and binds a name to the function automatically. But there is another way to define a function, a way that does not involve giving it a name. To define such an anonymous function, simply give the keyword fn followed by a match. Here is a named function f, defined using the original fun syntax, that adds 2 to its integer parameter:

```
- fun f x = x + 2;
val f = fn : int -> int
- f 1;
val it = 3 : int
```

Here is the same computation using an anonymous function defined with an fn expression:

```
- fn x => x + 2;
val it = fn : int -> int
- (fn x => x + 2) 1;
val it = 3 : int
```

The anonymous function fn x => x + 2 is exactly the same as the named function f and can be applied to parameters in exactly the same way. In fact, what fun does can now be defined in terms of val and fn, since fun f x = x + 2 has the same effect as val f = fn x => x + 2.[1]

Anonymous functions are useful when you need a small function in just one place and don't want to clutter up the program by giving that function a name. For example, suppose a sorting function called quicksort takes two inputs—the list to be sorted and a comparison function that decides whether one element should come before another. (You implemented such a function if you did Exercise 7 in

1. There is actually a slight but important difference between a definition with fun and one with val and fn. The scope of the definition of f produced by fun f includes the function body being defined, while the scope of the definition of f produced by val f = fn does not. So only the fun version can be recursive.

Chapter 7.) The comparison function is trivial. It seems ugly to clutter up the program with a named function like this:

```
- fun intBefore (a,b) = a < b;
val intBefore = fn : int * int -> bool
- quicksort ([1,4,3,2,5], intBefore);
val it = [1,2,3,4,5] : int list
```

It would be simpler just to make an anonymous comparison function at the point where it is needed, as in these two examples:

```
- quicksort ([1,4,3,2,5], fn (a,b) => a < b);
val it = [1,2,3,4,5] : int list
- quicksort ([1,4,3,2,5], fn (a,b) => a > b);
val it = [5,4,3,2,1] : int list
```

There is an even shorter way to write the two previous examples. The anonymous comparison functions in those examples do exactly what the < and > operators do. Why not just use the < and > operators directly as the comparison functions? Unfortunately, we cannot just write `quicksort(x, <)`, since ML expects < to be used as a binary operator. But there is a way to extract the function denoted by an operator: the `op` keyword. The value of the expression `op <` is the function used by the operator <. Using this trick, we can simply write:

```
- quicksort ([1,4,3,2,5], op <);
val it = [1,2,3,4,5] : int list
- quicksort ([1,4,3,2,5], op >);
val it = [5,4,3,2,1] : int list
```

Most functional languages have an expression like ML's `fn` expression, whose value is an anonymous function.[2]

9.4 Higher-Order Functions and Currying

Every function has an *order*, defined as follows:

> A function that does not take any functions as parameters and does not return a function value has *order 1*.

2. The idea of an expression whose value is an anonymous function goes back to the first versions of Lisp, which used the name `lambda` for something that works like `fn` in ML. In a theoretical form, the idea goes back to the branch of mathematics called the "lambda calculus," which manipulates anonymous functions with a notation that uses the Greek letter lambda (λ) to introduce a function. The name has become generic, so that any expression like ML's `fn` expression is called a "lambda expression."

> A function that takes a function as a parameter or returns a function value has *order n+1*, where *n* is the order of its highest-order parameter or returned value.

A function of order *n* is called an *nth order function*, and a function of any order greater than 1 is called a *higher-order function*. Higher-order functions are used much more often in functional languages like ML than in imperative languages.

As you know, functions in ML take exactly one parameter. You have already seen one way to squeeze multiple parameters into a function, which is to pass a tuple as the parameter. There is another way to do it using higher-order functions. You can write a function that takes the first parameter and returns another function. The new function takes the second parameter and returns the ultimate result. This trick is called *currying*.[3] The next example shows a function `f` with a tuple parameter, and a curried function `g` that does the same thing:

```
- fun f (a,b) = a + b;
val f = fn : int * int -> int
- fun g a = fn b => a + b;
val g = fn : int -> int -> int
- f (2,3);
val it = 5 : int
- g 2 3;
val it = 5 : int
```

The function `g` takes the first parameter and returns an anonymous function that takes the second parameter and returns the sum. Notice that `f` and `g` are called differently. `f` expects a tuple, but `g` expects just the first integer parameter. Because function application in ML is left associative, the expression `g 2 3` means `(g 2) 3`; that is, first apply `g` to the parameter `2` and then apply the resulting function (the anonymous function `g` returns) to the parameter `3`.

We do not have to create a tuple when calling `g`, but that is not the main advantage of currying. The real advantage is that we can call curried functions, passing only some of their parameters and leaving the rest for later. For example, we can call `g` with its first parameter only:

```
- val add2 = g 2;
val add2 = fn : int -> int
- add2 3;
val it = 5 : int
- add2 10;
val it = 12 : int
```

3. "Currying" is named not for the spicy food, but for the mathematician Haskell Brooks Curry (1900–1982), who made significant contributions to the mathematical theory of functional programming. There is also a major functional programming language named for him: Haskell.

Here g was used to create a function called add2 that knows how to add 2 to any parameter. In effect, add2 is a specialized version of g, with the first parameter fixed at 2 but the second parameter still open.

For a more practical example, imagine defining the quicksort function as a curried function that takes the comparison function first and the list to sort second. So it would have this type:

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

We could use this curried quicksort in the usual way, giving all of its parameters at once:

```
- quicksort (op <) [1,4,3,2,5];
val it = [1,2,3,4,5] : int list
```

Or we could use it to create specialized sorting functions by giving just the first parameter, like this:

```
- val sortBackward = quicksort (op >);
val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
val it = [5,4,3,2,1] : int list
```

As you can see, the curried quicksort is useful in more ways than a quicksort that takes its parameters as a tuple.

Of course, currying could be generalized for any number of parameters. Here is an example that adds three numbers together, first using a tuple and then in curried form:

```
- fun f (a,b,c) = a + b + c;
val f = fn : int * int * int -> int
- fun g a = fn b => fn c => a + b + c;
val g = fn : int -> int -> int -> int
- f (1,2,3);
val it = 6 : int
- g 1 2 3;
val it = 6 : int
```

Once you are sure that you understand how these currying examples work, you can be let in on a little secret: there is a much easier way to write curried functions. For example, these two definitions do exactly the same thing:

```
fun g a = fn b => fn c => a + b + c;
fun g a b c = a + b + c;
```

The second way is much shorter to write but identical in meaning. ML treats it as an abbreviation for the first way. This section started out showing you how to write curried functions the long way because the long way makes all the intermediate anonymous functions explicit. But once you really understand how this works, you should use the short way, since it is much easier to write and read.

9.5 Predefined Higher-Order Functions

This section will use three important predefined higher-order functions: `map`, `foldr`, and `foldl`. (As you might guess from the names, `foldr` and `foldl` are very similar, with one subtle but important difference.) Once you get comfortable using these functions you will find them very helpful. The exercises at the end of this chapter will give you some idea of the versatility of these functions.

The `map` Function

The `map` function is used to apply some function to every element of a list, collecting a list of the results. For example:

```
- map ~ [1,2,3,4];
val it = [~1,~2,~3,~4] : int list
```

This example applies the negation function, `~`, to every element of the list `[1,2,3,4]`. The result is the original list, with every element negated. Here are some other examples:

```
- map (fn x => x + 1) [1,2,3,4];
val it = [2,3,4,5] : int list
- map (fn x => x mod 2 = 0) [1,2,3,4];
val it = [false,true,false,true] : bool list
- map (op +) [(1,2),(3,4),(5,6)];
val it = [3,7,11] : int list
```

The last example applies the function of the + operator to a list of pairs. The result is a list of the sums formed from each pair.

You can tell from the way `map` is called that it is a curried function. It takes two parameters, but they are not grouped together as a tuple. Its type is

```
('a -> 'b) -> 'a list -> 'b list
```

If you call it with just the first parameter, you get a function that transforms lists using a fixed function. For example:

```
- val f = map (op +);
val f = fn : (int * int) list -> int list
- f [(1,2),(3,4)];
val it = [3,7] : int list
```

The most important thing to remember about using `map` is that the result will always be a list that is the same length as the input list. When you have a problem to solve that involves converting one list into another list of the same length, and when each element of the output list depends only on the corresponding element of the input list, you may be able to use `map` to solve the problem. If these requirements are not met, you may be able to use `foldr` or `foldl` instead.

The `foldr` Function

The `foldr` function is used to combine all the elements of a list into one value. This example adds up all the elements of a list:

```
foldr (op +) 0 [1,2,3,4];
```

The `foldr` function takes three parameters: a function *f*, a starting value *c*, and a list of values $[x_1, \ldots, x_n]$. It starts with the rightmost element x_n and computes $f(x_n,c)$. Then it folds in the next element x_{n-1}, computing $f(x_{n-1},f(x_n,c))$. It continues in this way until all the elements have been combined. The result is

$$f(x_1,f(x_2,\ldots f(x_{n-1},f(x_n,c))\ldots))$$

For example, `foldr (op +) 0 [1,2,3,4]` adds up the list by computing 1+(2+(3+(4+0))). Here is `foldr` in action:

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
- foldr (op * ) 1 [1,2,3,4];
val it = 24 : int
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldr (op ::) [5] [1,2,3,4];
val it = [1,2,3,4,5] : int list
```

Did you notice the space after the `*` in the expression `(op * )`? Writing the expression without that extra space would confuse ML, since `*)` is used to close ML comments.

You can tell from the way `foldr` is called that it is a curried function. It takes three parameters, but not grouped together as a tuple. This is its type:

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

One common way to use `foldr` is to give it the first two parameters, but not the third. The result is a function that takes a list and folds it with a fixed function and an initial value:

```
- val addup = foldr (op +) 0;
val addup = fn : int list -> int
- addup [1,2,3,4,5];
val it = 15 : int
```

The examples above show `foldr` being used with operators: `op +`, `op ::`, and so forth. In the exercises at the end of the chapter, you will rarely find that there is an operator that performs exactly the function you need. Usually, you have to write a little anonymous function for `foldr`, so the call will look more like this:

```
foldr (fn (a,b) => function body) c x
```

Here, *c* is the starting value for the fold—usually a constant—and *x* is the input list. Some important tips for using `foldr`:

- On the first call of the anonymous function, `a` will be the rightmost element from the list *x* and `b` will be the starting value *c*.
- On each subsequent call of the anonymous function, `a` will be the next element from the list *x* and `b` will be the result accumulated so far—that is, the previous value returned by the anonymous function.
- These values all have the same type: `b`, *c*, the value returned by the anonymous function, and the value returned by the `foldr`.
- The type of the elements of the list *x* is the same as the type of `a`.
- The starting value *c* is what `foldr` will return if the list *x* is empty.

These tips can help you figure out how to use `foldr` to solve programming problems. For example, suppose you want to write a function `thin` that takes an `int list` and returns the same list, but with all the negative numbers eliminated. You can rule out using `map` for this, since the result list does not necessarily have the same length as the input list. So you start with the pattern described above:

```
fun thin L = foldr (fn (a,b) => function body) c L;
```

The result is supposed to be an `int list`, so it follows that `b`, *c*, and the value returned by the anonymous function must also be of type `int list`. Furthermore, if the input list `L` is the empty list, the output should also be the empty list; so it follows that the starting value *c* should be `[]`. All that remains is to write the body of the anonymous function. It takes an `int a` and an `int list b`, and returns either `a::b` (if `a` is to be accumulated in the result) or just `b` (if `a` is to be omitted). This is simply

```
fun thin L =
  foldr (fn (a,b) => if a < 0 then b else a::b) [] L;
```

The `foldl` Function

Like `foldr`, the `foldl` function is used to combine all the elements of a list into one value. In fact, `foldr` and `foldl` sometimes produce the same results:

```
- foldl (op +) 0 [1,2,3,4];
val it = 10 : int
- foldl (op * ) 1 [1,2,3,4];
val it = 24 : int
```

All the tips about how to use `foldr` also apply to `foldl`, with one subtle difference: `foldl` starts with the leftmost element in the list and proceeds from left to right.

The `foldl` function takes the same three parameters as `foldr`: a function f, a starting value c, and a list of values $[x_1, \ldots, x_n]$. It starts with the *leftmost* element x_1 and computes $f(x_1,c)$. Then it folds in the next element x_2, computing $f(x_2,f(x_1,c))$. It continues in this way until all the elements have been combined. The result is

$$f(x_n,f(x_{n-1},\ldots f(x_2,f(x_1,c))\ldots))$$

For example, `foldl (op +) 0 [1,2,3,4]` adds up the list by computing 4+(3+(2+(1+0))). Compare this with the function computed by `foldr`. The `foldr` function starts with the rightmost element of the list; the `foldl` function starts with the leftmost element of the list. For operations that are associative and commutative, like addition and multiplication, this difference is invisible. For other operations, like concatenation, it matters a lot. Compare these `foldr` and `foldl` results:

```
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldl (op ^) "" ["abc","def","ghi"];
val it = "ghidefabc" : string
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```

9.6 Conclusion

This chapter introduced the following parts of ML:

- The general syntax for a match.
- The `case` expression (using matches).
- The idea of function values and anonymous functions.

- The `fn` expression for creating anonymous functions (using matches).
- The idea of higher-order functions.
- The idea of currying.
- The long and short forms for writing curried functions.
- The predefined, curried, higher-order functions `map`, `foldr`, and `foldl`.

Exercises

The first 25 exercises should all have one-line solutions using `map`, `foldr`, or `foldl`. You can also use other predefined functions, of course, but *do not write any additional named functions and do not use explicit recursion*. If you need helper functions, use anonymous ones. For example, if the problem says "write a function `add2` that takes an `int list` and returns the same list with 2 added to every element," your answer should be

```
fun add2 x = map (fn a => a + 2) x;
```

You have seen some of these problems before. The trick now is to solve them in this new, concise form.

Exercise 1 Write a function `il2rl` of type `int list -> real list` that takes a list of integers and returns a list of the same numbers converted to type `real`. For example, if you evaluate `il2rl [1,2,3]` you should get `[1.0,2.0,3.0]`.

Exercise 2 Write a function `ordlist` of type `char list -> int list` that takes a list of characters and returns the list of the integer codes of those characters. For example, if you evaluate `ordlist [#"A",#"b",#"C"]` you should get `[65,98,67]`.

Exercise 3 Write a function `squarelist` of type `int list -> int list` that takes a list of integers and returns the list of the squares of those integers. For example, if you evaluate `squarelist [1,2,3,4]` you should get `[1,4,9,16]`.

Exercise 4 Write a function `multpairs` of type `(int * int) list -> int list` that takes a list of pairs of integers and returns a list of the products of each pair. For example, if the input is `[(1,2),(3,4)]`, your function should return `[2,12]`.

Exercise 5 Write a function `inclist` of type `int list -> int -> int list` that takes a list of integers and an integer increment, and returns the same list of integers but with the integer increment added to each one. For example, if you evaluate `inclist [1,2,3,4] 10` you should get `[11,12,13,14]`. Note that the function is curried.

Exercise 6 Write a function `sqsum` of type `int list -> int` that takes a list of integers and returns the sum of the squares of those integers. For example, if you evaluate `sqsum [1,2,3,4]` you should get `30`.

Exercise 7 Write a function `bor` of type `bool list -> bool` that takes a list of boolean values and returns the logical OR of all of them. If the list is empty, your function should return `false`.

Exercise 8 Write a function `band` of type `bool list -> bool` that takes a list of boolean values and returns the logical AND of all of them. If the list is empty, your function should return `true`.

Exercise 9 Write a function `bxor` of type `bool list -> bool` that takes a list of boolean values and returns the logical exclusive OR of all of them. (It should return `true` if the number of `true` values in the list is odd and `false` if the number of `true` values is even.) If the list is empty, your function should return `false`.

Exercise 10 Write a function `dupList` of type `'a list -> 'a list` whose output list is the same as the input list, but with each element of the input list repeated twice in a row. For example, if the input list is `[1,3,2]`, the output list should be `[1,1,3,3,2,2]`. If the input list is `[]`, the output list should be `[]`.

Exercise 11 Write a function `mylength` of type `'a list -> int` that returns the length of a list. (Of course, you may not use the predefined `length` function to do it.)

Exercise 12 Write a function `il2absrl` of type `int list -> real list` that takes a list of integers and returns a list containing the absolute values of those integers, converted to real numbers.

Exercise 13 Write a function `truecount` of type `bool list -> int` that takes a list of boolean values and returns the number of trues in the list.

Exercise 14 Write a function maxpairs of type (int * int) list -> int list that takes a list of pairs of integers and returns the list of the max elements from each pair. For example, if you evaluate maxpairs [(1,3),(4,2),(~3,~4)] you should get [3,4,~3].

Exercise 15 Write a function myimplode that works just like the predefined implode. In other words, it should be a function of type char list -> string that takes a list of characters and returns the string containing those same characters in that same order.

Exercise 16 Write a function lconcat of type 'a list list -> 'a list that takes a list of lists as input and returns the list formed by appending the input lists together in order. For example, if the input is [[1,2],[3,4,5,6],[7]], your function should return [1,2,3,4,5,6,7]. (There is a predefined function like this called concat, which of course you should not use.)

Exercise 17 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.

Exercise 18 Write a function min of type int list -> int that returns the smallest element of a list of integers. Your function need not behave well if the list is empty.

Exercise 19 Write a function member of type ''a * ''a list -> bool so that member(e,L) is true if and only if e is an element of list L.

Exercise 20 Write a function append of type 'a list -> 'a list -> 'a list that takes two lists and returns the result of appending the second one onto the end of the first. For example, append [1,2,3] [4,5,6] should evaluate to [1,2,3,4,5,6]. Do not use predefined appending utilities, like the @ operator or the concat function. Note that the function is curried.

Exercise 21 Define a function less of type int * int list -> int list so that less(e,L) is a list of all the integers in L that are less than e (in any order).

Exercise 22 Write a function evens of type int list -> int list that takes a list of integers and returns the list of all the even elements from the original list (in the original order). For example, if you evaluate `evens [1,2,3,4]` you should get `[2,4]`.

Exercise 23 Write a function `convert` of type `('a * 'b) list -> 'a list * 'b list`, that converts a list of pairs into a pair of lists, preserving the order of the elements. For example, `convert [(1,2),(3,4),(5,6)]` should evaluate to `([1,3,5],[2,4,6])`.

Exercise 24 Define a function `mymap` with the same type and behavior as `map`, but without using `map`. (Note this should still be a one-liner: use `foldl` or `foldr`.)

Exercise 25 Represent a polynomial using a list of its (real) coefficients, starting with the constant coefficient and going only as high as necessary. For example, $3x^2 + 5x + 1$ would be represented as the list `[1.0,5.0,3.0]` and $x^3 - 2x$ as `[0.0,~2.0,0.0,1.0]`. Write a function `eval` of type `real list -> real -> real` that takes a polynomial represented this way and a value for `x` and returns the value of that polynomial at the given x. For example, `eval [1.0,5.0,3.0] 2.0` should evaluate to `23.0`, because when $x = 2$, $3x^2 + 5x + 1 = 23$. (This is the same as Exercise 5 in Chapter 7, except that it is now a curried function and must be written as a one-liner.)

These remaining exercises are not one-liners. They should be written without using `map`, `foldl`, or `foldr`.

Exercise 26 Define a function `mymap2` with the same type and behavior as `map`. (This is similar Exercise 24, but that exercise required the use of `foldl` or `foldr`, while this exercise forbids them.)

Exercise 27 Define a function `myfoldr` with the same type and behavior as `foldr`.

Exercise 28 Define a function `myfoldl` with the same type and behavior as `foldl`.