

Chapter 1

ML fundamentals

ML is a relatively new language, employing many modern language design concepts. If you’ve programmed before, you’ll find ML different in two important respects. (And if you haven’t programmed before, because of these differences, you won’t be at much of a disadvantage.)

ML is a **functional language**. ML programs strongly emphasize *functions* similar to mathematical functions. Functional programming does *not* include the notion of statements found in Ada, C++, or Java. The whole notion of statement is altogether lost, replaced with solely functions and expressions. This proves to be a radically different way of doing things. And it works.

Also, we will use an interactive **interpreter** to write our ML programs. This program is called Standard ML of New Jersey, abbreviated SML. To start SML, type “sml” at the Unix prompt.

```
% sml
Standard ML of New Jersey, Version 110.0.7, September 28, 2000
val use = fn : string -> unit
-
```

The minus sign is the prompt, signaling that SML is ready for you to give it some ML to interpret.

1.1 Identifiers

ML uses **identifiers** to represent values. An identifier has a name, a type, and a value. In this respect, an ML identifier is very similar to an Ada variable. But in one important respect they are different: *the value of an ML identifier never changes*. That is, when ML creates an identifier, it gives the identifier a value, and that identifier *always* stands for that value.

Let’s type something very simple for SML to interpret.

```
- val freezing = 32;
val freezing = 32 : int
```

In this line, we have told SML to create a new identifier, named `freezing`, with a value of 32. The `val` word is special to ML and designates that you are about to create an identifier. After that comes the identifier’s name, and then after the equals sign is the value to give it. A semicolon ends the line, telling SML to go ahead.

In this example, SML came back saying “`val freezing = 32 : int.`” SML’s response simply confirms that a new identifier `freezing` has been created, representing the value 32, with a type of `int` (for integer).

An ML integer is completely incomparable to its floating-point equivalent. To designate a floating-point number to ML, you can use a decimal point.

```
- val real_freezing = 32.0;
val real_freezing = 32.0 : real
```

Note that when SML confirms the creation of the identifier, it prints `real` to signify that this identifier represents a floating-point number.

If you simply write an expression in ML followed by a semicolon, the interpreter will assume that you meant to create an identifier named `it` with that value.

```
- freezing + 5;
val it = 37 : int
```

In this example, we wrote a very simple expression saying to add 5 to the value of the identifier `freezing`. SML evaluated this expression, determined that the answer was 37, and handled it *exactly* as if we had typed “`val it = 37;`”

```
- val it = 37;
val it = 37 : int
```

There’s nothing special about the `it` identifier, except for this implicit redefinition of `it` every time you write an expression.

ML doesn’t prevent you from creating new identifiers with the same name as existing identifiers. But *do not* think of this as *changing* an identifier’s value (a la Ada assignment statements): You are creating a *completely new* identifier, whose name happens to be the same as an old one, and which therefore hides the old identifier from future use.

```
- val freezing = 0.0;
val freezing = 0.0 : real
- freezing;
val it = 0.0 : real
```

In this example, we created a new identifier `freezing` with a value of 0.0. (Notice how this new identifier’s type is `real`.) Then we wrote the expression `freezing`, and we found that this expression evaluates to 0.0.

1.2 Expressions

Basically everything in ML is either an expression or an identifier definition. You can build simple expressions using the following rules.

- A constant (like 32 or 0.0) is an expression whose value is what that constant represents.
- An identifier (like `freezing`) is an expression whose value is the value that that identifier represents.
- A smaller expression may be enclosed in parentheses.
- Two smaller expressions may be joined by a **binary operator** (like ‘+’, ‘-’, or ‘*’).
- An expression may be preceded by a **unary operator**. (For example, ML uses ‘~’ for negation: “~(1 + 3)” represents the integer -4. The ML designers chose to use ‘~’ for negation because they wanted different symbols for negation and for subtraction.)

ML includes many operators, and it uses the algebraic order of operations.

- At the top is the negation operator ‘~’.

- Then come the multiplication and division operators: ‘*’, ‘/’, ‘div’, and ‘mod’. (ML uses the ‘/’ for dividing two *real* numbers, and it uses ‘div’ for dividing two *integer* numbers (in which case any remainder is ignored). The ‘mod’ operator works on two integers and its value is the remainder when the two integers are divided.
- Below this are addition (+) and subtraction (-).

Here are examples of a few expressions and their corresponding values.

<code>2 + 3 * 5</code>	<code>17</code>
<code>~(3.0 / 2.0 * 8.0) + 1.5</code>	<code>13.5</code>
<code>15 mod (3 + 4)</code>	<code>1</code>

Sometimes, you’ll accidentally write an expression that has a type error of some sort. SML uses inimitable techspeak for describing these errors; it takes some practice to be able to decode the messages.

```
- 2 * 1.5;
stdIn:6.1-6.8 Error: operator and operand don't agree [literal]
operator domain: int * int
operand:         int * real
in expression:
  2 * 1.5
```

The phrase “operator and operand don’t agree” is an indicator that you’re looking at a type error. Then SML says that it expected two integers (“operator domain” is a phrase meaning “expected”), and it saw an integer and a real (it uses the phrase “operand” to indicate what it saw). The problem here is that we’re trying to multiply an integer by a real, and multiplication can only handle either two integers or two reals. (The interpreter took you to mean to multiply two ints, since the left-hand side of the multiplication is an int.)

1.3 Functions

With what we’ve seen so far, ML is just a glorified calculator. To do any programmed computation, we need the notion of a *function*. Functions are integral to ML — so important, in fact, that ML is classified as a *functional* language.

Each function takes as an input some value called a **parameter** or an argument. Some functions will take several parameters. Then the function does some computation to compute its output, called its **return value**. The process of computing the return value could possibly be a complex process.

1.3.1 Using functions

To use a function, you can just write the function name followed by the value of its parameter. For example, built into ML is a function named `Math.sqrt`. This function takes a real number as a parameter and returns a real number (which approximates the square root of its parameter).

Let’s say we want to use this function to compute $\sqrt{5}$.

```
- Math.sqrt 5.0;
val it = 2.2360679775 : real
```

In the order of operations, function application comes second, just below negation and above the multiplication and division operators. Consider the following example.

```
- Math.sqrt 25.0 * 25.0
```

Because function application comes above multiplication in the order, the first thing that happens here is the call to `Math.sqrt`, and then the result gets multiplied by 25.0. So the interpreter responds with 125 (not 25, as it would if it did the multiplication first and then the square root).

```
val it = 125.0 : real
```

ML includes a variety of built-in functions for you to use in building your own functions. We can't possibly list them all yet — we need to understand ML better — but here are a few.

<code>real</code>	<code>int -> real</code>	returns the floating-point equivalent of its integer argument
<code>round</code>	<code>real -> int</code>	returns the nearest integer to its floating-point argument
<code>Math.sqrt</code>	<code>real -> real</code>	returns the square root of its argument
<code>Math.exp</code>	<code>real -> real</code>	returns e^x for an argument x

Notice that the first two give you a way of converting between integers and floating-point numbers.

1.3.2 Defining functions

Defining functions is relatively simple — not too different from defining mathematical functions.

```
- fun tempConv celsius = 9 * celsius div 5 + 32;
val tempConv = fn : int -> int;
```

Here we are using the word `fun` to signal that we are defining a function. The name following it, `tempConv`, is the name of the function, and the name following it, `celsius`, is a placeholder for its parameter value. Following the equal sign is an expression saying how to compute the return value of the function for a given `celsius`.

Now if want to use the function, we type the function name followed by what we want it to use for `celsius`.

```
- tempConv 100;
val it = 212 : int
```

Here we passed the number 100 as `tempConv`'s parameter. The function computes `9 * 100 div 5 + 32` — which is 212. This is the value of the expression `tempConv 100`.

Look again at how SML responded when we first defined the `tempConv` function.

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

It is saying that the user has defined a new identifier, called `tempConv`, whose value is a function. (Rather than attempt to represent the function on the screen, the SML interpreter just displays the word `fn`.) The type of the `tempConv` identifier is `int -> int` — it's a function from integers to integers.

In other words, SML's response indicates that defining functions is really just a different way of defining an identifier. This in itself is interesting. But there's another interesting point arising from SML's response: How did it determine that `celsius` would represent an integer?

1.3.3 Type inference

The answer: It assumed it! ML has a unique feature called **type inference**: The ML interpreter will analyze a function to determine what type you probably meant, and it will go with that assumption. Usually — surprisingly often — it will assume correctly. In this case it saw that we were multiplying 9 by `celsius`: since 9 is an integer, and you can only multiply an integer by an integer, it infers that `celsius` must be an integer. Similarly, because it adds two integers to get the result, and the sum of two integers is an integer, it concludes that the return type must be an integer. This isn't so impressive, but we will see some more amazing examples.

Of course, sometimes the interpreter will assume wrongly. Consider the following example.

```
- fun sq x = x * x;
val sq = fn : int -> int
```

In this case, it is ambiguous whether we wanted `sq` to work with integers or real numbers — the only thing we did with `x` was to multiply it with itself, and either interpretation is consistent. The ML interpreter assumed `int`. If we actually wanted `x` to be `real`, we have two options: The low-tech version is to multiply by `1.0`.

```
- fun sq x = 1.0 * x * x;
```

In the other option, we include type information in the function definition.

```
- fun sq x : real = x * x;
```

In this case, we are saying that `sq` returns a `real`. Since we can only get a `real` out of a multiplication if we are multiplying a `real` by a `real`, the interpreter infers that `x` must be a `real` also.

1.3.4 Multiple parameters

The functions we have seen so far deal only with single parameters. Often you want a function that takes more parameters. For example, say we want a function that takes four parameters representing the x - and y -coordinates of two points (x_0, y_0) and (x_1, y_1) , and we want to compute the distance between them. Recall that the distance is $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$. We'll use our `sq` function in writing the ML for this.

```
- fun dist x0 y0 x1 y1 = Math.sqrt (sq (x0 - x1) + sq (y0 - y1));
val dist = fn : real -> real -> real -> real -> real
```

The ML interpreter responds that we have defined a function `dist` that takes four `real` parameters and returns a `real`.

Now if we want to use the distance function, we can just list the parameters, one after the other. ML will associate the values with the parameters listed in the function definition according to the order they are listed.

```
- dist 0.0 0.0 3.0 4.0;
val it = 5.0 : real
```

Behind the scenes, the interpreter computed `x0 - x1` (which was `-3.0`) and squared this by passing `-3.0` to the `sq` function we just defined (`9.0`). Then it computed `y0 - y1` (which is `-4.0`) and computed its `sq` (`16.0`). It added these together, and passed the result (`25.0`) to the built-in function `Math.sqrt`, which computed the square root (`5.0`, the value returned by `dist` in this example).

Here's a more systematic representation of how the interpreter reasons. We'll call this a **derivation** of the function's value.

```
dist 0.0 0.0 3.0 4.0
= Math.sqrt (sq (0.0 - 3.0) + sq (0.0 - 4.0))
= Math.sqrt (sq ~3.0 + sq (0.0 - 4.0))
= Math.sqrt (~3.0 * ~3.0 + sq (0.0 - 4.0))
= Math.sqrt (9.0 + sq (0.0 - 4.0))
= Math.sqrt (9.0 + sq ~4.0)
= Math.sqrt (9.0 + ~4.0 * ~4.0)
= Math.sqrt (9.0 + 16.0)
= Math.sqrt 25.0
= 5.0
```

Be sure to step through the above sequence yourself — understanding the above sequence is important for later.

1.4 The Boolean type

ML provides a variety of types. So far we have just seen two: `int` and `real`. But next on our list are the **Boolean** values, named `bool` by ML. These are for values that can be only true or false.

ML has a few additional operators for dealing with Boolean values.

1. The comparison operators are just below addition and subtraction in the evaluation order: equals (`'='`), not-equals (`'<>'`), less-than (`'<'`), less-than-or-equals (`'<='`), greater-than (`'>'`), and greater-than-or-equals (`'>='`). These take two numbers and return a Boolean value.
2. Below this, there are the two operators `andalso` and `orelse`, for combining Boolean values together using the AND or OR of Boolean logic.

An observant reader should be wondering: If we have AND and OR from Boolean logic, why not NOT? The answer is that ML has this too, it's just not a special operator. One of the built-in functions is `not`, which takes a `bool` parameter and returns a `bool` — namely, the opposite of its parameter. Notice what this implies in terms of the order of operations: functions are just above multiplication in the order, and so the Boolean NOT is above the comparison operators. This is somewhat counterintuitive.

We can use these in defining identifiers of type `bool`.

```
- val flag = not (5 > 8);
val it = true : bool
```

1.5 if expressions

We'll use Boolean expressions most often in `if` expressions.

```
- val abs x = if x >= 0.0 then x else ~x;
val abs = fn : real -> real
```

Here we have defined a function called `abs`, which takes a parameter `x` and returns its absolute value — x if x is at least 0.0, and $-x$ if not.

A warning about using `if`: Every time you use an `if`, you **must** have an `else`! This is because the ML `if` is an *expression*, and expressions must always have a value in all circumstances.

An `if` is an expression like any other; you can put it anywhere you can put another expression.

```
- fun len x y = (if x < y then ~1 else 1) * (x - y);
val len = fn : int -> int -> int
- len 5 8;
val it = 3 : int
```

In this example, the `if` computes -1 (since $x < y$), and so `len` returns $-1 \cdot (x - y) = -1 \cdot -3 = 3$.

Often, you'll want to put an `if` expression inside the `else` clause of another.

```
- fun gradePoint pts =
=   if pts >= 90.0 then 4
=   else if pts >= 80.0 then 3
=   else if pts >= 70.0 then 2
=   else if pts >= 60.0 then 1
=   else 0;
val gradePoint = fn : real -> int
```

This example introduces nothing new about `if`: We just placed a regular `if` in the `else` of another `if`, which was itself in the `else` of another `if`, which was itself in the `else` of another `if`. (The equal signs on the left side are the prompts SML prints when it is waiting for you to type more, because it knows you haven't finished saying what you wanted to say yet. They're not part of the ML function.)

Chapter 2

Recursion

At this point, we have a way of having ML compute simple expressions. But complex operations involve doing essentially the same thing repeatedly. In ML we accomplish this by using **recursion** — which basically means we have a function use itself.

2.1 Recursion at work

Recursion seems circular at first — how could a self-referential function mean anything useful? — but in controlled circumstances it can do something.

Consider the following function to compute the factorial of n , $n!$ (that is, $1 \cdot 2 \cdot 3 \cdots (n - 1) \cdot n$).

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

This is recursive, because in some circumstances (namely, when n exceeds 1), it uses itself to compute its own value.

Let's see how the ML interpreter would compute `fact 4`.

```
fact 4 = 4 * fact (4 - 1)  
       = 4 * fact 3  
       = 4 * (3 * fact (3 - 1))  
       = 4 * (3 * fact 2)  
       = 4 * (3 * (2 * fact (2 - 1)))  
       = 4 * (3 * (2 * fact 1))  
       = 4 * (3 * (2 * fact 1))  
       = 4 * (3 * (2 * 1))  
       = 4 * (3 * 2)  
       = 4 * 6  
       = 24
```

We'll call this technique for representing how a recursive function evaluates its value a **derivation**.

2.2 The inductive method

We'll see two general structures for recursive programs. The first is the *inductive method*, which is the technique you should consider first. If it looks like the inductive method doesn't work, then you can move onto thinking about the helper-function method of the next section.

2.2.1 Writing a recursive program

Our second example is really not too different, but we'll go through the design process more carefully: We want to write a function that takes an integer n as a parameter and computes the sum of the squares from 1 to n (that is, $1^2 + 2^2 + 3^2 + \dots + (n-1)^2 + n^2$). Before you go on, think for a moment about how you would do this using ML.

The program we write will have the form

```
- fun sumSquares n = ...
```

The thought process you can go through to write this program is the following: If I'm looking at a particular n (say, $n = 10$), and I knew the function's value for all smaller values of n (so I know the sum of the squares up to 1^2 , ... up to 8^2 , and up to 9^2), how would that help me compute the function value for n ?

What you should see is that the sum of the squares up to 10^2 is of course the sum of the squares up to 9^2 , plus 10^2 . In this case, if I know the value of `sumSquares (n - 1)`, it is not too difficult to find the value of `sumSquares n` — namely, I just need to add n^2 to it. This leads to the following first-cut solution.

```
- fun sumSquares n = sumSquares (n - 1) + n * n;
```

But this function has a major problem. What is it? It may help if you try to do a derivation to see how it computes `sumSquares 3`.

Let's try this.

```
sumSquares 3 = sumSquares (3 - 1) + 3 * 3
              = sumSquares 2 + 3 * 3
              = (sumSquares (2 - 1) + 2 * 2) + 3 * 3
              = (sumSquares 1 + 2 * 2) + 3 * 3
              = ((sumSquares (1 - 1) + 1 * 1) + 2 * 2) + 3 * 3
              = ((sumSquares 0 + 1 * 1) + 2 * 2) + 3 * 3
              = (((sumSquares (0 - 1) + 0 * 0) + 1 * 1) + 2 * 2) + 3 * 3
              = (((sumSquares ~1 + 0 * 0) + 1 * 1) + 2 * 2) + 3 * 3
              = (((sumSquares (~1 - 1) + ~1 * ~1) + 0 * 0) + 1 * 1) + 2 * 2 + 3 * 3
              = (((sumSquares ~2 + ~1 * ~1) + 0 * 0) + 1 * 1) + 2 * 2 + 3 * 3
```

It'll never stop! This is called **infinite recursion**. Any time it occurs, it's an error, because it means the program never computes an answer — and of course a nonexistent answer is never correct. (For the same reason, on multiple choice questions with no penalties for wrong answers, you should always guess.) How can we fix this?

In this case, what's happening is that we're missing a **base case**. For a recursive function, a base case is a situation where we don't make a recursive call to a function. Every recursive function must have at least one base case — a recursive function without any base cases must infinitely recurse no matter what.

In our first-cut solution, there is no base case — in all situations we make a recursive call. We need a base case. To determine your base case, consider: What is the simplest possible case for the function? That is, for what could you give an immediate answer without any further thinking?

For fact, the simplest case — and the base case in our definition — was when n was at most 1. In that case, we said, the answer was a flat 1: no recursion needed.

For this problem, too, the simplest case is when the parameter n is 1. We can immediately say, then, that the answer is 1, without any consideration.* Incorporating this base case using `if` gives us a working solution.

*You might argue (and I would) that *actually* the simplest case is when $n < 1$, in which case the answer is 0. But this involves arguing that if I add no numbers together, I get 0, which is confusing.


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

Thus, there are two major questions you should ask yourself when writing a recursive function. The first is: What parameter value is the most trivial case? This will be the base case for your recursive function. The second question you should ask: How would knowing the function values for smaller parameters help me compute the function value for a given value?

Taken together, these two questions constitute what I will call the **inductive technique** for building a recursive function. The inductive technique will not always work — in the next chapter, we'll see some problems where it doesn't, and how we can write programs to solve them. But any time recursion seems necessary, you should first try the inductive technique, and if that fails you can go on to trying other things.

2.2.2 Multiply-recursive functions

The **Fibonacci numbers** are the sequence of numbers 1, 1, 2, 3, 5, 8, 13, ...: Each number is the sum of the preceding two in the sequence. Fibonacci numbers turn up in sunflowers, pine cones, and other equally useful places. We want to write a function `fib` that takes an integer n and computes the n th Fibonacci number.

Employing the inductive method, we ask ourselves: What is the simplest case? In this example, the trivial case — one requiring no computation whatever — is for $n = 1$, when the answer is 1. In this example, there's actually another trivial case for us to handle: $n = 2$. This case, also, requires no computation.

And the second question: If we're trying to compute the n th Fibonacci number, and we know the Fibonacci numbers for $i < n$, how can we do it? In this case, our Fibonacci number definition gives it away: We'll add the $(n - 1)$ st Fibonacci with the $(n - 2)$ nd Fibonacci.

Putting these two answers together, here's our function.

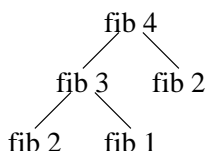
```
- fun fib n = if n = 1 then 1
=           else if n = 2 then 1
=           else fib (n - 1) + fib (n - 2);
val fib = fn : int -> int
```

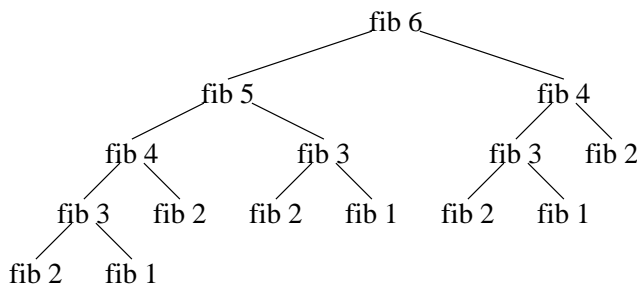
This function is interesting because it calls itself *twice* in order to compute its own value.

How would the interpreter compute `fib 4`? Here's the derivation.

```
fib 4 = fib (4 - 1) + fib (4 - 2)
      = fib 3 + fib (4 - 2)
      = (fib (3 - 1) + fib (3 - 2)) + fib (4 - 2)
      = (fib 2 + fib (3 - 2)) + fib (4 - 2)
      = (1 + fib (3 - 2)) + fib (4 - 2)
      = (1 + fib 1) + fib (4 - 2)
      = (1 + 1) + fib (4 - 2)
      = 2 + fib (4 - 2)
      = 2 + fib 2
      = 2 + 1
      = 3
```

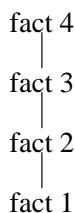
There's another representation of a recursive function's computation that is also useful, called the **recursion tree**.



Figure 2.1: Recursion tree for `fib 6`.

In a recursion tree, each node represents a call to `fib`. Connected below a given node are the recursive calls made by that node in the process of computing its return value. For example, below `fib 4`, we have `fib 3` and `fib 2`, because in order to compute `fib 4`, we had to also compute `fib 3` and `fib 2` (and then we happened to add their return values). Figure 2.1 contains the recursion tree for computing `fib 6`.

For singly-recursive functions like `fact`, the recursion tree isn't very interesting. Here is the recursion tree for `fact 4`.



Derivations and recursion trees are both useful for visualizing how a recursive program works. Since recursion trees are more graphical and concise, they are often more useful.

2.3 The helper-function method

The second technique for writing recursive functions is more of a catch-all technique for applications where the inductive method doesn't work. Sometimes it won't, because sometimes there just won't be an answer to the second question of how knowing the values for lesser parameters would help with computing the value for a given set of parameters.

2.3.1 Using `let-in-end`

Before we look at helper functions in the context of recursion, we need to study `let` expressions. A `let` expression gives you the capability to temporarily define an identifier for the duration of an expression. This is frequently a matter of convenience.

```

fun distanceFallen t =
  let val g = 9.78;
  in 0.5 * g * t * t
  end;

```

(This function computes how far an object will fall towards Earth in a certain number of seconds.) We can have any sequence of identifier and function definitions between `let` and `in`; between `in` and `end` is an expression that computes the value of the overall `let` expression.

As with all of our programs, ML does not care about line breaks and spaces — the line breaks are for the convenience of the human reader, not because the computer needs them to be there.

In this example, between `let` and `in` we define a new identifier `g`, with a value of 9.78. Then between `in` and `end` we indicate how to compute the return value — in this case, $(1/2)gt^2$. What we have done is to define `g` to make the return value expression a little cleaner.

This definition of `g` lasts only for the duration of the `let` expression.

```
- val g = 42;
val g = 42 : int
- distance_fallen 1.0;
val it = 4.89 : real
- g;
val it = 42 : int
```

In this example, even though `g` was defined in the `let` expression within `distanceFallen`, this did not affect the `g` we had defined outside the `let` expression.

You can put several definitions between `let` and `in` when it is convenient. The following computes the distance traveled by a projectile fired at a given velocity and angle. (That is, it does if my physics hasn't rusted out.)

```
fun distance vel angle =
  let
    val g = 9.78;
    val horz_vel = vel * Math.cos angle;
    val vert_vel = vel * Math.sin angle;
    val time_aloft = 2.0 * vert_vel / g;
  in
    time_aloft * horz_vel
  end;
```

(Notice that our definition of `time_aloft` uses values defined earlier in the `let` expression.) By using `let`, we can break the computation into more manageable pieces. This definition is much easier to understand than the equivalent function that doesn't use `let`

```
fun distance vel angle =
  (2.0 * vel * Math.sin angle / 9.78) * (vel * Math.cos angle);
```

You can place any number of identifier definitions between `let` and `end`. Function definitions can also appear here. (After all, we've already seen that ML really thinks of a function definition as a slightly different form of identifier definition.) In the helper-function method for constructing a recursive function, in fact, we'll use `let` to define subfunctions.

2.3.2 Computing primality

Consider the following problem: We want to develop a function that takes an `int` as a parameter and returns a `bool` indicating whether the given integer is prime or not.

The inductive method doesn't help here. If I want to know whether 143 is prime, telling me which numbers before 143 are prime isn't extremely helpful. To determine primality, I really need to try several numbers out, to see if any divide into 143.

What we'll do is to define a different function within a `let`. This helper function will exist solely to iterate through a sequence of numbers to see whether any divide into 143 (or whatever the parameter happens to be). Here's the function.

```

fun isPrime num =
  let
    fun testFactor i =
      if i * i > num then true
      else if num mod i = 0 then false
      else testFactor (i + 1);
  in
    testFactor 2
  end;

```

The `testFactor` helper function takes a parameter `i` on which we're currently working, and sees whether we've passed $\sqrt{\text{num}}$ yet. If we have, then we've tested all the possibilities with no success and so we know `num` must be prime and so we return `true`. If we haven't, we see whether this `i` divides into `num`. If it does, then we know `num` isn't prime and so we return `false`. If neither of these apply, then we make a recursive call to `testFactor` with the parameter `i` being one more than before.

Let's do a derivation to see how this works for `isPrime 77`.

```

isPrime 77 = testFactor 2
           = testFactor 3
           = testFactor 4
           = testFactor 5
           = testFactor 6
           = testFactor 7
           = false

```

In this example, `testFactor` continued making recursive calls until it reached 7, at which point it found that `num mod i` was zero, and so it returned `false`.

2.3.3 Helper-function technique defined

This is the helper-function technique: The inductive method didn't work, because we really need to refer to the parameter in the process of determining the answer. (In this case, we really needed to try dividing the `isPrime` parameter by several possibilities.) So we define a helper function inside a `let` expression to go through the repetition needed to accomplish the task required.

Essentially, the helper-function technique gives a way of translating a loop (a la imperative programming) into a recursive function. In the case of primality testing, we might write an imperative program as follows.

```

i := 2;
loop
  if i * i > num then
    return true;
  elsif num mod i = 0 then
    return false;
  else
    i := i + 1;
  end if;
end loop;

```

We translated this into a helper function using the following technique. The helper function corresponds to the loop of this program. The loop uses a single variable `i` in its processing, so our helper function has a single parameter `i`. Because we initialize `i` to be 2 before entering the loop, we use `testFactor 2` to enter the helper function (in the `in` part of the `let` expression). And the helper function is a systematic translation of the body of the loop: If the loop continues for another iteration, this translates to a recursive call with the updated parameter/variables. If the loop exits, then this corresponds to a base case for the helper function's recursion.

When the inductive technique is feasible, it is preferable to the helper-function technique, as it is typically easier to understand and more in line with the philosophy of functional programming. If you are an inveterate imperative programmer, you may initially have difficulty with the inductive technique, but you'll be a better programmer (even a better imperative programmer) for understanding it. Don't use the helper-function technique as a crutch for simulating imperative programs in a functional programming language.

On the other hand, the similarity between loops and helper functions leads to an interesting result: Anything you can do with an imperative programming language can also be done in a functional language. Proving this involves showing a technique that takes an imperative program and systematically translates it into a functional program. The resulting program won't be beautiful — and it certainly won't be in line with the functional programming philosophy. But the helper-function technique is the essential ingredient to this proof.

2.3.4 Computing a square root

Let's look at another example: computing the square root of a number x . We've already seen that there's a built-in function `Math.sqrt` that accomplishes this, but let's see how we might build our own function to do a similar thing.

We'll accomplish this using **binary search**: We'll start with a wide range where the square root might lie, and we'll successively halve it until it is very narrow. To narrow the range, we'll choose the middle number in the range and determine whether it lies above or below the square root by comparing its square to x .

This technique can't be coded using the inductive technique, because it requires in the processing a reference to x (to determine on which side of the square root the middle number lies). So we'll have to use the helper-function method.

In this example, the helper function will have *two* parameters, representing the lower and upper bounds of the current range.

```
fun squareRoot x =
  let
    fun binarySearch low high =
      let
        val m = (low + high) / 2.0;
      in
        if m * m > x + 0.0001 then binarySearch low m
        else if m * m < x - 0.0001 then binarySearch m high
        else m
      end;
    in
      binarySearch 0.0 x
    end;
```

Within the helper function, we compute the middle number m between the parameters `low` and `high`. Then we see if m^2 is too much more than x . If it is, we make a recursive call to search in the lower half of the range. If it is too much less than x , we make a recursive call to the upper half of the range. And otherwise (if m^2 is very close to x), we return the number m .

2.3.5 Fibonacci revisited

Let's look at an alternative implementation of Fibonacci number computation. If you look at Figure 2.1, you may notice something peculiar: In the course of computing `fib 6`, we computed `fib 3` three times. This is quite a waste.

In fact, if you think about how `fib` works, to compute `fib n`, it adds 1 to itself `fib n` times: the return values for a leaf of the tree (that is, a base case) is 1, while the return value for a non-leaf is just the sum of the return values for its two children. So the return value of each node is just a count of the number of leaves descended from that node.

This is quite slow: The n th Fibonacci number, it turns out, is quite close to $1.618^n / \sqrt{5}$. (Here, 1.618 is actually the *golden ratio*, $(1 + \sqrt{5})/2$.) The speed of this program is proportional to this, so it has exponential growth. In fact, I timed our earlier definition of `fib` program on my computer and found the following.

```
fib 30    took 0.112 seconds
fib 35    took 1.280 seconds
fib 40    took 14.851 seconds
fib 60    took 7 years, 1 month, 12.5 days
```

(I'm just guessing on that last one; I didn't actually wait that long.)

There's another algorithm for computing Fibonacci numbers, following the way you might actually list them: Work from the bottom and go upwards. That is, compute 2 by adding the previous two Fibonacci; then compute 3; then 5; then 8, 13, 21, etc. The inductive technique won't lead you to this algorithm, but you can use the helper-function technique to write a function using this algorithm. In this helper function, we'll use three parameters, indicating which Fibonacci we're currently on, what that Fibonacci is, and what the previous Fibonacci is.

```
fun fastFib n =
  let
    fun ithFib i this prev : real =
      if i >= n then this else ithFib (i + 1) (this + prev) this;
  in
    ithFib 2 1 1
  end;
```

Here we're saying that if we've reached the desired Fibonacci, the helper function can just return the current Fibonacci. Otherwise, we go to the next Fibonacci by making a recursive call for the next number of the sequence (that is, `this + prev`), where `this` is the Fibonacci preceding it.

The timings for this program are much more reasonable.

```
fastFib 30    took 0.77 milliseconds
fastFib 35    took 0.88 milliseconds
fastFib 40    took 0.99 milliseconds
fastFib 60    took 1.49 milliseconds
```

As you can see, the growth of `fastFib`'s running time is *linear* as opposed to `fib`'s *exponential* growth. With `fastFib`, computing 5 more Fibonacci takes an additional 0.5 milliseconds, whereas `fib` takes 11 times more time to accomplish the same thing.

Chapter 3

More about ML functions

Eventually we'll get into working with more complex data than just numbers and Booleans, but first we need to revisit functions and learn more details about how functions work in ML. We glossed over details about functions before, but they're so important to ML programming that we really need to understand them to be ML masters.

3.1 Functions as parameters

In ML, functions are treated on the same level as raw data. What this means is that you can work with functions in the same way you can work with other types of data. For example, a function might take a function as a parameter, or it might compute a different function. For example, consider the following function that takes a function and scales it by a constant.

```
- fun scaleFunction func factor =  
  = let  
  =   fun funcScaled x = factor * func x;  
  =   in  
  =     funcScaled  
  =   end;  
val scaleFunction = fn : (int -> int) -> int -> (int -> int)
```

This function takes two parameters: a function `func` from integers to integers, and an integer `factor`. It computes a new function `funcScaled` (a function from integers to integers) and returns this new function.

Now if we have a function of some sort, like $3x^2 + 5$, and we want to scale it by a factor of 2, we can call `scaleFunction` to accomplish this.

```
- fun example x = 3 * x * x + 5;  
val example = fn : int -> int  
- val scaled = scaleFunction example 2;  
val scaled = fn : int -> int  
- scaled 1;  
val it = 16 : int
```

Here we define the value `scaled` to be what `scaleFunction` returns. Since `scaleFunction` returns a function from integers to integers, `scaled` is this function from integers to integers. In this example, in fact, it is twice the `example` function: `scaled` computes $6x^2 + 10$. Now given the number 1, `scaled` returns $6 \cdot 1^2 + 10 = 16$.

3.2 Multiple parameters revisited

Recall when we defined `dist`, a function that takes four parameters.

```
- fun dist x0 y0 x1 y1 = Math.sqrt (sq (x0 - x1) + sq (y0 - y1));
val dist = fn : real -> real -> real -> real -> real
```

We glossed over the response of the interpreter here without really thinking about it. Hidden in it is an interesting tidbit about how ML works. The response says that `dist` is a function that takes a *single* `real` as a parameter, and it returns a `real -> real -> real -> real`. Of course, this is itself a function taking a `real` as a parameter and returning a `real -> real -> real`, which is a function taking a `real` as a parameter and returning a `real -> real`, which is a function taking a `real` as a parameter and returning a `real`.

ML really doesn't have a capacity for storing functions that take more than one parameter at a time, because it doesn't need it. Instead, it remembers a multiple-parameter function as being a function that takes the first parameter and computes another function that takes the second parameter and returns a value.

Usually, this doesn't make any difference — and you can write perfectly reasonable programs without understanding this at all (which is why we glossed over it before). But this fact can lead to some interesting results. For example, the following would be a completely legitimate thing to do in ML.

```
- val dist_00 = dist 0.0 0.0;
val dist_00 = fn : real -> real -> real
```

What we have done here is to create a new identifier, representing a function — namely, the function returned by passing `0.0` into `dist` and then `0.0` into that return value. This gives us a `real -> real -> real`, which we can use subsequently.

```
- dist_00 3.0 4.0;
val it = 5.0 : real
- dist_00 12.0 5.0;
val it = 13.0 : real
```

3.3 Anonymous functions

Since functions are so prevalent in ML, it's useful to be able to define functions without having to go through the bother of naming them. For example, when we used `scaleFunction` before, we had to name our example function (we chose `example`), even though we really didn't want to refer to the function more than once.

ML provides a way for you to describe a function without giving it a name. The following is a briefer version of our function-scaling example employing an anonymous function.

```
- val scaled = scaleFunction (fn x => 3 * x * x + 5) 2;
val scaled = fn : int -> int
- scaled 1;
val it = 16 : int
```

Here we've defined an unnamed, throwaway function that takes a single parameter x and returns $3x^2 + 5$, and we've passed this function into `scaleFunction`. The word `fn` begins an anonymous function definition, followed by the name of the parameter and a right arrow `=>` to separate the parameter from the return value expression. (This arrow is a slight inconsistency from defining functions with `fun`, where an equals sign separates the parameter from the expression, but that's how it is.)

There are two important restrictions on these anonymous functions. The minor restriction is that they can have only one parameter. (This restriction is just minor, because we can easily define a function that takes the first parameter and returns a function taking the second parameter and computing the return value for both, similar to what we saw in Section 3.2.) The major restriction is that, because there is no name with which to refer to it, an anonymous function cannot be recursive.

3.4 Identifier binding time

When it defines a function, the ML interpreter uses the identifiers defined at the time of the function's definition. To understand this distinction, consider the following transcript.

```
- val g = 9.78;
val g = 9.78 : real
- fun distanceFallen t = 0.5 * g * t * t;
val distanceFallen = fn : real -> real
- distanceFallen 1.0;
val it = 4.89 : real
- val g = 1.62;
val g = 1.62 : real
- distanceFallen 1.0;
val it = 4.89 : real
```

We defined `g` to represent the Earth's gravity, and then we defined the `distanceFallen` function to use this `g`. Later we changed `g` for the Moon's gravity, but this did not affect the behavior of the function.

This issue becomes important in writing larger programs. In these larger programs, you break the program into subproblems, each subproblem handled by a function. And then you write functions that combine these functions into the problem at hand. Now if, in the process of writing this overall function, you discover that a subproblem's function is wrong, you will redefine this subproblem's function. But because of the binding-time issue illustrated by the above transcript, you must also redefine the overall function to affect the overall function's behavior.

Consider the following example.

```
- fun sq x : real = x * x * x;
val sq = fn : real -> real
- fun dist x0 y0 x1 y1 = Math.sqrt (sq (x1 - x0) + sq (y1 - y0));
val dist = fn : real -> real -> real -> real -> real
- dist 0.0 0.0 3.0 4.0;
val it = 9.53939201417 : real
```

"Hold on," you think, "that test was supposed to return 5!" On further investigation, you notice that `sq` was misdefined. So you repair it and retest.

```
- fun sq x : real = x * x;
val sq = fn : real -> real
- dist 0.0 0.0 3.0 4.0;
val it = 9.53939201417 : real
```

If you're not familiar with the binding-time issue, this response will throw you for a loop: How could `dist` be unchanged if I changed how `sq` works? Of course, the answer is that `dist` is still using the old value of `sq`; if we want to use the new `sq`, we have to redefine `dist`.

```
- fun dist x0 y0 x1 y1 = Math.sqrt (sq (x1 - x0) + sq (y1 - y0));
val dist = fn : real -> real -> real -> real -> real
- dist 0.0 0.0 3.0 4.0;
val it = 5.0 : real
```

3.5 Polymorphic functions

Sometimes you can write a function that is completely ambiguous about the type of its parameter. For these functions, ML will maintain ambiguity, assuming you wish to keep the function as general as possible. Such functions are called **polymorphic functions**, because they will change shape based on the parameters you give it.

Consider, for example, the following definition of the identity function.

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

The 'a in the interpreter's response is pronounced *alpha*; it is a placeholder for whatever type is appropriate, given the parameters you use when calling the function. The interpreter's reply indicates that `identity` is a function taking an *alpha* as a parameter and returning an *alpha*. We can pass anything at all to this function, and it will reinterpret `identity` to mean what we want.

```
- identity 3;
val it = 3 : int
- identity true;
val it = true : bool
- identity real;
val it = fn : int -> real
```

In the first case, the interpreter ran `identity` as a function from integers to integers. In the second, it ran `identity` as a function from Booleans to Booleans. And the third time, it ran `identity` as a function from functions from integers to reals to functions from integers to reals.

More impressive examples are possible. The following function takes two parameters f and g and computes their *composition* — that is, the function that, given a parameter x , computes $f(g(x))$.

```
- fun compose f g = (fn x => f (g x));
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

The response here indicates that `compose` is a function taking two functions as parameters. The first function is from alphas to betas, and the second function is from gammas to alphas. (Notice that ML has inferred that the return type of g must match the parameter type of f .) The function `compose` returns a function from gammas to betas.

```
- val realRound = compose real round;
val realRound = fn : real -> real
- realRound 3.8;
val it = 4.0 : real
```

In this example, we created a function `realRound`, which is the composition of the built-in function `real` and the built-in function `round`. It takes *alpha* to be `int`, and *beta* and *gamma* to be `real`, and it computes the composition based on this.

In fact, I fibbed a bit when I wrote down how SML would respond given my definition of `scaleFunction` earlier. There was nothing in the function that indicated what type the parameter function took as its parameter. For this reason, `scaleFunction` is actually polymorphic. The interpreter would respond as follows.

```
val scaleFunction = fn : ('a -> int) -> int -> 'a -> int
```

So the function could just as easily scale a function from reals to integers, or even functions that return integers given a function from reals to reals.

3.6 Pattern matching and cases

ML includes a way to define a function's behavior for different cases of parameters using a technique called **pattern matching**. In this technique, we list possible cases, separated by a vertical bar (`|`). Here is a redefinition of the factorial function `fact` using pattern matching.

```
fun fib 1 = 1
  | fib 2 = 1
  | fib n = fib (n - 1) + fib (n - 2);
```

In evaluating a function's value for a given parameter, the interpreter will try each of the cases and see whether it matches the parameter. For example, if the parameter is 1, then this matches the first case listed, and so the return value is 1. If the parameter is 3, neither the first case nor the second case matches, but 3 does match the third case (where we take *n* to be 3); so the return value is the value of the expression `fib (3 - 1) + fib (3 - 2)`.

Using the numeric values we have seen so far, a pattern can be only either a particular value (like 1 or 2), or it can be an identifier name. The identifier will match anything.

If your function doesn't cover all the cases, SML will kindly warn you about the problem, in case it indicates an error in your program. (Sometimes it's fine.)

```
- fun fib 1 = 1
=   | fib 2 = 1
=   | fib 3 = 2;
stdIn:17.1-19.14 Warning: match nonexhaustive
      1 => ...
      2 => ...
      3 => ...

val fib = fn : int -> int
```

The interpreter here still created the `fib` function, but if you call it with a parameter that matches none of the cases, it spits back an error.

```
- fib 5;
uncaught exception nonexhaustive match failure
  raised at: stdIn:19.13
```

Pattern matching for the functions we've seen so far is just a minor convenience — basically, it's sometimes an alternative to using `if` in the return value expression. But in many situations, it is necessary. In the next chapter we'll see some of these cases, and we'll be using pattern matching aggressively as we develop our functions.

Chapter 4

Composite types

We've only worked with **elementary data types** so far — integers, floating-point numbers, and Booleans. These types can only represent one elemental piece of data. But we often want to be able to represent several pieces of data at once. To do this, we need **composite data types**.

4.1 Tuples

A **tuple** in ML is a structure that has several data types put together. A tuple could have an integer and a real number, representing perhaps a student's ID number and grade.

```
- val student = (94827, 50.0 + 33.08);  
val student = (94827, 83.08) : int * real
```

A tuple's value is represented as several values listed in parentheses, with commas separating the values. A tuple's type lists the types of the separate values in order, separated by asterisks (*).

Some built-in functions take a tuple as a parameter. For example, the built-in function `Math.pow` takes a tuple containing two real numbers x and y as a parameter, and returns the result x^y .

```
- Math.pow (2.0, 5.0);  
val it = 32.0 : real
```

Tuples are useful for parameters when we want to group some together. For example, when we wrote our `dist` function, it would be nice if we could actually have two parameters instead of four, with each parameter being a tuple of two numbers representing a point.

```
- fun dist (x0, y0) (x1, y1) = Math.sqrt (sq (x0 - x1) + sq (y0 - y1));  
val dist = fn : real * real -> real * real -> real
```

We're using pattern matching here: The pattern `(x0, y0)` matches a tuple of two values, and we let `x0` represent the first value of the tuple and `y0` represent the second value.

If we want to use the function, we pass it two tuples.

```
- dist (0.0, 0.0) (3.0, 4.0);  
val it = 5.0 : real
```

We can of course define an identifier to be a tuple, and then pass the identifier's value into the function.

```
- val origin = (0.0, 0.0);  
val origin = (0.0, 0.0) : real * real  
- dist origin (3.0, 4.0);  
val it = 5.0 : real
```

Tuples are particularly useful when we want a function to return multiple values. For example, we might want a function that translates a point by a given offset.

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

Returning a tuple is more than a convenience: Before this, writing a function that could return multiple values was an impossibility.

4.2 Lists

Lists in ML give a way of representing several data elements of the same type. The distinction between tuples and lists gives an interesting tradeoff: A tuple type allows you to mix types together, but it restricts you to a fixed combination of types. A list is of variable length, but each item of the list must be of the same type.

You can denote a particular list in ML by enclosing it in brackets, with the different elements of the list separated by commas.

```
- [2, 3, 5, 7, 11];
val it = [2,3,5,7,11] : int list
```

The ML interpreter's response gives the type of this list: It is a list of integers. Notice that the type doesn't indicate anything about the length of the list — it just gives the type of the elements of the list.

There is a special pre-defined constant list called `nil` by the interpreter, which is the empty list, `[]`.

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

As this transcript indicates, `nil` is polymorphic. We'll use `nil` frequently while writing our programs.

4.2.1 The cons operator

Lists have an expression operator called the **cons operator** `::`. Given a data element on the left side, and a list of data elements on the right, the cons operator computes a list starting with the element on left and followed by the list on right.

```
- 2 :: [3,5];
val it = [2,3,5] : int list
```

Notice the asymmetry of this operator — while other operators we have seen have the same type on either side of the operator, the cons operator has a data element on the left side and a list of data elements of the same type on the right side.

Also in contrast to the other operators, the cons operator is **right-associative**. Other operators, like subtraction, are left-associative: $2 - 3 - 5$ is understood as $(2 - 3) - 5$. But the cons operator goes the other way: $2 :: 3 :: [5]$ is understood as $2 :: (3 :: [5])$.

The cons operator gives us a way of building up a list. Consider the following function, designed to build a list of the numbers from n down to 1.

```
fun countDown n = if n = 0 then [] else n :: countDown (n - 1);
```

We have used the inductive method to build this function: Given a list of numbers from $n - 1$ down to 1, we can reason, we can construct a list of numbers from n down to 1 by simply putting n on the front using the cons operator.

4.2.2 Simple list functions

When we work with lists, we'll frequently want to write functions that take a list as a parameter and iterate through the list. We'll write these functions using the inductive method — thinking to ourselves: If we want to know the result for a given list of length n , how could we figure it out given the results for lists of length less than n ?

Summing a list of integers

For example, say we want to compute the sum of everything in a list. If we have a list of four numbers `[2, 3, 5, 7]`, it would be easy to figure out the sum if we knew the sum for lists of three integers, because then we could simply add 2 to the sum of the list of three integers `[3, 5, 7]`. The base case for this reasoning would be a list of zero integers. The sum of no numbers at all is zero.

Our ML implementation of this reasoning employs pattern matching.

```
fun sumList nil = 0
  | sumList (data::rest) = data + sumList rest;
```

Our first case handles the base case: The sum of the numbers in the empty list is 0. The second case is the inductive step: Given a list with `data` at the front and `rest` being the numbers following it, the sum of everything in `data::rest` is `data` plus the sum of the numbers in `rest`.

Here's a derivation demonstrating how this would work with `[2, 3, 5, 7]`.

```
sumList [2,3,5,7] = 2 + sumList [3,5,7]
                  = 2 + (3 + sumList [5,7])
                  = 2 + (3 + (5 + sumList [7]))
                  = 2 + (3 + (5 + (7 + sumList nil)))
                  = 2 + (3 + (5 + (7 + 0)))
                  = 17
```

The names of the identifiers `data` and `rest` aren't special: What's special is the use of the `cons` operator in the pattern, which matches what's on the left-hand side to the first item of the list and what's on the right-hand side to the rest of the list.

This structure defines a template for probably 90% of the list functions we write:

```
fun functionName nil = ... base case...
  | functionName (data::rest) =
    ... inductive step combining data and functionName rest...
```

We'll now see two more examples employing this same structure.

Finding a student's grade

Say we have a class represented by a list of students, each student represented by a tuple of an integer ID number and a real grade. We want to write a function to find the grade for the student with a given ID number. Using the same reasoning as before, we ask ourselves this: Suppose we knew how to determine how to find the grade in a shorter list; would that help us to find the grade in a given list?

We answer: Of course! We can immediately check whether the ID number is at the front of the list; if it's not, then it must be in the rest of it, and so we can use our ability to find the grade in this shorter list. This reasoning is exactly what we can write in our function.

```
fun findGrade id nil = 0.0
  | findGrade id ((data, grade)::rest) =
    if data = id then grade else findGrade id rest;
```

We decided to make a base case here, though technically the problem definition didn't call for it, just so something reasonable happens if the function is given an ID number for a student not in the list.

Notice the use of pattern-matching in the second case here: We match `data` to be the first element of the tuple at the head of the list and `grade` to be the second element of the tuple; and we let `rest` be everything after this first student. The ML interpreter will interpret this appropriately.

```
val findGrade = fn : 'a -> ('a * real) list -> real
```

The interpreter inferred that the second parameter is a list of tuples by looking at the pattern in the second case; it knows the second item in each tuple is a real number because in one situation the function returns the real number 0.0 and in another it returns the second element of the tuple; hence the second element of the tuple must be a real. It infers that the first element of the list is `'a`, indicating that this is a polymorphic function. (The *two* quotes are to indicate that the type must support equality-testing.)

Inserting into an ordered list

In the final example problem, we want to write a function that takes a parameter x and a list L of integers that the function can assume is in increasing order. The function should return a list including both x and all the elements of L , maintaining the same increasing order.

Again, we reason: If we know how to insert x into a list shorter than L , how does that help us insert x into L ?

```
fun insAscending x nil = [x]
  | insAscending x (data::rest) =
      if x <= data then x::data::rest;
      else data::(insAscending x rest)
```

This solution observes that, if x is at most the first item in L , then we simply want to return a list with x first and then L following. Otherwise, we know x follows the first item in the list, and so the result would insert x into the items following the first item, and then add the first item of L at the front of the list.

4.2.3 Complex list functions

Sometimes this template doesn't fit. To see some examples, we'll investigate an implementation of the **merge sort** algorithm — a particular algorithm for sorting a list of items. This is a recursive algorithm that works by first splitting the list into two equal-sized pieces. Then, using merge sort recursively, the algorithm sorts each of these pieces. Then we merge the two sorted lists together. Figure 4.1 has a recursion tree diagramming of how this algorithm might sort a list of numbers.

This algorithm has three parts: splitting the list, sorting each half, and then merging them. Each diverges from the standard list algorithm. We'll look at the first part, then the third, then the second.

In the first part, we want to create a function that takes a list as a parameter and returns a list containing half of the elements of the parameter list. To do this, we'll write `everyOther`, which returns every other element in the list. While the simple list functions go down the list one element at a time, in this function we'll go down the list two elements at a time. Each time, we'll only add the first of the two elements to the returned list.

```
fun everyOther nil = nil
  | everyOther [x] = [x]
  | everyOther (x::y::rest) = x :: (everyOther rest);
```

To write this, we had to include a second base case — if there's just one element, we can't really talk about the first two elements of the list. (This second case is analogous to the second base case required in our recursive implementation of `fib`.)

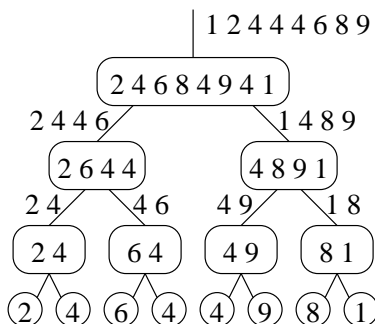


Figure 4.1: An example recursion tree for merge sort. The lines are labeled by return values.

The third part — merging two lists — involves a function taking two lists as parameters. In each recursive call, we look at the front of both of the lists; the lesser will go at the front of the returned list. Then we make a recursive call on both lists, with the lesser removed from its list. In this recursive call, one of the parameters will be identical and one of the parameters will be one element shorter. Our base case is when one of the two lists is empty.

```

fun merge nil list = list
  | merge list nil = list
  | merge (data0::rest0) (data1::rest1) =
    if data0 < data1
    then data0 :: (merge rest0 (data1::rest1))
    else data1 :: (merge (data0::rest0) rest1);

```

And our second (and final) part is to sort a list. We'll break the parameter list into two halves (one by calling `everyOther` on the parameter list and the other by calling `everyOther` on the parameter list with the first element removed), sort each half using a recursive call, and then using `merge` to sort them. Our base case is when we reach an empty list or a one-element list — in both cases, sorting the list is trivial.

```

fun mergeSort nil = nil
  | mergeSort [data] = [data]
  | mergeSort (data::rest) =
    let
      val oddElts = everyOther (data::rest);
      val evenElts = everyOther rest;
      val sortedOddElts = mergeSort oddElts;
      val sortedEvenElts = mergeSort evenElts;
    in
      merge sortedOddElts sortedEvenElts
    end;

```

Writing list functions like these take a bit of inspiration; all three used the inductive method, but none of the cases used the typical list-function template of combining the head of the list with the answer for the list with this head element removed.

4.2.4 Built-in list functions

ML includes a host of built-in list functions to aid you in writing programs that work with lists. These functions represent common operations you might want to perform on lists; it's useful to study these tools so that you don't end up reinventing the wheel (or the hammer, if you prefer not to mix metaphors).

For example, it's frequently useful to find the sum of everything in a list. The `length` function does this. Using this with the `sumList` function we wrote earlier, we can define a function that finds the average of a list of integers.

```
- length [3,4,5,6];
val it = 4 : int
- fun average lst = real (sumList lst) / real (length lst);
val it = fn : int list -> real
```

Or you may want to append two lists together. ML provides `@`, an operator that takes two lists on either side and appends them.

```
- [1,3,5,7] @ [2,4,6,8]
val it = [1,3,5,7,2,4,6,8] : int list
```

The `map` function takes a function and a list and returns a list of the results of applying the function to each element of the list in order. For example, say we want to convert a list of integers into a list of reals; we can just use `map` to apply the function `real` to each element of the list.

```
- map real [2,3,5];
val it = [2.0, 3.0, 5.0] : real list
```

Or suppose we want to sum up the squares of all the numbers in a list. No problem: We call `sumList` on the list returned by mapping a squaring function on each element of the list.

```
- sumList (map (fn x => x * x) [2,3,5,7]);
val it = 87 : int
```

(Notice the use of the anonymous squaring function here.)

There are many other built-in functions — this is just a sample. Whenever you want to do something that sounds like a common thing to want to do, check whether it's already been built before attempting to build your own.

4.3 Strings (and characters)

So far we have been working exclusively with programs for numbers. Often, however, ML functions have to work with text instead. ML provides two data types for dealing with text — the *character* and the *string*.

4.3.1 Constant values

The **character** is the more elementary type: It's just a single character, like a letter or a digit. In program text, it's represented by a sharp sign followed by the letter enclosed in double-quotes.

```
- val initial = #"C";
val it = #"C" : char
```

A **string** is a sequence of several letters. It allows you to represent a word or a sentence. A constant string can be represented in ML by enclosing it in double-quotes.

```
- val sentence = "ML is fun.";
val sentence = "ML is fun." : string
```

You may be wondering, “If a string is enclosed in double-quotes, how can I get a double-quote into a string?” The answer is that you precede the double-quote by the **escape character**, the backslash ‘\’.

```
- val quote = "She said, \"ML is fun.\"";
val quote = "She said, \"ML is fun.\" : string
```

Don't be fooled by the backslashes the interpreter prints in its response — it's just showing an escape character itself so that you the reader don't get confused about whether the double-quote is ending the string or is part of the string. Internally, that backslash isn't there.

“Ok,” you may think, “but what if I want a backslash in my string?” You can place the escape character before a backslash too.

```
- val directory = "C:\\WINDOWS";
val directory = "C:\\WINDOWS" : string
```

You can also use the escape character to denote special characters. The most important of these is the **newline character**, indicating a line break. A backslash followed by a lower-case *n* gives you this.

```
- val poem = "Roses are red.\nViolets are blue.\n";
val poem = "Roses are red.\nViolets are blue.\n" : string
```

4.3.2 Working with strings

Like lists, ML has several built-in functions for working with text that you might need for building functions that manipulate text. Again, we'll just sample a few.

The first is the built-in operator `^` (a caret). It works like the `@` operator for appending lists, but `^` appends strings.

```
- "ML" ^ " is fun."
val it = "ML is fun." : string
```

In some sense, the built-in `explode` and `implode` functions are really all you need to know about strings. They give a way of translating strings to and from lists of characters. The `explode` function takes a string and returns a list of characters in it; the `implode` takes a list of characters and constructs a string including those characters.

```
- explode it;
val it = [#"M",#"L",#" ",#"i",#"s",#" ",#"f",#"u",#"n",#"."] : char list
- implode it;
val it = "ML is fun." : string
```

If you've mastered working with lists, then all you need to learn is these two functions, and you do whatever you want with strings also. For example, if we want to capitalize all the letters of a string, we can use these functions with `map` and the built-in character function `Char.toUpperCase`.

```
- implode (map Char.toUpperCase (explode "ML is fun.));
val it = "ML IS FUN." : string
```

In fact there is a `String.map` function which is the `string` type's answer to the `list` type's `map` function. We could use that instead here.

```
- String.map Char.toUpperCase "ML is fun.";
val it = "ML IS FUN." : string
```

Of course there are many other built-in string functions. When you want to work with strings, check out a reference guide to see how much of what you want can be done with what is already built into ML.