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 7 A Second Look at ML

7.1 Introduction

This chapter continues the introduction to the ML programming language. It will discuss ML patterns, a very important part of the language. You have already used simple ML patterns without realizing it. Using patterns in a more sophisticated way, you can write functions that are both more compact and more readable. This chapter will also show how to define local variables for a function using `let` expressions.

7.2 Patterns You Already Know

ML *patterns* are a very important part of the language. You have already seen several kinds of simple patterns. For example, you have seen that ML functions take a single parameter, like the parameter `n` in this simple squaring function:

```
fun f n = n * n;
```

You have also seen how to specify a function with more than one input by using tuples, like the tuple of two items `(a,b)` in this simple multiplying function:

```
fun f (a, b) = a * b;
```

If the language you are most familiar with is C, C++, or Java, you may have an important misconception to lose. In the previous examples, `n` and `(a,b)` look like a parameter and a parameter list, but they are actually patterns in ML. ML automatically tries to match values with its patterns and takes action depending on whether or not they match. The pattern `n`, for instance, matches any parameter, while the pattern `(a,b)` matches any tuple of two items. Patterns occur in several different parts of ML syntax, not just in function parameters. It is important to start thinking about them as patterns, and not just as simple parameter lists.

Patterns do more than just match data; they also introduce new variables. The pattern `n` matches any parameter and introduces a variable `n` that is bound to the value of that parameter. The pattern `(a,b)` matches any tuple of two items and introduces two variables, `a` and `b`, that are bound to the two components of that tuple. ML supports patterns that are far more powerful than these two simple examples, but they all work by matching data and introducing new variables.

7.3 More Simple Patterns

The simplest pattern in ML is _, the underscore character. It is a pattern that matches anything and does not introduce any new variables. The following example shows a function `f` that takes one parameter and ignores its value. If the value of a parameter doesn't matter, use the _ pattern to match the parameter without introducing a variable.

```
- fun f _ = "yes";
val f = fn : 'a -> string
- f 34.5;
val it = "yes" : string
- f [];
val it = "yes" : string
```

The function `f` can now be applied to any parameter *of any type*. (The function's type, `'a -> string`, indicates that its input can be of any type and its output is always a string.) The function `f` ignores its parameter and always returns the string value `"yes"`. The function could have been defined without the underscore, as in

```
fun f x = "yes";
```

That would introduce an unused variable `x`. In ML, as in most languages, you should avoid introducing variables if you don't intend to use them.

The underscore pattern matches everything. At the other extreme, you can make a pattern that matches just one thing—a constant. The next example defines a function `f` that only works if its parameter is the integer constant `0`:

```
- fun f 0 = "yes";
Warning: match nonexhaustive
          0 => ...
val f = fn : int -> string
- f 0;
val it = "yes" : string
```

The function `f` returns the string value `"yes"` if applied to the constant `0`. No other application of the function is defined. Almost any constant can be used as a pattern. The only restriction is that it must be of an equality-testable type. You can't use real constants as patterns.

The ML language system showed `f`'s type as `int -> string`, but it gave the warning message "match nonexhaustive." The language system is warning us that we defined `f` without using patterns that cover the whole domain type (in this case, the integers). So we have defined `f` in a way that can lead to runtime errors. If `f` is called on an integer value that isn't `0`, it won't work.

```
- f 0;
val it = "yes" : string
- f 1;
uncaught exception Match [nonexhaustive match failure]
```

7.4 Complex Patterns

Any list of patterns is a legal pattern. For example, this function selects and returns the first element from a list of two elements:

```
- fun f [a, _] = a;
Warning: match nonexhaustive
          a :: _ :: nil => ...
val f = fn : 'a list -> 'a
- f [#"f", #"g"];
val it = #"f" : char
```

In the example, `[a,_]` is the pattern for `f`'s parameter. Note the square brackets—this is a list, not a tuple. The pattern `[a,_]` matches any list of exactly two items and introduces the variable `a` (bound to the first element of the list). This is, again, a nonexhaustive function definition; the domain includes lists of any type, but the function will fail if a list has more or less than two elements.

The cons operator (`::`) may also appear in patterns. Any cons of patterns is a legal pattern. For example,

```
- fun f (x :: xs) = x;
Warning: match nonexhaustive
          x :: xs => ...
val f = fn : 'a list -> 'a
- f [1, 2, 3];
val it = 1 : int
```

The pattern `x :: xs` matches any non-empty list and introduces the variables `x` (bound to the head element of the list) and `xs` (bound to the tail of the list). The parentheses around `x :: xs` are not really part of the pattern, but are necessary because of precedence. Function application has higher precedence than cons, so without parentheses ML would interpret `f x :: xs` as `(f x) :: xs`, which isn't what we wanted. This example produced another nonexhaustive function definition. This one is *almost* exhaustive—it matches almost every parameter of the domain type—but it will fail on the empty list.

7.5 A Summary of ML Patterns So Far

Here is a mini-language of patterns in ML:

- A variable is a pattern that matches anything and binds to it.
- An underscore (_) is a pattern that matches anything.
- A constant (of an equality type) is a pattern that matches only that constant value.
- A tuple of patterns is a pattern that matches any tuple of the right size, whose contents match the subpatterns.
- A list of patterns is a pattern that matches any list of the right size, whose contents match the subpatterns.
- A cons of patterns is a pattern that matches any non-empty list whose head and tail match the subpatterns.

You could easily give a BNF grammar for the language of ML patterns (see Exercise 1).

7.6 Using Multiple Patterns for Functions

ML allows you to specify multiple patterns for the parameter of a function, with alternative function bodies for each one. Here's an example:

```
fun f 0 = "zero"
|   f 1 = "one";
```

This defines a function of type `int -> string` that has two different function bodies: one to use if the parameter is 0, the other if the parameter is 1. This definition is still nonexhaustive, since it has no alternative for any other integer.

Below is the general syntax for ML function definitions, allowing multiple patterns. A function definition can contain one or more function bodies separated by the | token.

```
<fun-def> ::= fun <fun-bodies> ;
<fun-bodies> ::= <fun-body> | <fun-body> '|' <fun-bodies>
```

Each function body repeats the function name, but gives a different pattern for the parameter and a different expression:

```
<fun-body> ::= <fun-name> <pattern> = <expression>
```

The same *<fun-name>* must be repeated in each alternative.[1]

Alternate patterns like this can have overlapping patterns. For example, this is legal:

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

Here, the function `f` has one alternative that covers the constant 0 and another that covers all values. The patterns overlap, so which alternative will ML execute when the function is called? ML tries the patterns in the order they are listed and uses the first one that matches.

7.7 Pattern-Matching Style

These two function definitions are equivalent:

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

fun f n =
  if n = 0 then "zero"
  else "non-zero";
```

1. This is redundant. From the language interpreter's point of view, it doesn't add any information to have that function name repeated over and over, and it makes extra work, since the interpreter must check that the function name is the same each time. Repeating the function name does, however, make the function definitions easier for people to read.

The first one is in the pattern-matching style; the second one accomplishes the same thing without giving alternative patterns. Many ML programmers prefer the pattern-matching style, since it usually gives shorter and more legible functions.

This example, without the pattern-matching style, was used back in Chapter 5:

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

It can be written in the pattern-matching style like this:

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

This is easier to read, since it clearly separates the base case from the recursive case. Here is another earlier example that doesn't use the pattern-matching style:

```
fun reverse L =
   if null L then nil
   else reverse(tl L) @ [hd L];
```

It can be written in the pattern-matching style like this:

```
fun reverse nil = nil
|   reverse (first :: rest) = reverse rest @ [first];
```

This shows another advantage of using pattern matching. By matching the compound pattern `(first :: rest)`, we are able to extract the head and tail of the list without having to explicitly apply the `hd` and `tl` functions. That makes the code shorter and easier to read. (It doesn't make it any faster, though. In either case, the ML language system has to find the head and tail of the list.)

Functions that operate on lists often have the same structure as this `reverse` function: one alternative for the base case (`nil`) and one alternative for the recursive case (`first :: rest`), making a recursive call using the tail of the list (`rest`). Any function that needs to visit all the elements of a list will have a similar recursive structure. Suppose we want to compute the sum of all the elements of a list:

```
fun f nil = 0
|   f (first :: rest) = first + f rest;
```

See how the function definition follows the same structure? It just says that the sum of the elements in an empty list is 0, while the sum in a non-empty list is the first element plus the sum of the rest. Suppose we want to count how many `true` values are in a list of booleans:

```
fun f nil = 0
|   f (true :: rest) = 1 + f rest
|   f (false :: rest) = f rest;
```

The inductive case is broken into two parts, one to use if the first element is true and one if the first element is false. But you can still see the underlying structure. Here's one more example. Suppose we want to make a new list of integers in which each integer is one greater than it was in the original list:

```
fun f nil = nil
|   f (first :: rest) = first + 1 :: f rest;
```

This same pattern-matching structure occurs in many of the exercises at the end of this chapter.

Patterns in ML are powerful, but they have one important restriction: the same variable name cannot be used more than once in a pattern. For example, suppose you want to write a function that works on pairs and does one thing if the two elements are equal and something else if they are not. You might try to write it like this:

```
fun f (a, a) = … for pairs of equal elements
|   f (a, b) = … for pairs of unequal elements
```

But the pattern `(a,a)` is illegal because it uses the same variable name more than once. The only way to write this is in a non-pattern-matching style:

```
fun f (a, b) =
  if (a = b) then … for pairs of equal elements
  else … for pairs of unequal elements
```

Patterns occur in many places in ML programs. For instance, you can use patterns in `val` definitions like this:

```
- val (a, b) = (1, 2.3);
val a = 1 : int
val b = 2.3 : real
- val a :: b = [1, 2, 3, 4, 5];
val a = 1 : int
val b = [2,3,4,5] : int list
```

Notice in the last example that the pattern `a :: b` does not cover all lists—in particular, it does not cover the empty list. In other contexts this might cause a problem, but it does no harm here since the list that is matched to the pattern is not the empty list.

Later chapters will show additional ML constructs that use patterns.

7.8 Local Variable Definitions

So far, we have used only two kinds of variable definitions: `val` definitions at the top level and the variables defined by patterns for function parameters. There is a way to make additional, local variable definitions in ML, using the `let` expression. A `let` expression looks like this:

<let-exp> `::= let` *<definitions>* `in` *<expression>* `end`

The *<definitions>* part is a sequence of any number of definitions, such as `val` definitions, that hold only within the *<let-exp>*. The *<expression>* is evaluated in an environment in which the given definitions hold. The value of the *<expression>* is then used as the value of the entire *<let-exp>*. Here is an example:

```
- let val x = 1 val y = 2 in x + y end;
val it = 3 : int;
- x;
Error: unbound variable or constructor: x
```

The expression evaluates `x + y` in an environment in which `x` is 1 and `y` is 2. That value, 3, is the value of the entire `let` expression. The definition for `x` is not permanent. It is local to the `let` expression. Variables defined in a `let` expression between `let` and `in` are visible only from the point of definition to `end`.

For readability, you usually would not write a `let` expression all on one line. Rather, you would break it up and indent it like this:

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

Some ML programmers put a semicolon after each definition. This is optional.

One reason for using a `let` expression is to break up a long expression and give meaningful names to the pieces. For example, this function converts from days to milliseconds:

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

Each definition in the let part can be used in subsequent definitions, as well as in the final expression of the in part.

When you use let to define variables, you can use pattern matching at the same time to extract the individual parts of compound values. Consider this halve function. It takes a list parameter and returns a pair of lists, each containing half the elements of the original list.

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a :: b :: cs) =
      let
        val (x, y) = halve cs
      in
        (a :: x, b :: y)
      end;
```

Notice how the val defines both x and y by pattern matching. The recursive call to halve returns a pair of lists, and the val definition binds x to the first element of the pair and y to the second. The let expression in that function could have been written like this instead:

```
let
  val halved = halve cs
  val x = #1 halved
  val y = #2 halved
in
  (a :: x, b :: y)
end;
```

The first version, using pattern matching, is more compact and easier to read. (In general, if you find yourself using the # notation to extract an element from a tuple, think twice. You can usually get a better solution using pattern matching.)

The halve function divides a list into a pair of half-lists. Here are some examples of halve in operation:

```
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1, 2];
val it = ([1],[2]) : int list * int list
- halve [1, 2, 3, 4, 5, 6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```

To better understand how halve works, let's break it down into its three alternatives:

`halve nil = (nil, nil)`	The `halve` of an empty list is, of course, just two empty half-lists.
`halve [a] = ([a], nil)`	The `halve` of a one-element list puts that one element in the first half-list and no elements in the second half-list.

`halve (a :: b :: cs) =`
`  let`
`    val (x, y) = halve cs`
`  in`
`    (a :: x, b :: y)`
`  end;`
	The `halve` of a list of two or more elements is computed recursively. It first gets the `halve` of the rest of the list, after the first two elements. Then it adds the first element to the first half-list and the second element to the second half-list.

The `halve` function is part of a simple merge-sort implementation. For more practice with ML, let's go ahead and see the rest of the merge sort. Here is a function that merges two sorted lists of integers:

```
fun merge (nil, ys) = ys
|   merge (xs, nil) = xs
|   merge (x :: xs, y :: ys) =
      if (x < y) then x :: merge(xs, y :: ys)
      else y :: merge(x :: xs, ys);
```

(The type of this `merge` function is `int list * int list -> int list`. ML infers this type because integers are the default type for the `<` operator.) The `merge` function takes a pair of sorted lists and combines them into a single sorted list:

```
- merge ([2], [1, 3]);
val it = ([1,2,3]) : int list
- merge ([1, 3, 4, 7, 8], [2, 3, 5, 6, 10]);
val it = [1,2,3,3,4,5,6,7,8,10] : int list
```

To better understand how `merge` works, let's break it down into its three alternatives.

`merge (nil, ys) = ys`	The `merge` of two lists, if the first one is empty, is just the second one.
`merge (xs, nil) = xs`	The `merge` of two lists, if the second one is empty, is just the first one.

`merge (x :: xs, y :: ys) =`
`  if (x < y) then x :: merge(xs, y :: ys)`
`  else y :: merge(x :: xs, ys);`
	The `merge` of two non-empty lists is computed recursively. The smallest element is attached to the front of the recursively computed `merge` of the remainder of the elements.

Now with a `halve` and a `merge` function, we can easily create a merge sort. Here, again, we take advantage of the `let` expression in ML:

```
fun mergeSort nil = nil
|   mergeSort [a] = [a]
|   mergeSort theList =
      let
        val (x, y) = halve theList
      in
        merge(mergeSort x, mergeSort y)
      end;
```

This `mergeSort` function sorts lists of integers:

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

To better understand how `mergeSort` works, let's break it down into its three alternatives.

`mergeSort nil = nil`	An empty list is already sorted. `mergeSort` just returns it.
`mergeSort [a] = [a]`	A list of one element is already sorted too. `mergeSort` just returns it.

`mergeSort theList =`
`  let`
`    val (x, y) = halve theList`
`  in`
`    merge(mergeSort x, mergeSort y)`
`  end;`
	To sort a list of more than one element, `halve` it into two halves, recursively sort the halves, and `merge` the two sorted halves.

7.9 Nested Function Definitions

The previous functions halve and merge are not very useful by themselves; they are really just helpers for mergeSort. They could be locally defined inside the mergeSort definition like this:

```
(* Sort a list of integers. *)
fun mergeSort nil = nil
|   mergeSort [e] = [e]
|   mergeSort theList =
      let
        (* From the given list make a pair of lists
         * (x, y), where half the elements of the
         * original are in x and half are in y. *)
        fun halve nil = (nil, nil)
        |   halve [a] = ([a], nil)
        |   halve (a :: b :: cs) =
              let
                val (x, y) = halve cs
              in
                (a :: x, b :: y)
              end;

        (* Merge two sorted lists of integers into
         * a single sorted list. *)
        fun merge (nil, ys) = ys
        |   merge (xs, nil) = xs
        |   merge (x :: xs, y :: ys) =
              if (x < y) then x :: merge(xs, y :: ys)
              else y :: merge(x :: xs, ys);

        val (x, y) = halve theList
      in
        merge(mergeSort x, mergeSort y)
      end;
```

As this example shows, fun definitions can be made inside let, just like val definitions. The effect is to define a function that is visible only from the point of the definition to the end of the let. This organization has the advantage of making halve and merge invisible to the rest of the program, which makes it clear to the reader that they will be used in only this one place. There is another potential advantage to this nesting of functions, which was not used in the example above: the inner functions can refer to variables from the containing function. We'll see more about this in Chapter 12.

The previous example also shows the use of comments in ML. Comments in ML programs start with `(*` and end with `*)`. In ML, as in all other programming languages, programmers use comments to make programs more readable.[2]

7.10 Conclusion

This chapter introduced ML patterns and the pattern-matching style for function definitions. It introduced the ML `let` expression for local function definitions. A long merge-sort example demonstrated how to use both patterns and `let` expressions in ML. This chapter also showed how to write local function definitions in ML and how to write comments.

Exercises

Exercise 1 Give a BNF grammar for the language of ML patterns. Use the non-terminal symbol <*pattern*> as the start symbol. Use the non-terminal symbols <*name*> and <*constant*>, without defining productions for them, for the appropriate parts of the language.

In all the following exercises, wherever possible, practice using pattern-matching function definitions. With some SML/NJ installations, you will encounter a new warning message while solving these exercises: "`Warning: calling polyEqual`." This warning indicates that your code is comparing two values for equality, at a point where the compiler doesn't know the runtime types of those values. This is perfectly legal in ML, and in fact is required for some of the exercises below. The compiler gives a warning message only because it is a somewhat inefficient operation. In general, it is safe to ignore this warning message.

Do not, however, ignore any other error or warning messages. In particular, you should make complete function definitions that do not give rise to any "`match nonexhaustive`" warnings.

Exercise 2 Define 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 the list `L`.

2. Some programmers claim that their ML programs are so clearly written as to be "self-documenting," requiring no comments. Indeed, some programmers have made this claim for their code in just about every programming language ever invented. But no program is ever as self-documenting as it seems to its author.

Exercise 3 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`.

Exercise 4 Define a function `repeats` of type `''a list -> bool` so that `repeats(L)` is true if and only if the list `L` has two equal elements next to each other.

Exercise 5 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 $x = 2$, $3x^2 + 5x + 1 = 23$.

Exercise 6 Write a `quicksort` function of type `int list -> int list`. Here's a review of the quicksort algorithm. First pick an element and call it the pivot. (The head of the list is an easy choice for the pivot.) Partition the rest of the list into two sublists, one with all the elements less than the pivot and another with all the elements not less than the pivot. Recursively sort the sublists. Combine the two sublists (and the pivot) into a final sorted list.

Exercise 7 Functions can be passed as parameters just like other values in ML. For example, consider these function definitions:

```
fun square a = a * a;
fun double a = a + a;
fun compute (n, f) = f n;
```

The functions `square` and `double` take a single `int` parameter and return an `int` result. The function `compute` takes a value `n` and a function `f`, and returns the result of calling that function `f` with `n` as its parameter. So `compute(3,square)` evaluates to 9, while `compute(3,double)` evaluates to 6. Chapter 9 will explore this important aspect of ML in more detail. For this exercise, you need only the simple function-passing technique just illustrated.

Make another version of your `quicksort` function, but this time of type `'a list * ('a * 'a -> bool) -> 'a list`. The second parameter should be a function that performs the role of the < comparison in your original function. (*Hint:* This should require only minor changes to your previous `quicksort` definition.)

Why would you want to define such a function? Because it is much more useful than the original one. For example, suppose you defined `icmp` and `rcmp` like this:

```
fun icmp (a, b) = a < b;
fun rcmp (a : real, b) = a < b;
```

You could now use `quicksort(L, icmp)` to sort an integer list `L`, and you could use `quicksort(M, rcmp)` to sort a real list `M`. And if you defined

```
fun ircmp (a, b) = a > b;
```

then you could use `quicksort(L, ircmp)` to sort the integer list `L` in reverse order.

In the following exercises, implement sets as lists, where each element of a set appears exactly once in the list and the elements appear in no particular order. Do not assume you can sort the lists. Do assume that input lists have no duplicate elements, and do guarantee that output lists have no duplicate elements.

Exercise 8 Write a function to test whether an element is a `member` of a set.

Exercise 9 Write a function to construct the `union` of two sets.

Exercise 10 Write a function to construct the `intersection` of two sets.

Exercise 11 Write a function to construct the `powerset` of any set. A set's powerset is the set of all of its subsets. Consider the set $A = \{1,2,3\}$. It has various subsets: $\{1\}$, $\{1,2\}$, and so on. Of course the empty set, $\varnothing$, is a subset of every set. The powerset of A is the set of all subsets of A:

$$\{x \mid x \subseteq A\} = \{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$$

Your `powerset` function should take a list (representing the set) and return a list of lists (representing the set of all subsets of the original set). `powerset [1,2]` should return `[[1,2],[1],[2],[]]` (in any order). Your `powerset` function need not work on the untyped empty list; it may give an error message when evaluating `powerset nil`. But it should work on a typed empty list, so `powerset (nil : int list)` should give the right answer (`[[]]`).