Functional Programming

5

Chapter 3 introduced assembly language which was a very prescriptive language. Certain operands had to be on the operand stack before an instruction could be executed. These details had to be dealt with even though the programmer was trying to solve a bigger problem than how to execute the next instruction. This was solved by learning some patterns of assembly language instructions that could be used to solve bigger problems like implementing a loop. Of course, even writing a loop is more prescriptive than trying to compute the sum of some list of integers.

Chapter 4 moved on to Java and C++ where programming was less prescriptive. Most programmers learn to program imperatively first. Object-oriented languages are imperative languages where objects are created and the states of objects are updated as program execution proceeds. Thinking about maintaining and updating the states of objects is a lot less prescriptive than thinking about which instruction to execute next.

This chapter introduces *functional programming*. Functional languages, like Standard ML, obviously concentrate more heavily on writing and calling functions. However, the term *functional programming* doesn't say what functional programming languages lack. Specifically, pure functional languages lack assignment statements and iteration. Iteration relates to the ability to iterate or repeat code as in a loop of some sort. It is impossible in a pure functional language to declare a variable that gets updated as your program executes! If you think about it, if there are no variables, then there isn't any reason for a looping construct in the language. Iteration and variables go hand in hand. But, how do you get any work done without variables? The primary mode of programming in a functional language is through recursion.

Functional languages also contain a feature that other languages don't. They allow functions to be passed to functions as parameters. We say that these functions are higher-order. Higher-order functions take other functions as parameters and use them. There are many useful higher order functions that are derived from common patterns of computation. Particular instances of these patterns commonly have one small

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_5

```
1  program P;
2    var b : integer;
3    function a() : integer;
4    begin
5      b:=b+2;
6      return 5
7     end;
8  begin
9    b:=10;
10   write(a()+b)
11   (* or write(b+a()) *)
12 end.
```

Fig. 5.1 Commutativity

difference between them. If that small difference is left as a function to be defined later, we have one function that requires another function to complete its implementation. Higher-order functions may be customized by providing some of their functionality later. In some ways this is the functional equivalent of what inheritance or interfaces provide us in object-oriented languages.

These two features, lack of variables and higher-order functions, drastically change the way in which you think about programming. Programming recursively takes some time to get used to, but in the end it is a very nice way to program. Programming recursively is more declarative than prescriptive. Writing imperative programs is prescriptive. When programming declaratively we can focus on *what* we want to say about a problem instead of exactly *how* to solve a problem.

But why would we want to get rid of variables in a programming language? The problem is that variables often make it hard to reason about our programs. Functional languages are more mathematical in nature and have certain rules like commutativity and associativity that they follow. Rules like associativity and commutativity can make it easier to reason about our programs.

Practice 5.1 Is addition commutative in C++, Pascal, Java, or Python? Will *write(a+b)* always produce the same value as *write(b+a)*? Consider the Pascal program in Fig. 5.1. What does this program produce? What would it produce if the statement were *write(b+a())*?
You can check your answer(s) in Section 5.26.1.

5.1 Imperative Versus Functional Programming

You are probably familiar with at least one imperative language. Languages like C, C++, Java, Python, and Ruby are considered imperative languages because the fundamental construct is the assignment statement. In each of these languages we declare variables and assign them values, updating those variables as a program's execution progresses.

Imperative languages are heavily influenced by the von Neumann architecture of computers that includes a store and an program counter; the computation model has control structures that iterate over instructions that make incremental modifications of memory. Assignment of values to variables, for loops, and while loops are all part of imperative languages. The principal operation is the assignment of values to variables. Programs are statement oriented, and they carry out algorithms with statement level sequential control. In other words, computing is done by side-effects.

Sometimes problems with imperative programs stem from these side-effects. It is difficult to reason about a program that relies on side-effects. If we wish to reuse the code of an imperative program then we must be sure that the same conditions are true before the reused code executes since imperative code relies on a certain machine state. As programmers we sometimes forget which preconditions are required and what postconditions result from executing a segment of code. That can lead to bugs in our programs.

Functional languages are based on the mathematical concept of a function and do not reflect the underlying von Neumann architecture. These languages are concerned with data objects and values instead of variables. The principal operation is function application.

Functions are treated as first-class objects that may be stored in data structures, passed as parameters, and returned as function results. Primitive functions are generally supplied with the language implementation. Functional languages allow new functions to be defined by the programmer. Functional program execution consists of the evaluation of an expression, and sequential control is replaced by recursion.

There is no assignment statement. Values are communicated primarily through the use of parameters and return values. Without variables, loop statements don't have a purpose and so they also don't exist in pure functional languages.

Pure functional languages have no side-effects other than possibly reading some input from the user. Scheme is a pure functional language. In general, functional languages avoid or at least isolate code with side-effects. Even input and output operations in functional languages do not update the state of variables within a program.

What is amazing is that it has been proven that exactly the same things can be computed with functional languages as can be computed with imperative languages. This is known because a Turing machine, the theoretical basis for imperative programming and the design of the computer, have been proven equivalent in power to the Lambda Calculus, the basis for all functional programming languages.

You might be surprised by the number and types of languages that support functional programming. Of course, Standard ML was designed as a functional language

from the ground up, but languages like C++, Java, and Python also support functional programming. While C++, Java, and Python are also object-oriented imperative languages, they all support functional programming as well. Functional programming does not depend so much on the language, but how you use the language. The rest of this chapter will introduce the functional style of programming. It all started with the lambda calculus, which is briefly considered next.

5.2 The Lambda Calculus

All functional programming languages are derived either directly or indirectly from the work of Alonzo Church and Stephen Kleene. The lambda calculus was defined by Church and Kleene in the 1930s, before computers existed. At the time, mathematicians were interested in formally expressing computation in some written form other than English or other informal language. The lambda calculus was designed as a way of expressing those things that can be computed. It is a very small, functional programming language. In the lambda calculus, a function is a mapping from the elements of a domain to the elements of a codomain given by a rule. Consider the function $cube(x) = x^3$. What is the value of the identifier *cube* in the definition $cube(x) = x^3$? Can this function be defined without giving it a name?

$\lambda x.x^3$ defines the function that maps each x in the domain to x^3. We can say that this definition or *lambda abstraction*, $\lambda x.x^3$, is the value bound to the identifier *cube*. We say that x^3 is the *body* of the lambda abstraction. Every lambda *abstraction* in lambda notation is a function of one identifier. However, lambda *expressions* may contain more than one identifier.

The expression $y^2 + x$ can be expressed as a lambda abstraction in one of two ways:

$$\lambda x.\lambda y.y^2 + x$$
$$\lambda y.\lambda x.y^2 + x$$

In the first lambda abstraction the x is the first parameter to be supplied to the expression. In the second lambda abstraction the y is the parameter to get a value first. In either case, the abstraction is often abbreviated by throwing out the extra λ. In abbreviated form the two abstractions would become $\lambda xy.y^2 + x$ and $\lambda yx.y^2 + x$.

5.2.1 Normal Form

To say the lambda calculus, or any language, has a normal form means that each expression that can be reduced has a simplest form. It means that we can reduce more complex expressions to simpler expressions in some mechanical way. The lambda calculus exhibits a property called *confluence*.

$$
\begin{aligned}
&\underline{(\lambda xyz.xz(yz))(\lambda x.x)(\lambda xy.x)} \\
&\Rightarrow \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda xy.x)} \\
&\Rightarrow \lambda z.\underline{(\lambda x.x)z((\lambda xy.x)z)} \\
&\Rightarrow \lambda z.\underline{z((\lambda xy.x)z)} \\
&\Rightarrow \lambda z.z(\underline{\lambda y.z})\square
\end{aligned}
$$

Fig. 5.2 Normal Order Reduction

Confluence means that one or more reduction strategies (or intermixing them) always leads to the same normal form of an expression, assuming the expression can be reduced by the reduction strategy. This property of confluence was proven in the Church–Rosser theorem.

Function application (i.e. calling a function) in lambda notation is written with a lambda abstraction followed by the value to call the abstraction with. Such a combination is called a *redex*.

To call $\lambda x.x^3$ with the value 2 for x we would write

$$(\lambda x.x^3)2$$

This combination of lambda abstraction and value is called a *redex*.

A redex is a lambda expression that may be reduced. Typically a lambda expression contains several redexes that may be chosen to be reduced. Function application is left-associative meaning that if more than one redex is available at the same level of parenthetical nesting, the left-most redex must be reduced first. If the left-most outer-most redex is always chosen for reduction first, the order of reduction is called normal order reduction. When a redex is reduced by applying the lambda calculus equivalent of function application it is called a β-reduction (pronounced beta-reduction).

The normal order reduction of $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda xy.x)$ is given in Fig. 5.2. The redex to be β-reduced at each step is underlined.

Practice 5.2 Another reduction strategy is called applicative order reduction. Using this strategy, the left-most inner-most redex is always reduced first. Use this strategy to reduce the expression in Fig. 5.2. Be sure to parenthesize your expression first so you are sure that you left-associate redexes.
You can check your answer(s) in Section 5.26.2.

In practice problem 5.2 you should have reduced the lambda expression to the same reduced lambda expression derived from the normal order reduction in Fig. 5.2. If you didn't, you did something wrong. If you want more experience with reducing lambda expressions you may wish to consult a lambda expression interpreter. One excellent interpreter was written by Peter Sestoft and is available on the web. It

is located at http://www.itu.dk/people/sestoft/lamreduce/. Be sure to read his help page to get familiar with the syntax required for entering lambda expressions in his interpreter. Also be aware that his interpreter does not understand math symbols like $+$. Instead, you can use a p to represent addition if needed. Sestoft's lambda calculus interpreter is for the pure lambda calculus without knowledge of Mathematics or any other language.

5.2.2 Problems with Applicative Order Reduction

Sometimes, applicative order reduction can lead to problems. For instance, consider the expression $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$.

Practice 5.3 Reduce the expression $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$ with both normal order and applicative order reduction. Don't spend too much time on this! *You can check your answer(s) in Section* 5.26.3.

This practice problem shows why the definition of confluence includes the phrase, *assuming the expression can be reduced by the reduction strategy*. Applicative order may not always result in the expression being reduced. No fear, if that happens we are free to use normal order reduction for a while since intermixing reduction strategies will not affect whether we arrive at the normal form for the expression or not.

5.3 Getting Started with Standard ML

Standard ML (or just SML) is a functional language based on Lisp which in turn is based on the lambda calculus. Important ML features are listed below.

- SML is higher-order supporting functions as first-class values.
- It is strongly typed like Pascal, but more powerful since it supports polymorphic type checking. With this strong type checking it is pretty infrequent that you need to debug your code!! What a great thing!!!
- Exception handling is built into Standard ML. It provides a safe environment for code development and execution. This means there are no traditional pointers in ML. Pointers are handled like references in Java.
- Since there are no traditional pointers, garbage collection is implemented in the ML system.
- Pattern-matching is provided for conveniently writing recursive functions.
- There are built-in advanced data structures like lists and recursive data structures.
- A library of commonly used functions and data structures is available called the *Basis Library*.

There are several implementations of Standard ML. Standard ML of New Jersey and Moscow ML are the most complete and certainly the most popular. There is also a SML.NET implementation that targets the Microsoft .NET run-time library and can be integrated with other .NET languages. There is an MLj implementation that targets the Java Virtual Machine. Poly/ML is another implementation that includes support for Windows programming. While many implementations exist, they all support the same definition of SML. If you write a Standard ML program that runs in one environment, it'll run on any other implementation as long as you are not using platform specific functions.

SML has been successfully used on a variety of large programming projects. It was used to implement the entire TCP protocol on the FOX Project at Carnegie Mellon. It has been used to implement server side scripting on web servers. It was originally designed as a language to write theorem provers and has been used extensively in this area. It has been used in hardware design and verification. It has also been used in programming languages research.

The rest of this chapter introduces SML. By the end of the chapter you should understand and be able to use many of the important features of the language. This text is based on the Standard ML of New Jersey implementation. You can download SML of New Jersey from smlnj.org. SML of New Jersey is available for most platforms so you should be able to find an implementation for your needs.

Once you've installed SML you can open a terminal window and start the interpreter. Typing *sml* at the command-line will start the interactive mode of the interpreter. Typing *ctl-d* will terminate the interpreter. You can type expressions and programs directly in at the interpreter's prompt or you can type them in a file and use that file within SML. To do this you type the word *use* as follows:

```
Standard ML of New Jersey v110.59
- use "myfile.txt";
```

SML will take whatever you have typed in the file and evaluate it just as if you had typed it directly into the interpreter.

The examples and practice problems in this chapter introduce SML. The following sections introduce important aspects of SML and ready the reader to write more complicated programs in the next chapter.

5.4 Expressions, Types, Structures, and Functions

Functional programming focuses on the evaluation of expressions. In SML you can evaluate expressions right in the intepreter. When evaluating an expression you will notice that type information is displayed along with the result of the expression evaluation. The dialog below contains some interactive expression evaluations in the SML interpreter.

In SML the identifier *it* is bound to the result of the last successfully evaluated expression. This is convenient if you want to use the result in a subsequent expression.

```
- 6;
val it = 6 : int
- 5*3;
val it = 15 : int
- ~1;
val it = ~1 : int
- 5.0 * 3.0;
val it = 15.0 : real
- true;
val it = true : bool
- 5 * 3.0;
Error: operator and operand don't agree
  operator domain: int * int
  operand:         int * real
  in expression:
5 * 3.0
-
```

Fig. 5.3 Interpreter Interaction

The last expression result can be referred to as *it* in the subsequent, interactively entered expression.

The interaction presented in Fig. 5.3 contains a negative one written as ~ 1 in SML. While a little unconventional, ~ is the unary negation operator in SML, distinguishing it from the binary subtraction operator.

SML has a very rigorous type system. In fact, the type system for SML has been proved sound. That means that any correctly typed program is guaranteed to be free of type errors. SML is statically typed like C++ and Java. That means that all type errors are detected at compile-time and not at run-time. Robin Milner proved this for Standard ML. ML is the only widely distributed language whose type system has been formally defined and proven type correct.

While being formally defined and rigorous, the type system of ML is remarkably flexible. It is polymorphic. We'll see what this means for us soon. Many of the types in ML are also implicitly expressed. In C++ and Java the type of every variable and function must be declared. You may notice in Fig. 5.3 that the programmer never entered any types for the expressions given there. In most cases Standard ML's type system frees the programmer from having to specify types in a program since they are mostly determined automatically.

You may have also noticed that there is a type error in Fig. 5.3. ML is polymorphic but it is also strongly typed. Since 5 is an integer in SML and 3.0 is a real, the two cannot be multiplied together. If you should have the need to multiply an integer and a real it can be done, but you must explicitly convert one of the types. The intepreter

interaction below show some code to multiply an integer and a real, producing a real number.

```
- Real.fromInt(5) * 3.0;
val it = 15.0 : real
-
```

The integer 5 is converted to 5.0 by calling a function called fromInt in the structure called Real. A Structure in SML is a grouping of functions and types. A structure is like a module in Python or an include in C++. There are several structures that make up the *Basis Library* for Standard ML. The basis library is available in SML when the interpreter is started. The structures in the basis library include Bool, Int, Real, Char, String, and List. *Appendix B* or the website http://standardml.org/Basis contain descriptions of many of these structures.

A function in SML takes one or more arguments and returns a value. The signature of a function is the type of the function. In other words, a function's type is its signature. The signature of the function *fromInt* in the Real structure is

```
val fromInt : int -> real
```

This signature indicates that *fromInt* takes an *int* as an argument and returns a *real*. From the name of the function, and the fact that it is part of the Real structure, we can ascertain that it creates a *real* number from an *int*.

The type on the left side of the arrow (i.e. the ->) is the type of the arguments given to the function. The type on the right side of the arrow is the type of the value returned by the function. The *fromInt* function takes an *int* as an argument and returns a *real*.

Practice 5.4 Write expressions that compute the values described below. Consult the basis library in *Appendix B* as needed.

1. Divide the integer bound to x by 6.
2. Multiply the integer x and the real number y giving the closest integer as the result.
3. Divide the real number 6.3 into the real number bound to x.
4. Compute the remainder of dividing integer x by integer y.

You can check your answer(s) in Section 5.26.4.

5.5 Recursive Functions

Recursion is the way to get things done in a functional language. Recursion happens when a function calls itself. Because of the principle of referential transparency a function must never call itself with the same arguments. If it were to do that, then

the function would do exactly what it did the last time, call itself with the same arguments, which would then.... Well, you get the picture!

To spare ourselves from this problem we insist on two things happening. First, every recursive function must have a base case. A base case is a simple subproblem that we are trying to solve that doesn't require recursion. We must write some code that checks for the simple problem and simply returns the answer in that case.

The second rule of recursive functions requires them to call themselves on some simpler or smaller subproblem. In some way each recursive call should take a step toward the base case of the problem. If each recursive call advances toward the base case then by the mathematical principle of induction we can conclude the function will work for all values on which the function is defined! The trick is not to think about this too hard. The recursive case is often referred to as the inductive case.

Writing functional programs is much more declarative than the prescriptive programming of assembly and imperative programming in languages like C++, Python, and Java. What this statement is really saying is that when writing recursive functions we think much less about how it works and more about the structure of the data. This leads to a few simple steps that can be applied to writing any recursive function. Memorize these steps and practice them and you can write any recursive function.

1. Decide what the function is named, what arguments are passed to it, and what the function should return.
2. At least one of the arguments must get smaller each time. Most of the time it is only one argument getting smaller. Decide which one that will be.
3. Write the function declaration, declaring the name, arguments types, and return type if necessary.
4. Write a base case for the argument that you decided will get smaller. Pick the smallest, simplest value that could be passed to the function and just return the result for that base case.
5. The next step is the crucial step. You don't write the next statement from left to right. You write from the inside out at this point.
6. Make a recursive call to the function with a smaller value. For instance, if it is a list you decided will get smaller, call the function with the tail of the list. If an integer is the argument getting smaller, call the function with the integer argument minus 1. Call the function with the required arguments and in particular with a smaller value for the argument you decided would get smaller at each step.
7. Now, here's a leap of faith. That call you made in the last step worked! It returned the result that you expected for the arguments it was given. Use that result in building the result for the original arguments passed to the function. At this step it may be helpful to try a concrete example. Assume the recursive call worked on the concrete example. What do you have to do with that result to get the result you wanted for the initial call? Write code that uses the result in building the final result for your concrete example. By considering a concrete example it will help you see what computation is required to get your final result.
8. That's it! Your function is complete and it will work if you stuck to these guidelines.

```
fun babsqrt(x,guess) =
  if Real.abs(x-guess*guess) <
     x/1000000.0 then
    guess
  else
    babsqrt(x,(guess + x/guess)/2.0);
```

Fig. 5.4 Square Root

To define a function in SML we write the keyword *fun* followed by a function name, parameters, an equal sign, and the body of the function. The syntax is quite similar to defining functions in other languages. The main difference is the body of the function. Instead of being a sequence of statements with variable assignment, the body of the function will be an expression.

One important expression in SML is the *if-then-else* expression. This is not an *if-then-else* statement. Instead, it's an *if-then-else* expression. An *if-then-else* expression gives one of two values and those values must be type compatible. The easiest way to understand *if-then-else* expressions is to see one in practice.

The Babylonian method of computing square root of a number, x, is to start with an arbitrary number as a *guess*. If $guess^2 = x$ we are done. If not, then let the next guess be $(guess + x/guess)/2.0$. To write this as a recursive function we must find a base case and be certain that our successive guesses will approach the base case. Since the Babylonian method of finding a square root is a well-known algorithm, we can be assured it will converge on the square root. The base case has to be written so that when we get close enough, we will be done. Let's let the *close enough* factor be one millionth of the original number.

The SML code in Fig. 5.4 implements this function. Looking at the code there are two things to observe. The base case comes first. If the guess is within one millionth of the right value then the function returns the guess as the square root. The other observation is the recursive call brings us closer to the solution.

Practice 5.5 $n!$ is called the factorial of n. It is defined recursively as $0! = 1$ and $n! = n * (n-1)!$. Write this as a recursive function in SML.
You can check your answer(s) in Section 5.26.5.

Practice 5.6 The Fibonacci sequence is a sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13, ... Subsequent numbers in the sequence are derived by adding the previous two numbers in the sequence together. This leads to a recursive definition of the Fibonacci sequence. What is the recursive definition of Fibonacci's sequence? HINT: The first number in the sequence can be thought of as the zeroeth element, then the first element is next and so on. So, *fib(0) = 0*. After arriving at the definition, write a recursive SML function to find the nth element of the sequence.
You can check your answer(s) in Section 5.26.6.

5.6 Characters, Strings, and Lists

SML has separate types for characters and strings. A character literal begins with a pound sign (i.e. #). The character is then surrounded by double quotes. So, the first character in the alphabet is represented as #"a" in SML. There are several functions available in the Char structure for testing and converting characters. The signature of the functions in the Char structure is given in *Appendix B*.

Strings in SML are not simply sequences of characters as they are in some languages. A string in SML is its own primitive type. There are functions for converting between strings and characters of course. You can consult *Appendix B* for a list of those functions. A string literal is text surrounded by double quotes. The backslash character (i.e. \) is an escape character in strings. This means to include a double quote in a string you can write \" as part of the string. A `\n` is the newline character in a string and `\t` is the tab character as they are in many languages.

Perhaps the most powerful data structure in SML is the list. A list is polymorphic meaning that there are many list types in SML. However, the list functions all work on any type of list. Since it is impossible to determine all the types in SML (because programmers may define their own types), a list's type is parameterized by a type variable. A list's type is written as *'a list*. When the type of the list is known, the type variable *'a* is replaced by the type it represents. So, a list of integers has type *int list*. You may have figured this out already, but lists in SML must be homogeneous. This means all the elements of a list must have the same type. This is not like some languages, but there is a good reason for this restriction. Requiring lists to be homogeneous makes static checking of the types in SML possible and the type checker sound and complete.

A list is constructed in one of several ways. First, an empty list is represented as *nil* or by the empty list (i.e. *[]*). A list may be represented as a literal by putting a left bracket and a right bracket around the list contents, as in *[1,4,9,16]*. A list may also be constructed using the list constructor which is written *::*, and pronounced *cons*. In the functional language Lisp the same list construction operator is written *cons*

```
:: : 'a * 'a list -> 'a list
@ : 'a list * 'a list -> 'a list
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

Fig. 5.5 Function Signatures

so it is called the *cons* operator by many functional programmers. The *cons* operator takes an element on the left side of it and a list on the right side and constructs a new list of its two arguments. A list may be constructed by concatenating two lists together. List concatenation is represented with the @ symbol. The following are all valid list constructions in SML.

- [1,4,9,16]
- 1 ::[4,9,16,25]
- #"a" ::#"b" ::[#"c"]
- 1 ::2 ::3 ::nil
- ["hello","how"]@["are","you"]

The third example works because the *::* constructor is right-associative. So the rightmost constructor is applied first, then the one to its left, and so on. The signatures of the list constructor and some list functions are given in Fig. 5.5.

Practice 5.7 The following are NOT valid list constructions in SML. Why not? Can you fix them?

- #"a" ::["beautiful day"]
- "hi" ::"there"
- ["how","are"] ::"you"
- [1,2.0,3.5,4.2]
- 2@[3,4]
- [] ::3

You can check your answer(s) in Section 5.26.7.

You can select elements from a list using the hd and tl functions. The *hd* (pronounced *head*) of a list is the first element of the list. The *tl* is the tail or all the rest of the elements of the list. Calling the *hd* or *tl* functions on the empty list will result in an error. Using these two functions and recursion it is possible to access each element of a list. The code in Fig. 5.6 illustrates a function called implode that takes a list

```
fun implode(lst) =
  if lst = [] then ""
  else str(hd(lst))^implode(tl(lst))
```

Fig. 5.6 The Implode Function

```
fun length(x) =
    if null x then 0
    else 1+length(tl(x))
fun append(L1, L2) =
    if null L1 then L2
    else hd(L1)::append(tl(L1),L2)
```

Fig. 5.7 Two List Functions

of characters as an argument and returns a string comprised of those characters. So, *implode([#"H",#"e",#"l",#"l",#"o"])* would yield *"Hello"*.

When writing a recursive function the trick is to not think too hard about how it works. Think of the base case or cases and the recursive cases separately. So, in Fig. 5.6 the base case is when the list is empty (since a list is the parameter). When the list is empty, the string the function should return should also be empty.

The recursive case is when when the list is not empty. In that case, there is at least one element in the list. If that is true then we can call *hd* to get the first element and *tl* to get the rest of the list. The head of the list is a character and must be converted to a string. The rest of the list is converted to a string by calling some function that will convert a list to a string. This function is called *implode*! We can just assume it will work. That is the nature of recursion. The trick, if there is one, is to trust that recursion will work. Later, we will explore exactly why we can trust recursion.

Practice 5.8 Write a function called *explode* that will take a string as an argument and return a list of characters in the string. So, *explode("hi")* would yield *[#"h",#"i"]*. HINT: How do you get the first character of a string?
You can check your answer(s) in Section 5.26.8.

The code in Fig. 5.7 contains a couple more examples of list functions. The length function counts the number of elements in a list. It must be a list because the *tl* function is used. The append function appends two lists by taking each element from the first list and consing it onto the result of appending the rest of the first list to the second list.

Practice 5.9 Use the append function to write reverse. The reverse function reverses the elements of a list. Its signature is

```
reverse = fn: 'a list -> 'a list
```

You can check your answer(s) in Section 5.26.9.

5.7 Pattern Matching

Frequently, recursive functions rely on several recursive and several base cases. SML includes a nice facility for handling these different cases in a recursive definition by allowing pattern matching of the arguments to a function. Pattern matching works with literal values like 0, the empty string, and the empty list. Generally, you can use pattern matching if you would normally use equality to compare values. Real numbers are not equality types. The *real* type only approximates real numbers. The code in Fig. 5.4 shows how two real numbers are compared for equality.

You can also use constructors in patterns. So the list constructor *::* works in patterns as well. Functions like the append function (i.e. the infix @) and string concatenation (i.e. ^) don't work in patterns. These functions are not constructors of values and cannot be efficiently or deterministically matched to patterns of arguments.

Append can be written using pattern-matching as shown in Fig. 5.8. The extra parens around the recursive call to append are needed because the *::* constructor has higher precedence than function application.

Practice 5.10 Rewrite reverse using pattern-matching.
You can check your answer(s) in Section 5.26.10.

```
fun append(nil,L2) = L2
  | append(h::t,L2) = h::(append(t,L2))
```

Fig. 5.8 Pattern Matching

5.8 Tuples

A tuple type is a cross product of types. A two-tuple is a cross product of two types, a three-tuple is a cross product of three types, and so on. (5,6) is a two-tuple of *int * int*. The three tuple *(5,6,"hi")* is of type *int * int * string*.

You might have noticed the signature of some of the functions in this chapter. For instance, consider the signature of the append function. Its signature is

```
val append : 'a list * 'a list -> 'a list
```

This indicates it's a function that takes as its argument an *'a list * 'a list* tuple. In fact, every function takes a single argument and returns a single value. The sole argument might be a tuple of one or more values, but every function takes a single argument as a parameter. The return value of a function may also be a tuple.

In many other languages we think of writing function application as the function followed by a left paren, followed by comma separated arguments, followed by a right paren. In Standard ML (and most functional languages) function application is written as a function name followed by the value to which the function is applied. This is just like function application in the lambda calculus. So, we can think of calling a function with zero or more values, but in reality every function in ML is passed on argument, which may be a tuple. In Standard ML rather than writing

```
append([1,2],[3])
```

it is more appropriate to write

```
append ([1,2],[3])
```

because function application is a function name followed by the value to which it will be applied. In this case append is applied to a tuple of *'a list * 'a list*.

5.9 Let Expressions and Scope

Let expressions are simply syntax for binding a value to an identifier to later be used in an expression. They are useful when you want to document your code by assigning a meaningful name to a value. They can also be useful when you need the same value more than once in a function definition. Rather than calling a function twice to get the same value, you can call it once and bind the value to an identifier. Then the identifier can be used as many times as the value is needed. This is more efficient than calling a function multiple times with the same arguments.

Consider a function that computes the sum of the first *n* integers as shown in Fig. 5.9. Let expressions define identifiers that are local to functions. The identifier called *sum* in Fig. 5.9 is not visible outside the *sumupto* function definition. We say the scope of *sum* is the body of the let expression (i.e. the expression given between the *in* and *end* keywords). Let expressions allow us to declare identifiers with limited scope.

```
fun sumupto(0) = 0
  | sumupto(n) =
    let val sum = sumupto(n-1)
    in
      n + sum
    end
```

Fig. 5.9 Let Expression

Limiting scope is an important aspect of any language. Function definitions also limit scope in SML and most languages. The formal parameters of a function definition are not visible beyond the body of the function.

Binding values to identifiers should not be confused with variable assignment. A binding of a value to an identifier is a one time operation. The identifier's value cannot be updated like a variable. A practice problem will help to illustrate this.

Practice 5.11 What is the value of x at the various numbered points within the following expression? Be careful, it's not what you think it might be if you are relying on your imperative understanding of code.

```
let val x = 10 in
    (* 1. Value of x here? *)
    let val x = x+1
    in
        (* 2. Value of x here? *)
        x
    end;
    (* 3. Value of x here? *)
    x
end
```

You can check your answer(s) in Section 5.26.11.

Bindings are not the same as variables. Bindings are made once and only once and cannot be updated. Variables are meant to be updated as code progresses. Bindings are an association between a value and an identifier that is not updated.

SML and many modern languages use static or lexical scope rules. This means you can determine the scope of a variable by looking at the structure of the program without considering its execution. The word lexical refers to the written word and lexical or static scope refers to determining scope by looking at how the code is written and not the execution of the code. Originally, LISP used dynamic scope rules. To determine dynamic scope you must look at the bindings that were active when the code being executed was called. The difference between dynamic and

```
1  let fun a() =
2          let val x = 1
3              fun b() = x
4          in
5            b
6          end
7      val x = 2
8      val c = a()
9  in
10   c()
11 end
```

Fig. 5.10 Scope

static scope can be seen when functions may be nested in a language and may also be passed as parameters or returned as function results.

The difference between dynamic and static scope can be observed in the program in Fig. 5.10. In this program the function *a*, when called, declares a local binding of *x* to 1 and returns the function *b*. When *c*, the result of calling *a*, is called it returns a 1, the value of *x* in the environment where *b* was defined, not a 2. This result is what most people expect to happen. It is static or lexical scope. The correct value of *x* does not depend on the value of *x* when it was called, but the value where the function *b* was written.

While static scope is used by many programming languages including Standard ML, Python, Lisp, and Scheme, it is not used by all languages. The Emacs version of Lisp uses dynamic scope and if the equivalent Lisp program for the code in Fig. 5.10 is evaluated in Emacs Lisp it will return a value of 2.

It is actually harder to implement static scope than dynamic scope. In dynamically scoped languages when a function is returned as a value the return value can include a pointer to the code of the function. When the function *b* from Fig. 5.10 is executed in a dynamically scoped language, it simply looks in the current environment for the value of *x*. To implement static scope, more than a pointer to the code is needed. A pointer to the current environment is needed which contains the binding of *x* to the value at the time the function was defined. This is needed so when the function *b* is evaluated, the right *x* binding can be found. The combination of a pointer to a function's code and its environment is called a *closure*. Closures are used to represent function values in statically scoped languages where functions may be returned as results and nested functions may be defined. Chapters 3 and 4 introduced closures and Fig. 5.10 provides an example in Standard ML showing why they are necesssary for statically scoped languages.

5.10 Datatypes

The word datatype is often loosely used in computer science. In ML, a datatype is a special kind of type. A datatype is a tagged structure that can be recursively defined. This type is powerful in that you can define enumerated types with it and you can define recursive data structures like lists and trees.

Datatypes are user-defined types and are generally recursively defined. There are infinitely many datatypes in Standard ML. Defining a datatype is like creating a class in C++ without any methods and only public data. In C/C++ we can create an enumerated type by writing the declaration found in Fig. 5.11. This defines a type called TokenType of eleven values: *identifier* is 0, *keyword* is 1, *number* is 2, etc. You can declare a variable of this type as follows.

```
TokenType t = keyword;
```

However, until C++11 there was nothing preventing you from executing the statement

```
t = 1; //this is the keyword value.
```

In this example, even though *t* is of type *TokenType*, it could be assigned an integer with compilers prior to C++11. This is because the TokenType type was just another name for the integer type in C++ prior the C++11. Assigning *t* to 1 didn't bother C++ in the least. In fact, assigning *t* to 99 wouldn't bother C++ either prior to C++11. In Standard ML, and now in C++, we can't use integers and datatypes (or enums) interchangeably.

```
- datatype TokenType = Identifier | Keyword | Number |
    Add | Sub | Times | Divide | LParen | RParen | EOF |
    Unrecognized;
datatype TokenType = Identifier | Keyword | Number | ...
- val x = Keyword;
x = Keyword : TokenType
```

Datatypes allow programmers to define their own types. Normally, a datatype includes other information. Datatypes are used to represent structured data of some sort. By adding the keyword *of*, a datatype value can include a tuple of other types as

```
1 enum TokenType {
2     identifier,keyword,
3     number,add,sub,times,
4     divide,lparen,
5     rparen,eof,unrecognized
6 };
```

Fig. 5.11 C++ Enum Type

```
datatype
  AST = add' of AST * AST
      | sub' of AST * AST
      | prod' of AST * AST
      | div' of AST * AST
      | negate' of AST
      | integer' of int
      | store' of AST
      | recall';
```

Fig. 5.12 An AST Datatype

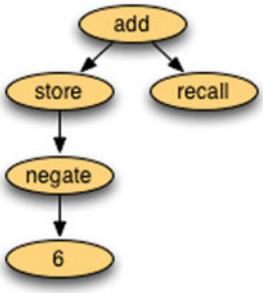

Fig. 5.13 An AST in SML

part of its definition. A datatype can represent any kind of recursive data structure. That includes lists, trees, and other structures that are related to lists and trees. In Fig. 5.12 we have a tree definition with a mix of unary and binary nodes.

Datatypes allow a programmer to write a recursive function that can traverse the data given to it. Functions can use pattern matching to handle each case in a datatype with a pattern match in the function.

In the datatype given in Fig. 5.12 the *add'* value can be thought of as a node in an *AST* that has two children, each of which are *ASTs*. The datatype is recursive because it is defined in terms of itself. The code in Fig. 5.12 is the entire definition of abstract syntax trees for expressions in a calculator language. Store nodes in the tree store their value in the one memory location of the calculator. Recall nodes recall the memory location of the calculator. The *negate'* node represents unary negation of the value we get when evaluating its child. So ~6 is a valid expression if we let the tilde sign represent unary negation as it does in Standard ML.

The abstract syntax tree for ~6S+R is drawn graphically in Fig. 5.13. The value add'(store'(negate'(integer'(6))), recall') is the SML way of representing the AST shown in Fig. 5.13. A function can be written to evaluate such an abstract syntax tree based on the patterns in a value like this and this is done later in the chapter.

```
1  fun evaluate(add'(e1,e2),min) =
2      let val (r1,mout1)= evaluate(e1,min)
3          val (r2,mout) = evaluate(e2,mout1)
4      in
5        (r1+r2,mout)
6      end
7
8    | evaluate(sub'(e1,e2),min) =
9      let val (r1,mout1)= evaluate(e1,min)
10         val (r2,mout) = evaluate(e2,mout1)
11     in
12       (r1-r2,mout)
13     end
```

Fig. 5.14 Pattern Matching Function Results

You can use pattern matching on datatypes. For instance, to evaluate an expression tree you can write a recursive function using pattern-matching. Each pattern that is matched in such a function corresponds to processing one node in the tree. Each subtree can be processed by a recursive call to the same function. In Fig. 5.14, the parameter *min* is the value of the memory before evaluating the given node in the abstract syntax tree. The value *mout* is the value of memory after evaluating the node in the abstract syntax tree.

This example code in Fig. 5.14 illustrates how to use pattern-matching with datatypes and patterns in a *let* construct. This is one way to write the evaluate function to evaluate the abstract syntax trees defined in Fig. 5.12. *mout1* is the value of memory after evaluating *e1*. This is passed to evaluating *e2* as the value of the memory before evaluating *e2*. The value of memory after evaluating *e2* is the value of memory after evaluating the sum/difference of the two expressions. This pattern of passing the memory through the evaluation of the tree is called *single-threading* the memory in the computation.

Practice 5.12 Define a datatype for integer lists. A list is constructed of a head and a tail. Sometimes this constructor is called *cons*. The empty list is also a list and is usually called *nil*. However, in this practice problem, to distinguish from the built-in *nil* you could call it *nil'*.
You can check your answer(s) in Section 5.26.12.

Practice 5.13 Write a function called *maxIntList* that returns the maximum integer found in one of the lists you just defined in practice problem 5.12. You can consult *Appendix B* for help with finding the max of two integers.
You can check your answer(s) in Section 5.26.13.

5.11 Parameter Passing in Standard ML

The types of data in Standard ML include integers, reals, characters, strings, tuples, lists, and the user-defined datatypes presented in the last section. If you look at these types in this chapter and in *Appendix B* you may notice that there are no functions that modify the existing data. The substring function defined on strings returns a new string. In fact most functions on the types of data available in Standard ML return a new value without mutating the arguments passed to them. Not all data in Standard ML is immutable, but most of it is.

There is one type of data that is mutable in Standard ML. A reference is a reference to a value of a determined type. References may be mutated to enable the programmer to program using the imperative style of programming. References are discussed in more detail later in this chapter. The array type in Standard ML is a list of references so by arrays are generally considered mutable data types as well, but only because arrays are lists of references.

The absence of mutable data, except for references, has some impact on the implementation of the language. Values are passed by reference in Standard ML. However, the only time that matters is when a reference is passed as a parameter or one of the few mutable types of objects is passed to a function. Otherwise, the immutability of all data means that how data is passed to a function is irrelevant. This is nice for programmers as they don't have to be concerned about which functions mutate data and which construct new data values. For most practical purposes, there is only one operation that mutates data, the assignment operator (i.e. *:=*) and the only data it can mutate is a reference. In addition, because most data is immutable and passed by reference, parameters are passed efficiently in ML.

5.12 Efficiency of Recursion

Once you get used to it, writing recursive functions isn't too hard. In fact, it can be easier than writing iterative solutions. But, just because you find a recursive solution to a problem, doesn't mean it's an effficient solution to a problem. Consider the Fibonacci numbers. The recursive definition leads to a very straightforward recursive solution. However, as it turns out, the simple recursive solution is anything but efficient. In fact, given the definition in Fig. 5.15, fib(42) took six seconds to compute

```
1  fun fib(0) = 0
2    | fib(1) = 1
3    | fib(n) = fib(n-1) + fib(n-2)
```

Fig. 5.15 The Fib Function

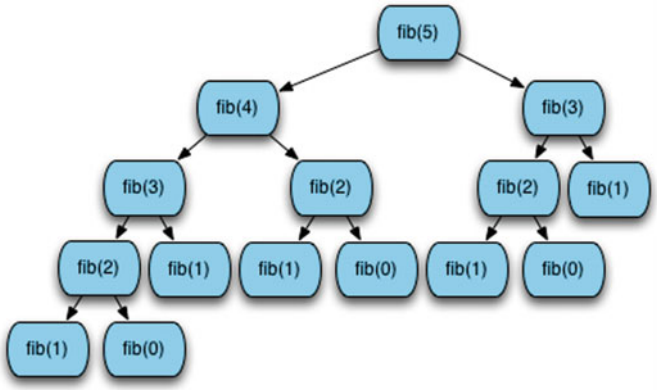

Fig. 5.16 Calls to Calculate fib(5)

on a 2.66 GHz MacBook Pro with 8 GB of RAM. Fib(43) took a third longer, jumping to nine seconds.

The Fibonacci numbers can be computed with the function definition given in Fig. 5.15. This is a very inefficient way of calculating the Fibonacci numbers. The number of calls to fib increases exponentially with the size of n. This can be seen by looking at a tree of the calls to fib as in Fig. 5.16. The number of calls required to calculate *fib(5)* is 15. If we were to enumerate the calls required to calculate *fib(6)* it would be everything in the *fib(5)* call tree plus the number of nodes in the *fib(4)* call tree, *15+9=25*. The number of calls grows exponentially.

Practice 5.14 One way of proving that the *fib* function in Fig. 5.15 is exponential is to show that the number of calls for *fib(n)* is bounded by two exponential functions. In other words, there is an exponential function of n that will always return less than the number of calls required to compute *fib(n)* and there is another exponential function that always returns greater than the number of required calls to compute *fib(n)* for some choice of starting n and all values greater than it. If the number of calls to compute *fib(n)* lies in between then the *fib* function must have exponential complexity. Find two exponential functions of the form c^m that bound the number of calls required to compute *fib(n)*.
You can check your answer(s) in Section 5.26.14.

```
fun fib(n) =
  let fun fibhelper(count,current,previous) =
      if count = n then previous
      else fibhelper(count+1,previous+current,current)
  in
      fibhelper(0,1,0)
  end
```

Fig. 5.17 An Efficient Fib Function

From this analysis you have probably noticed that there is a lot of the same work being done over and over again. It may be possible to eliminate a lot of this work if we are smarter about the way we write the Fibonacci function. In fact it is. The key to this efficient version of *fib* is to recognize that we can get the next value in the sequence by adding together the previous two values. If we just carry along two values, the current and the next value in the sequence, we can compute each Fibonacci number with just one call. The code in Fig. 5.17 demonstrates how to do this. With the new function, computation of fib(43) is instantaneous.

Using a helper function may lead to a better implementation in some situations. In the case of the *fib* function, the *fibhelper* function turns an exponentially complex function into a linear time function. The code in Fig. 5.17 uses a helper function that is private to the *fib* function because we don't want other programmers to call the *fibhelper* function directly. It is meant to be used by the *fib* function. We also wouldn't want to have to remember how to call the *fibhelper* function each time we called it. By hiding it in the *fib* function we can expose the same interface we had with the original implementation, but implement a much more efficient function.

The helper function uses a pattern called an accumulator pattern. The helper function makes use of an accumulator to reduce the amount of work that is done. The work is reduced because the function keeps track of the last two values computed by the helper function to aid in computing the next number.

Practice 5.15 Consider the reverse function from practice problem 5.10. The *append* function is called *n* times, where *n* is the length of the list. How many cons operations happen each time append is called? What is the overall complexity of the reverse function?
You can check your answer(s) in Section 5.26.15.

5.13 Tail Recursion

One criticism of functional programming centers on the heavy use of recursion that is seen by some critics as overly inefficient. The problem is related to the use of caches in modern processors. Depending on the block size of an instruction cache, the code surrounding the currently executing code may be readily available in the cache. However, when the instruction stream is interrupted by a call to a function, even the same function, the cache may not contain the correct instructions. Retrieving instructions from memory is much slower than finding them in the cache. However, cache sizes continue to increase and even imperative languages like C++ and Java encourage many calls to small functions or methods given their object-oriented nature. So, the argument in favor of fewer function calls has certainly diminished in recent years.

It is still the case that a function call takes longer than executing a simple loop. When a function call is made, extra instructions are executed to create a new activation record. In addition, in pipelined processors the pipeline is disrupted by function calls. Standard ML of New Jersey, Scheme, and some other functional languages have a mechanism where they optimize certain recursive functions by reducing the storage on the run-time stack and eliminating calls. In certain cases, recursive calls can be automatically transformed to code that can be executed using jump or branch instructions. For this optimization to be possible, the recursive function must be tail recursive. A tail recursive function is a function where the very last operation of the function is the recursive call to itself.

The *factorial* function is presented in Fig. 5.18. Is factorial tail recursive? The answer is no. Tail recursion happens when the very last thing done in a recursive function is a call to itself. The last thing done in Fig. 5.18 is the multiplication.

When factorial 6 is invoked, activation records are needed for seven invocations of the function, namely factorial 6 through factorial 0. Without each of these stack frames, the local values of n, n=6 through n=0, will be lost so that the multiplication at the end can not be carried out correctly.

At its deepest level of recursion all the information in the expression,

$$(6 * (5 * (4 * (3 * (2 * (1 * (factorial 0)))))))$$

is stored in the run-time execution stack.

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

Fig. 5.18 Factorial

```
fun factorial n =
    let fun tailfac(0,prod) = prod
          | tailfac(n,prod) = tailfac(n-1,prod*n)
    in
      tailfac(n,1)
    end
```

Fig. 5.19 Tail Recursive Factorial

Practice 5.16 Show the run-time execution stack at the point that factorial 0 is executing when the original call was factorial 6.
You can check your answer(s) in Section 5.26.16.

The *factorial* function can be written to be tail recursive. The solution is to use a technique similar to the *fib* function improvement made in Fig. 5.17. An accumulator is added to the function definition. An accumulator is an extra parameter that can be used to accumulate a value, much the way you would accumulate a value in a loop. The accumulator value is initially given the identity of the operation used to accumulate the value. In Fig. 5.19 the operation is multiplication. The identity provided as the initial value is 1.

The function presented in Fig. 5.19 is the tail recursive version of the *factorial* function. The tail recursive function is the *tailfac* helper function. Note that although *tailfac* is recursive, there is no need to save it's local environment when it calls itself since no computation remains after the call. The result of the recursive call is simply passed on as the result of the current function call. A function is tail recursive if its recursive call is the last action that occurs during any particular invocation of the function.

Practice 5.17 Use the accumulator pattern to devise a more efficient reverse function. The append function is not used in the efficient reverse function. HINT: What are we trying to accumulate? What is the identity of that operation?
You can check your answer(s) in Section 5.26.17.

5.14 Currying

A binary function, for example, + or @, takes both of its arguments at the same time. a+b will evaluate both *a* and *b* so that values can be passed to the addition operation. There can be an advantage in having a binary function take its arguments one at a

time. Such a function is called *curried* after Haskell Curry. ML functions take their parameters one at a time because all functions take exactly one argument. A curried function takes one argument as well. However, that function of one parameter may in turn return a function that takes a single argument. This is probably best illustrated with an example. Here is a function that takes a pair of arguments as its input via a single tuple.

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

The function *plus* takes one argument that just happens to be a tuple. Calling the function means providing it a single tuple.

```
- plus (5,8);
val it = 13 : int
```

ML functions can be defined with what looks like more than one parameter:

```
- fun cplus (a:int) b = a+b;
val cplus = fn : int -> (int -> int )
```

Observe the signature of the function *cplus*. It appears to take two arguments, but takes them one at a time. Actually, *cplus* takes only one argument. The *cplus* function returns a function that takes the second argument. The second function has no name.

```
- cplus 5 8;
val it = 13 : int
```

Function application is left associative. The parens below show the order of operations.

```
- (cplus 5) 8;
val it = 13 : int
```

The result of *(cplus 5)* is a function that adds *5* to its argument.

```
- cplus 5;
val it = fn : int -> int
```

We can give this function a name.

```
- val add5 = cplus 5;
val add5 = fn : int -> int
- add5 8;
val it = 13 : int
```

The *add5* function adds 5 to whatever might be passed to it.

Practice 5.18 Write a function that given an uncurried function of two arguments will return a curried form of the function so that it takes its arguments one at a time.

Write a function that given a curried function that takes two arguments one at a time will return an uncurried version of the given function.
You can check your answer(s) in Section 5.26.18.

Curried functions allow partial evaluation, a very interesting topic in functional languages, but beyond the scope of this text. It should be noted that Standard ML of New Jersey uses curried functions extensively in its implementation. *Appendix B* contains many functions whose signatures reflect that they are curried.

5.15 Anonymous Functions

The beginning of this chapter describes the lambda calculus. In that section we learned that functions can be characterized as first class objects. Functions can be represented by a lambda abstraction and don't have to be assigned a name. This is also true in SML. Functions in SML don't need names. The anonymous function $\lambda xy.y^2+x$ can be represented in ML as

```
fn x => fn y => y*y + x;
```

The anonymous function can be applied to a value in the same way a named function is applied to a value. Function application is always the function first, followed by the value.

```
- (fn x => fn y => y*y + x) 3 4;
val it = 19 : int
```

We can define a function by binding a lambda abstraction to an identifier:

```
- val f = fn x => fn y => y*y + x;
val f = fn: int -> int -> int
- f 3 4;
val it = 19 : int
```

This mechanism provides an alternative form for defining functions as long as they are not recursive; in a *val* declaration, the identifier being defined is not visible in the expression on the right side of the arrow. For recursive definitions a *val rec* expression is required. To define a recursive function using the anonymous function form you must use *val rec* to declare it.

```
- val rec fac = fn n => if n=0 then 1 else n*fac(n-1);
val fac = fn: int -> int
- fac 7;
val it = 5040:int
```

This *val rec* definition of a function is the way all functions are defined in SML. The functional form used when the keyword *fun* is used to define a function is translated into *val rec* form. The *fun* form of function definition is called *syntactic sugar*. Syntactic sugar refers to another way of writing something that gets treated the same way in either case. Usually *sugared* forms are the *nicer* way to write something.

5.16 Higher-Order Functions

The unique feature of functional languages is that functions are treated as first-class objects with the same rights as other objects, namely to be stored in data structures, to be passed as a parameter, and to be returned as function results. Functions can be bound to identifiers using the keywords fun, *val*, and *val rec* and may also be stored in structures. These are examples of functions being treated as values.

```
- val fnlist = [fn (n) => 2*n, abs, ~, fn (n) => n*n];
val fnlist = [fn,fn,fn,fn] : (int -> int) list
```

Notice each of these functions takes an int and returns an int. An ML function can be defined to apply each of these functions to a number. The *construction* function applies a list of functions to a value.

```
- fun construction  nil n = nil
    | construction (h::t) n = (h n)::(construction t n);
val construction = fn : ('a -> 'b) list -> 'a -> 'b list
- construction [op +, op *, fn (x,y) => x - y] (4,5);
val it = [9,20,~1] : int list
```

Construction is based on a functional form found in FP, an early functional programming language developed by John Backus. It illustrates the possibility of passing functions as arguments. Since functions are first-class objects in ML, they may be stored in any sort of structure. It is possible to imagine an application for a stack of functions or even a tree of functions.

A function is called higher-order if it takes a function as a parameter or returns a function as its result. Higher-order functions are sometimes called functional forms since they allow the construction of new functions from already defined functions.

The usefulness of functional programming comes from the use of functional forms that allow the development of complex functions from simple functions using abstract patterns. The *construction* function is one of these abstract patterns of computation. These functional forms, or patterns of computation, appear over and over again in programs. Programmers have recognized these patterns and have abstracted out the details to arrive at several commonly used higher-order functions. The next sections introduce several of these higher-order functions.

5.16.1 Composition

Composing two functions is a naturally higher-order operation that you have probably used in algrebra. Have you ever written something like f(g(x))? This operation can be expressed in ML. In fact, ML has a built-in operator called *o* which represents composition. This example code demonstrates how composition can be written and used.

```
- fun compose f g x = f (g x);
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
- fun add1 n = n+1;
val add1 = fn : int -> int
- fun sqr n:int = n*n;
val sqr = fn : int -> int
- val incsqr = compose add1 sqr;
val incsqr = fn : int -> int
- val sqrinc = compose sqr add1;
val sqrinc = fn : int -> int
```

Observe that these two functions, *incsqr* and *sqrinc*, are defined without the use of parameters.

```
- incsqr 5;
val it = 26 : int
- sqrinc 5;
val it = 36 : int
```

ML has a predefined infix function *o* that composes functions. Note that *o* is uncurried.

```
- op o;
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- val incsqr = add1 o sqr;
val incsqr = fn : int -> int
- incsqr 5;
val it = 26 : int
- val sqrinc = op o(sqr,add1);
val sqrinc = fn : int -> int
- sqrinc 5;
val it = 36 : int
```

5.16.2 Map

In SML, applying a function to every element in a list is called *map* and is predefined. It takes a unary function and a list as arguments and applies the function to each element of the list returning the list of results.

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- map add1 [1,2,3];
val it = [2,3,4] : int list
- map (fn n => n*n - 1) [1,2,3,4,5];
val it = [0,3,8,15,24] : int list
- map (fn ls => "a"::ls) [["a","b"],["c"],["d","e","f"]];
val it = [["a","a","b"],["a","c"],["a","d","e","f"]] :
           string list list
- map real [1,2,3,4,5];
val it = [1.0,2.0,3.0,4.0,5.0] : real list
```

```
fun map f nil = nil
  | map f (h::t) = (f h)::(map f t);
```

Fig. 5.20 The Map Function

The map function is predefined in the List structure, but is provided in Fig. 5.20 for your reference.

Practice 5.19 Describe the behavior (signatures and output) of these functions:

- map (map add1)
- (map map)

Invoking *(map map)* causes the type inference system of SML to report

```
language
stdIn:12.27-13.7 Warning: type vars not generalized
   because of value restriction are instantiated to
   dummy types (X1,X2,...)
```

This warning message is OK. It is telling you that to complete the type inference for this expression, SML had to instantiate a type variable to a dummy variable. When more type information is available, SML would not need to do this. The warning message only applies to the specific case where you created a function by invoking *(map map)*. In the presence of more information the type inference system will interpret the type correctly without any dummy variables.
You can check your answer(s) in Section 5.26.19.

5.16.3 Reduce or Foldright

Higher-order functions are developed by abstracting common patterns from programs. For example, consider the functions that find the sum or the product of a list of integers. In this pattern the results of the previous invocation of the function are used in a binary operation with the next value to be used in the computation.

In other words, to add up a list of values you start with either the first or last element of the list and then add it together with the value next to it. Then you add the result of that computation to the next value in the list and so on. When we start

with the end of the list and work our way backwards through the list the operation is sometimes called foldr (i.e. foldright) or reduce.

```
- fun sum nil = 0
    | sum ((h:int)::t) = h + sum t;

val sum = fn : int list -> int
- sum [1,2,3,4,5];
val it = 15 : int

- fun product nil = 1
    | product ((h:int)::t) = h * product t;

val product = fn : int list -> int
- product [1,2,3,4,5];
val it = 120 : int
```

Each of these functions has the same pattern. If we abstract the common pattern as a higher-order function we arrive at a common higher-order function called *foldr*. *foldr* is an abbreviation for foldright. The *foldr* function keeps applying its function to the result and the next item in the list.

```
- fun foldr f init nil = init
    | foldr f init (h::t) = f(h, foldr f init t);

val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr op + 0 [1,2,3,4,5];
val it = 15 : int
- foldr op * 1 [1,2,3,4,5];
val it = 120 : int
```

Now *sum* and *product* can be defined in terms of *reduce*.

```
- val sumlist = List.foldr (op +) 0;
val sumlist = fn : int list -> int
- val mullist = List.foldr op * 1;
val mullist = fn : int list -> int
- sumlist [1,2,3,4,5];
val it = 15 : int
- mullist [1,2,3,4,5];
val it = 120 : int
```

SML includes two predefined functions that reduce a list, *foldr* and *foldl* which stands for foldleft. They behave slightly differently.

```
- List.foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- List.foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- fun abdiff (m,n:int) = abs(m-n);
val abdiff = fn : int * int -> int
- foldr abdiff 0 [1,2,3,4,5];
val it = 1 : int
```

```
- foldl abdiff 0 [1,2,3,4,5];
val it = 3 : int
```

Practice 5.20 How does *foldl* differ from *foldr*? Determine the difference by looking at the example code in this section. Then, describe the result of these functions invocations.

- foldr op :: nil ls
- foldr op @ nil ls

You can check your answer(s) in Section 5.26.20.

5.16.4 Filter

A predicate function is a function that takes a value and returns true or false depending on the value. By passing a predicate function, it is possible to filter in only those elements from a list that satisfy the predicate. This is a commonly used higher-order function called *filter*. If we had to write filter ourselves, this is how it would be written. This example also shows how it might be used.

```
- fun filter bfun nil = nil
    | filter bfun (h::t) = if bfun h then h::filter bfun t
                           else filter bfun t;

val it = fn : ('a -> bool) -> 'a list -> 'a list
- even;
val it = fn : int -> bool
- filter even [1,2,3,4,5,6];
val it = [2,4,6] : int list
- filter (fn n => n > 3) [1,2,3,4,5,6];
val it = [4,5,6] : int list
```

Practice 5.21 Use filter to select numbers from a list that are

- divisible by 7
- greater than 10 or equal to zero

You can check your answer(s) in Section 5.26.21.

5.17 Continuation Passing Style

Continuation Passing Style (or CPS) is a way of writing functional programs where control is made explicit. In other words, the continuation represents the remaining work to be done. This style of writing code is interesting because the style is used in the SML compiler. To understand cps it's best to look at an example. Let's consider the len function for computing the length of a list.

```
- fun len nil = 0
    | len (h::t) = 1+(len t);
val len = fn : 'a list -> int
```

To transform this to cps form we represent the rest of the computation explicitly as a parameter called k. In this way, whenever we need the continuation of the calculation, we can just write the identifier k. Here's the cps form of len and an example of calling it.

```
- fun cpslen nil k = k 0
    | cpslen (h::t) k = cpslen t (fn v => (k (1 + v)));
val cpslen = fn : 'a list -> (int -> 'b) -> 'b
- cpslen [1,2,3] (fn v => v);
val it = 3 : int
```

Practice 5.22 Trace the execution of cpslen to see how it works and how the continuation is used.
You can check your answer(s) in Section 5.26.22.

Notice that the recursive call to cpslen is the last thing that is done. This function is tail recursive. However, tail recursion elimination cannot be applied because the function returns a function and recursively calls itself with a function as a parameter. CPS is still important because it can be optimized by a compiler. In addition, since control flow is explicit (passed around as k), function calls can be implemented with jumps and many of the jumps can be eliminated if the code is organized in the right way.

Eliminating calls and jumps is important since calls have the effect of interrupting pipelines in RISC processors. Since functional languages make lots of calls, one of the criticisms of functional languages is that they were inefficient. With the optimization of CPS functions, functional languages get closer to being as efficient as imperative languages. In addition, as cache sizes and processor speeds increase the performance difference becomes less and less of an issue.

Practice 5.23 Write a function called *depth* that prints the longest path in a binary tree. First create the datatype for a binary tree. You can use the *Int.max* function in your solution, which returns the maximum of two integers.
First write a non-cps *depth* function, then write a cps *cpsdepth* function.
You can check your answer(s) in Section 5.26.23.

5.18 Input and Output

SML contains a TextIO structure as part of the basis library. The signature of the functions in the TextIO structure is given in *Appendix B*. It is possible to read and write strings to streams using this library of functions. The usual standard input, standard output, and standard error streams are predefined. Here is an example of reading a string from the keyboard. Explode is used on the string to show the vector type is really the string type. It also shows how to print something to a stream.

```
- val s = TextIO.input(TextIO.stdIn);
hi there
val s = "hi there\n" : vector
- explode(s);
val it = [#"h",#"i",#" ",#"t",#"h",#"e",
            #"r",#"e",#"\n"] : char list
- TextIO.output(TextIO.stdOut,s^"How are you!\n");
hi there
How are you!
- val it = () : unit
```

Since streams can be directed to files, the screen, or across the network, there really isn't much more to input and output in SML. Of course if you are opening your own stream it should be closed when you are done with it. Program termination will also close any open streams.

There are some TextIO functions that may or may not return a value. In these cases an *option* is returned. An *option* is a value that is either *NONE* or *SOME value*. An option is SML's way of dealing with functions that may or may not succeed. Functions must always return a value or end with an exception. To prevent the exception handling mechanism from being used for input operations that may or may not succeed, this idea of an option was created. Options fit nicely into the strong typing that SML provides. The *input1* function of the TextIO structure reads exactly one character from the input and returns an *option* as a result. The reason it returns an *option* and not the character directly is because the stream might not be ready for reading. The *valOf* function can be used to get the value of an *option* that is not *NONE*.

```
- val u = TextIO.input1(TextIO.stdIn);
```

```
hi there
val u = SOME #"h" : elem option
=
= ^C
Interrupt
- u;
val it = SOME #"h" : elem option
- val v = valOf(u);
val v = #"h" : elem
```

5.19 Programming with Side-effects

Standard ML is not a pure functional language. It is possible to write programs with side effects, such as reading from and writing to streams. To write imperative programs the language should support sequential execution, variables, and possibly loops. All three of these features are available in SML. The following sections show you how to use each of these features.

5.19.1 Variables in Standard ML

There is only one kind of variable in Standard ML. Variables are called references. It is interesting to note that you cannot update an integer, real, string, or many other types of values in SML. All these values are immutable. They cannot be changed once created. That is a nice feature of a language because then you don't have to worry about the distinction between a reference to a value and the value itself. Array objects are mutable because they contain a list of references.

A reference in Standard ML is typed. It is either a reference to an *int*, or a *string*, or some other type of data. References can be mutated. So a reference can be updated to point to a new value as your program executes. Declaring and using a reference variable is shown in this example code. In SML a variable is declared by creating a reference to a value of a particular type.

```
- val x = ref 0;
val x = ref 0 : int ref
```

The exclamation point is used to refer to the value to which a reference points. This is called the dereference operator. It is the similar to the star (i.e. *) in C++ which dereferences a pointer.

```
- !x;
val it = 0 : int
- x := !x + 1;
val it = () : unit
- !x;
val it = 1 : int
```

```
let val x = ref 0
in
  x:= !x + 1;
  TextIO.output(TextIO.stdOut,"The new value of x is "^
                Int.toString(!x)^"\n");
  !x
end
```

Fig. 5.21 Sequential Execution

The assignment operator (i.e. *:=*) operator updates the reference variable to point to a new value. The result of assignment is the empty tuple which has a special type called *unit*. Imperative programming in SML will often result in the unit type. Unlike ordinary identifiers you can bind to values using a *let val id = Expr in Expr end*, a reference can truly be updated to point to a new value.

It should be noted that references in Standard ML are typed. When a reference is created it can only point to a value of the same type it was originally created to refer to. This is unlike references in Python, but is similar to references in Java. A reference refers to a particular type of data.

5.19.2 Sequential Execution

If a program is going to assign variables new values or read from and write to streams it must be able to execute statements or expressions sequentially. There are two ways to write a sequence of expressions in SML. When you write a *let val id = Expr in Expr end* expression, the *Expr* in between the *in* and *end* may be a sequence of expressions. A sequence of expressions is semicolon separated. The code in Fig. 5.21 demonstrates how to write a sequence of expressions.

Evaluating this expression produces the following output.

```
The new value of x is 1
val it = 1 : int
```

In Fig. 5.21 semicolons separate the expressions in the sequence. Notice that semicolons don't terminate each line as in C++ or Java. Semicolons in SML are expression separators, not statement terminators. The last expression in a sequence of expressions is the value of the expression. All previously computed values in the sequential expression are thrown away. The *!x* is the last expression in the sequence in Fig. 5.21 so *1* is yielded as the value of the expression.

There are times when you may wish to evaluate a sequence of expressions in the absence of a *let* expression. In that case the sequence of expressions may be surrounded by parens. A left paren can start a sequence of expressions terminated by a right paren. The sequence of expressions is semicolon separated in either case. Here is some code that prints the value of x to the screen and then returns x + 1.

```
(TextIO.output(TextIO.stdOut,"The value of x is" ^
 Int.toString(x);
 x+1)
```

5.19.3 Iteration

Strictly speaking, variables and iteration are not needed in a functional language. Parameters can be passed in place of variable declarations. Recursion can be used in place of iteration. However, there are times when an iterative function might make more sense. For instance, when reading from a stream it might be more efficient to read the stream in a loop, especially when the stream might be large. A recursive function could overflow the stack in that case unless the recursive function were tail recursive and could be optimized to remove the recursive call.

A while loop in SML is written as *while Expr do Expr*. As is usual with while loops, the first *Expr* must evaluate to a boolean value. If it evaluates to *true* then the second *Expr* is evaluated. This process is repeated until the first *Expr* returns *false*.

5.20 Exception Handling

An exception occurs in SML when a condition occurs that requires special handling. If no special handling is defined for the condition the program terminates. As with most modern languages, SML has facilities for handling these exceptions and for raising user-defined exceptions. Consider the maxIntList function you wrote in practice problem 5.13. You probably had to figure out what to do if an empty list was passed to the function. One way to handle this is to raise an exception.

```
exception emptyList;

fun maxIntList [] = raise emptyList
  | maxIntList (h::t) = Int.max(h,maxIntList t) handle
                                emptyList => h
```

Invoking the maxIntList on an empty list can be handled using an exception handling expression. The handle clause uses pattern matching to match the right exception handler. To handle any exception the pattern _ can be used. The underscore matches anything. Multiple exceptions can be handled by using the vertical bar (i.e. |) between the handlers.

5.21 Encapsulation in ML

ML provides two language constructs that enable programmers to define new datatypes and hide their implementation details. The first of these language constructs we'll look at is the signature. The other construct is the structure.

5.21.1 Signatures

A signature is a means for specifying a set of related functions and types without providing any implementation details. This is analogous to an interface in Java or a template in C++. Consider the datatype consisting of a set of elements. A set is a group of elements with no duplicate values. Sets are very important in many areas of Computer Science and Mathematics. Set theory is an entire branch of mathematics. If we wanted to define a set in ML we could write a signature for it as follows.

The signature of a group of set functions and a set datatype is provided in Fig. 5.22. Notice this datatype is parameterized by a type variable so this could be a signature for a set of anything. You'll also notice that while the type parameter is $'a$ there are type variables named $''a$ within the signature. This is because some of these functions rely on the equals operator. In ML the equals operator is polymorphic and cannot be instantiated to a type. When this signature is used in practice the $'a$ and $''a$ types will be correctly instantiated to the same type.

Before a signature can be used, each of these functions must be implemented in a structure that implements the signature. This encapsulation allows a programmer to write code that uses these set functions without regards to their implementation.

```
1  signature SetSig =
2  sig
3     exception Choiceset
4     exception Restset
5     datatype 'a set = Set of 'a list
6     val emptyset   : 'a set
7     val singleton  : 'a -> 'a set
8     val member     : ''a -> ''a set -> bool
9     val union      : ''a set -> ''a set -> ''a set
10    val intersect  : ''a set -> ''a set -> ''a set
11    val setdif     : ''a set -> ''a set -> ''a set
12    val card       : 'a set -> int
13    val subset     : ''a set -> ''a set -> bool
14    val simetdif   : ''a set -> ''a set -> ''a set
15    val forall     : ''a set -> (''a -> bool) -> bool
16    val forsome    : ''a set -> (''a -> bool) -> bool
17    val forsomeone : 'a set -> ('a -> bool) -> bool
18 end
```

Fig. 5.22 The Set Signature

An implementation must be provided before the program can be run. However, if a better implementation comes along later it can be substituted without changing any of the code that uses the set signature.

5.21.2 Implementing a Signature

To implement a signature we can use the struct construct that we've seen before. In this case it is done as follows. A partial implementation of the *SetSig* signature is provided in Fig. 5.23.

Of course, the entire implementation of all the set functions in the signature is required. Some of these functions are left as an exercise.

Practice 5.24 1. Write the card function. Cardinality of a set is the size of the set.
2. Write the intersect function. Intersection of two sets are just those elements that the two sets have in common. Sets do not contain duplicate elements.

You can check your answer(s) in Section 5.26.24.

```
(***** An Implementation of Sets as a SML datatype *****)

structure Set : SetSig =
struct

exception Choiceset
exception Restset

datatype 'a set = Set of 'a list

val emptyset = Set []

fun singleton e = Set [e]

fun member e (Set [])     = false
  | member e (Set (h::t)) = (e = h) orelse member e (Set t)

fun notmember element st = not (member element st)

fun union (s1 as Set L1) (s2 as Set L2) =
    let fun noDup e = notmember e s2
     in
       Set ((List.filter noDup L1)@(L2))
    end

...
end
```

Fig. 5.23 A Set Structure

5.22 Type Inference

Perhaps Standard ML's strongest point is the formally proven soundness of its type inference system. ML's type inference system is guaranteed to prevent any run-time type errors from occurring in a program. This turns out to prevent many run-time errors from occurring in your programs. Projects like the Fox Project have shown that ML can be used to produce highly reliable large software systems.

The origins of type inference include Haskell Curry and Robert Feys who in 1958 devised a type inference algorithm for the simply typed lambda calculus. In 1969 Roger Hindley worked on extending this type inference algorithm. In 1978 Robin Milner independently from Hindley devised a similar type inference system proving its soundness. In 1985 Luis Damas proved Milner's algorithm was complete and extended it to support polymorphic references. This algorithm is called the Hindley-Milner type inference algorithm or the Milner-Damas algorithm. The type inference system is based on a very powerful concept called unification.

Unification is the process of using type inference rules to bind type variables to values. The type inference rules look like this.

IfThen

$$\frac{\varepsilon \vdash e_1 : bool \;\; \varepsilon \vdash e_2 : \alpha \;\; \varepsilon \vdash e_3 : \alpha}{\varepsilon \vdash if\ e_1\ then\ e_2\ else\ e_3 : \alpha}$$

This rule says that for an if-then expression to be correctly typed, the type of the first expression must be a *bool* and the types of the second and third expression must be unifiable. If those preconditions hold, then the type of the if-then expression is given by the type of either of the second two expressions (since they are the same). Unification happens when α is written twice in the rule above. The ε is the presence of type information that is used when determining the types of the three expressions and is called the type environment.

Here are two examples that suggest how the type inference mechanism works. In this example we determine the type of the following function.

```
fun f(nil,nil) = nil
  | f(x::xs,y::ys) = (x,y)::f(xs,ys);
```

The function f takes one parameter, a pair.

```
f: 'a * 'b -> 'c
```

From the nature of the argument patterns, we conclude that the three unknown types must be lists.

```
f: ('p list) * ('s list) -> 't list
```

The function imposes no constraints on the domain lists, but the codomain list must be a list of pairs because of the cons operation *(x,y) ::*. We know *x:'p* and *y:'s*. Therefore *'t='p *'s*.

```
f: 'p list * 's list -> ('p * 's) list
```

where *'p* and *'s* are any ML types. In this example the type of the function *g* is inferred.

```
fun g h x = if null x then nil
            else
              if h (hd x) then g h (tl x)
              else (hd x)::g h (tl x);
```

The function g takes two parameters, one at a time.

```
g: 'a -> 'b -> 'c
```

The second parameter, x, must serve as an argument to *null*, *hd*, and *tl*; it must be a list.

```
g: 'a -> ('s list) -> 'c
```

The first parameter, *h*, must be a function since it is applied to *hd x*, and its domain type must agree with the type of elements in the list. In addition, *h* must produce a boolean result because of its use in the conditional expression.

```
g: ('s -> bool) -> ('s list) -> 'c
```

The result of the function must be a list since the base case returns *nil*. The result list is constructed by the code *(hd x) ::g h (tl x)*, which adds items of type *'s* to the resulting list.

Therefore, the type of *g* must be:

```
g: ('s -> bool) -> 's list -> s list
```

Chapter 8 explores type inference in much more detail. A type checker for Standard ML is developed using Prolog, a programming language ideally suited to problems involving unification.

5.23 Building a Prefix Calculator Interpreter

The datatype definition in Fig. 5.12 provided an abstract syntax tree definition for a calculator language with one memory location. A related prefix calculator expression language is relatively easy to define and from that we can build an interpreter of prefix calculator expressions. Prefix expressions are comprised of an operator first followed by an expression or expressions. The prefix calculator expression language is defined by this LL(1) grammar.

$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$

$\mathcal{T} = \{S, R, number, \ , +, -, *, /\}$

$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow + \ E \ E \mid - \ E \ E \mid * \ E \ E \mid / \ E \ E \mid \sim E \mid S \ E \mid R \mid number$$

```
1  fun delimiter #" " = true
2    | delimiter #"\t" = true
3    | delimiter #"\n" = true
4    | delimiter _ = false
5
6  fun run() =
7    (TextIO.output(TextIO.stdOut,"Please enter a prefix calculator expression: ");
8     TextIO.flushOut(TextIO.stdOut);
9     let val line = TextIO.inputLine(TextIO.stdIn)
10        val tokens = String.tokens delimiter (valOf line)
11        val (ast,remainingTokens) = E(tokens)
12        val result = eval(ast)
13    in
14      if length(remainingTokens) <> 0 then
15        raise(eofException)
16      else ();
17      TextIO.output(TextIO.stdOut,"The answer is: "^Int.toString(result)^"\n")
18    end
19    handle eofException =>
20              TextIO.output(TextIO.stdOut,
21                "You entered an invalid prefix expression.\n")
22           | Option =>
23              TextIO.output(TextIO.stdOut,
24                "You entered invalid characters in the prefix expression.\n"))
```

Fig. 5.24 The Prefix Calc Interpreter Run Function

The only non-terminal in this grammar is *E*. The *S* is the store operator which stores the expression that follows it in the memory location. The *R* is the recall operator. The tilde (i.e. ~) is the unary negation operator. To implement an interpreter for this language we must first parse the expression and build an abstract syntax tree. Then the abstract syntax tree can be evaluated. The entire process can be encapsulated in a *run* function.

The *run* function that provides the overall design of the prefix calculator interpreter is provided in Fig. 5.24.

A number of things should be explained about this code. Line 8 flushes standard output. Without it the prompt does not print before the program starts waiting for input. Line 9 gets a line of input from the user. It is returned as a string option so on line 10 the *valOf* function is applied to get the string or raise an Option exception if *NONE* had been returned.

Line 10 calls the *tokens* function. All the tokens must be separated by spaces or tabs for the program to read the tokens correct. Here is an example of running this code.

```
- run();
Please enter a prefix calculator expression: + * S ~ 6 R 5
The answer is: 41
val it = () : unit
```

Line 11 calls the parser to parse the list of tokens. In this case, the list of tokens is passed to the parsing function. The parser returns a tuple with the AST as the first item of the tuple and the rest of the tokens as the second result. After parsing, line

14 checks to see that there are no more tokens left after parsing. If there are, then the *eofExceptioni* is raised.

Line 12 calls the evaluator function *eval* to interpret the AST. The *eval* function returns the result of evaluating the tree. Line 17 prints the result to the screen.

There are two handled exceptions. If the *eofException* is thrown then the expression did not parse correctly. If the Option exception is thrown there was a bad token in the input. Note that only integers are allowed for numbers in this implementation. This was decided by the AST definition in Fig. 5.12.

5.23.1 The Prefix Calc Parser

Parsing the expression is easy thanks to the LL(1) grammar for prefix calculator expressions. The *E* function is defined using pattern-matching in Fig. 5.25. Each time a token is consumed it is simply omitted from the remaining list of tokens. The tokens are single-threaded through the function. This just means the left over tokens are always passed on to the next piece to be parsed and the remaining tokens are always returned along with the AST when the *E* function returns.

The parser doesn't do any evaluation of the data. It simply works on building an AST for the expression. The evaluation of the AST comes later, by the evaluator.

Notice in line 39 that the *valOf* function is used on the result of the *Int.fromString* function. If the string being converted is not a valid value, the *valOf* will raise the Option exception terminating the run function with an appropriate error message.

Line 43 of Fig. 5.25 handles getting to the end of the input (i.e. the list of tokens) earlier than is expected. If the parser reaches this case the original expression was mal-formed and throwing the *eofException* is the appropriate response.

5.23.2 The AST Evaluator

To complete implementation of the prefix calculator the AST needs to be evaluated. The *eval* function presented in Fig. 5.26 provides this evaluation function. Line 1 declares a memory reference that is imperatively updated with the value stored in the calculator's memory.

Lines 2–9 provide the traditional binary operations of addition, subtraction, multiplication, and division. Because this calculator is only an integer calculator, the integer division *div* is used. Unary negation occurs on lines 10 and 11.

Line 12 stores a value in the memory of the calculator by first evaluating the subtree and then storing the value before returning it. Line 18 is responsible for recalling the value by returning the dereferenced memory location.

```
exception eofException;

fun E ("+"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (add'(ast1,ast2),rest2)
  end
  | E ("-"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (sub'(ast1,ast2),rest2)
  end
  | E ("*"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (prod'(ast1,ast2),rest2)
  end
  | E ("/"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (div'(ast1,ast2),rest2)
  end
  | E ("~"::rest) =
  let val (ast,rest1) = E(rest)
  in
    (negate'(ast),rest1)
  end
   | E ("S"::rest) =
  let val (ast,rest1) = E(rest)
  in
    (store'(ast),rest1)
  end
  | E ("R"::rest) = (recall',rest)
  | E (x::rest) =
    let val i = valOf(Int.fromString(x))
    in
      (integer'(i),rest)
    end
  | E nil = raise eofException;
```

Fig. 5.25 The Parser

```
1  val memory = ref 0;
2  fun eval(add'(t1,t2)) =
3        eval(t1) + eval(t2)
4    | eval(sub'(t1,t2)) =
5        eval(t1) - eval(t2)
6    | eval(prod'(t1,t2)) =
7        eval(t1) * eval(t2)
8    | eval(div'(t1,t2)) =
9        eval(t1) div eval(t2)
10   | eval(negate'(t)) =
11       ~1 * eval(t)
12   | eval(store'(t)) =
13     let val x = eval(t)
14     in
15       memory := x;
16       x
17     end
18   | eval(recall') = !memory
19   | eval(integer'(x)) = x
```

Fig. 5.26 The Evaluator

5.23.3 Imperative Programming Observations

There are a couple of Standard ML syntax issues that are good to recognize at this point. In Fig. 5.24, line 7 begins with a left paren. The left paren can be used to construct a tuple in Standard ML, but it is also used to *begin* a sequence of expressions. The last right paren on line 24 *ends* the sequence. Expressions are separated by semicolons in a sequence of expressions. This occurs on line 16 of Fig. 5.24. No semicolon appears after the expression on line 17 because semicolons only separate expressions, they do not terminate them. On line 16 the else clause has a *unit* (i.e. *()*) as its result. This is because the type generated by raising an exception is a *unit*, and the *then* and *else* clause return types must match.

5.24 Chapter Summary

This chapter introduced functional programming. For many this is a new way of thinking about programming. Recursion is the main pattern used in computing when writing in a functional programming style. Higher-order functions are an important part of functional programming. Certain patterns appear often in functional programs and these patterns have been implemented as some common higher-order functions like *map*, *filter*, *foldr*, and others.

An important thing to learn from this chapter is that functional programming is more declarative and less prescriptive than programming in an imperative language like C++ or Java. Standard ML is a good functional programming language but other languages like C++, Java, and Python support functional programming as well.

Standard ML has a strong type checker that has been proven sound and complete. That means that while more time is spent removing type errors from programs, much less time is spent debugging Standard ML programs. Experiments like the Fox Project at Carnegie Mellon have shown this is true for large software systems written in Standard ML as well.

Much more can be learned about Standard ML and the next chapter not only looks at some Standard ML tools for language implementation, but it also describes the implementation of a compiler that translates Standard ML to JCoCo assembly language.

Jeffrey Ullman's book on functional programming in Standard ML is a very good introduction and reference for Standard ML. It is more thorough than the topics provided in this text and contains many topics not covered here including discussion of arrays, functors, and sharings along with a few of the Basis structures. The topics presented here and in the next chapter give you a good introduction to the ideas and concepts associated with functional programming. Ullman's book and on-line tutorials and manual pages are another great resource for learning functional programming.

5.25 Exercises

In the exercises below you are encouraged to write other functions that may help you in your solutions. You might have better luck with some of the harder ones if you solve a simpler problem first that can be used in the solution to the harder problem.

You may wish to put your solutions to these problems in a file and then

```
- use "thefile";
```

in SML. This will make writing the solutions easier. You can try the solutions out by placing tests right within the same file. You should always comment any code you write. Comments in SML are preceded with a *(** and terminated with a ***)*.

1. Reduce $(\lambda z.z + z)((\lambda x.\lambda y.x + y)\ 4\ 3)$ by normal order and applicative order reduction strategies. Show the steps.
2. How does the SML interpreter respond to evaluating each of the following expressions? Evaluate each of these expression in ML and record what the response of the ML interpreter is.

   (a) *8 div 3;*
   (b) *8 mod 3;*
   (c) *"hi"^"there";*

(d) *8 mod 3 = 8 div 3 orelse 4 div 0 = 4;*
(e) *8 mod 3 = 8 div 3 andalso 4 div 0 = 4;*

3. Describe the behavior of the *orelse* operator in exercise 2 by writing an equivalent *if then* expression. You may use nested if expressions. Be sure to try your solution to see you get the same result.
4. Describe the behavior of the *andalso* operator in exercise 2 by writing an equivalent *if then* expression. Again you can use nested if expressions.
5. Write an expression that converts a character to a string.
6. Write an expression that converts a real number to the next lower integer.
7. Write an expression that converts a character to an integer.
8. Write an expression that converts an integer to a character.
9. What is the signature of the following functions? Give the signature and an example of using each function.

   (a) hd
   (b) tl
   (c) explode
   (d) concat
   (e) :: - This is an infix operator. Use the prefix form of *op ::* to get the signature.

10. The greatest common divisor of two integers, x and y, can be defined recursively. If y is zero then x is the greatest common divisor. Otherwise, the greatest common divisor of x and y is equal to the greatest common divisor of y and the remainder x divided by y. Write a recursive function called *gcd* to determine the greatest common divisor of x and y.
11. Write a recursive function called *allCaps* that given a string returns a capitalized version of the string.
12. Write a recursive function called *firstCaps* that given a list of strings, returns a list where the first letter of each of the original strings is capitalized.
13. Using pattern matching, write a recursive function called *swap* that swaps every pair of elements in a list. So, if *[1,2,3,4,5]* is given to the function it returns *[2,1,4,3,5]*.
14. Using pattern matching, write a function called *rotate* that rotates a list by n elements. So, *rotate(3,[1,2,3,4,5])* would return *[4,5,1,2,3]*.
15. Use pattern matching to write a recursive function called *delete* that deletes the nth letter from a string. So, *delete(3,"Hi there")* returns *"Hi here"*. HINT: This might be easier to do if it were a list.
16. Again, using pattern matching write a recursive function called *intpow* that computes x^n. It should do so with $O(log\ n)$ complexity.
17. Rewrite the *rotate* function of exercise 14 calling it *rotate2* to use a helper function so as to guarantee $O(n)$ complexity where n is the number of positions to rotate.
18. Rewrite exercise 14's *rotate(n,lst)* function calling it *rotate3* to guarantee that less than l rotations are done where l is the length of the list. However, the

outcome of rotate should be the same as if you rotated *n* times. For instance, calling the function as *rotate3(6,[1,2,3,4,5])* should return *[2,3,4,5,1]* with less than 5 recursive calls to *rotate3*.

19. Rewrite the *delete* function from exercise 15 calling it *delete2* so that it is curried.
20. Write a function called *delete5* that always deletes the fifth character of a string.
21. Use a higher-order function to find all those elements of a list of integers that are even.
22. Use a higher-order function to find all those strings that begin with a lower case letter.
23. Use a higher-order function to write the function *allCaps* from exercise 11.
24. Write a function called *find(s,file)* that prints the lines from the file named *file* that contain the string *s*. You can print the lines to *TextIO.stdOut*. The *file* should exist and should be in the current directory.
25. Write a higher-order function called *transform* that applies the same function to all elements of a list transforming it to the new values. However, if an exception occurs when transforming an element of the list, the original value in the given list should be used. For instance,

```
- transform (fn x => 15 div x) [1,3,0,5]
val it = [15,5,0,3] : int list
```

26. The natural numbers can be defined as the set of terms constructed from 0 and the $succ(n)$ where n is a natural number. Write a datatype called *Natural* that can be used to construct natural numbers like this. Use the capital letter O for your zero value so as not to be confused with the integer 0 in SML.
27. Write a *convert(x)* function that given a natural number like that defined in exercise 26 returns the integer equivalent of that value.
28. Define a function called *add(x,y)* that given x and y, two natural numbers as described in exercise 26, returns a natural number that represents the sum of x and y. For example,

```
- add(succ(succ(O)),succ(O))
val it = succ(succ(succ(O))) : Natural
```

    You may NOT use *convert* or any form of it in your solution.
29. Define a function called *mul(x,y)* that given x and y, two natural numbers as described in exercise 26, returns a natural that represents the product of x and y. You may NOT use *convert* or any form of it in your solution.
30. Using the *add* function in exercise 28, write a new function *hadd* that uses the higher order function called *foldr* to add together a list of natural numbers.
31. The prefix calculator intpreter presented at the end of this chapter can be implemented a little more concisely by having the parser not only parse the prefix expression, but also evaluate the expression at the same time. If this is to be done, the parser ends up returning a *unit* because the parser does not need to return an AST since the expression has already been evaluated. This means the definition of the *AST* is no longer needed. Rewrite the prefix calculator code presented at the end of this chapter to combine the *parse* and *eval* functions.

Remove any unneeded code from your implementation but be sure to cover all the error conditions as the version presented in this chapter.

32. Alter the prefix expression calculator to accept either integers or floating point numbers as input. The result should always be a float in this implementation.
33. Add an input operator to the prefix calculator. In this version, expressions like *+ S I 5* when evaluated would prompt the user to enter a value when the *I* was encountered. This expression, when evaluated, would cause the program to respond as follows.

```
Please enter a prefix calculator expression: + S I 5
? 4
The answer is: 9
```

34. The prefix calculator intrepeter presented in this chapter can be transformed into a prefix calculator compiler by having the program write a file called *a.casm* with a JCoCo program that when run evaluates the compiled prefix calculator expression. Alter the code at the end of this chapter to create a prefix caclulator compiler. Running the compiler should work like this.

```
% sml
- use "prefixcalc.sml";
- run();
Please enter a prefix calculator expression: + S 6 5
- <ctrl-d>
% coco a.casm
The answer is: 11
```

35. For an extra hard project, combine the previous two exercises into one prefix calc compiler whose programs when run can gather input from the user to be used in the calculation.
36. Rewrite the prefix calculator project to single thread the memory location through the *eval* function as shown in pattern Completing this project removes the imperatively updated memory location from the code and replaces it with a single-threaded argument to the *eval* function.

5.26 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

5.26.1 Solution to Practice Problem 5.1

Addition is not commutative in Pascal or Java. The problem is that a function call, which may be one or both of the operands to the addition operator, could have a

side-effect. In that case, the functions must be called in order. If no order is specified within expression evaluation then you can't even reliably write code with side-effects within an expression.

Here's another example of the problem with side-effects within code. In the code below, it was observed that when the code was compiled with one C++ compiler it printed 1,2 while with another compiler it printed 1,1. In this case, the language definition is the problem. The C++ language definition doesn't say what should happen in this case. The decision is left to the compiler writer.

```
int x = 1;
cout << x++ << x << endl;
```

The practice problem writes 17 as written. If the expression were b+a() then 15 would be written.

5.26.2 Solution to Practice Problem 5.2

With either normal order or applicative order function application is still left-associative. There is no choice for the initial redex.

$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)$
$\Rightarrow (\lambda yz.\underline{(\lambda x.x)z(yz)})(\lambda xy.x)$
$\Rightarrow \underline{(\lambda yz.z(yz))(\lambda xy.x)}$
$\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})$
$\Rightarrow \lambda z.z(\lambda y.z)\square$

5.26.3 Solution to Practice Problem 5.3

Normal Order Reduction

$\underline{(\lambda x.y)((\lambda x.xx)(\lambda x.xx))}$
$\Rightarrow y$

Applicative Order Reduction

$(\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$
$\Rightarrow (\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$
$\Rightarrow (\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$
$\Rightarrow (\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$ ...
You get the idea.

5.26.4 Solution to Practice Problem 5.4

```
x div 6;
Real.round(Real.fromInt(x) * y);
x / 6.3;
x mod y
```

5.26.5 Solution to Practice Problem 5.5

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

5.26.6 Solution to Practice Problem 5.6

The recursive definition is fib(0) = 0, fib(1) = 1, fib(n)=fib(n-1)+fib(n-2). The recursive function is:

```
fun fib(n) = if n = 0 then 1 else
             if n = 1 then 1 else
             fib(n-1) + fib(n-2)
```

5.26.7 Solution to Practice Problem 5.7

The solutions below are example solutions only. Others exist. However, the problem with each invalid list is not debatable.

1. You cannot cons a character onto a string list. “a” ::[”beautiful day”]
2. You cannot cons two strings. The second operand must be a list. “hi” :: [”there”]
3. The element comes first in a cons operation and the list second. “you” ::[”how”,”are”]
4. Lists are homogeneous. Reals and integers can’t be in a list together. [1.0,2.0,3.5,4.2]
5. Append is between two lists.2 ::[3,4] or *[2]@[3,4]*
6. Cons works with an element and a list, not a list and an element. 3 ::[]

5.26.8 Solution to Practice Problem 5.8

```
fun explode(s) =
  if s ="" then []
  else String.sub(s,0){~:}{:}
       (explode(String.substring(s,1,String.size(s)-1)))
```

5.26.9 Solution to Practice Problem 5.9

```
fun reverse(L) =
  if null L then []
  else append(reverse(tl(L)),[hd(L)])
```

5.26.10 Solution to Practice Problem 5.10

```
fun reverse([]) = []
  | reverse(h{~:}{:}t) = reverse(t)@[h]
```

5.26.11 Solution to Practice Problem 5.11

```
let val x = 10
in
    (* 1. Value of x = 10 *)
    let val x = x+1
    in
      (* 2. Value of x = 11 (hidden x still is 10) *)
      x
    end;
    (* 3. Value of x = 10 (hidden x is visible again) *)
    x
end
```

5.26.12 Solution to Practice Problem 5.12

```
datatype intlist = nil' | cons of int * intlist;
```

5.26.13 Solution to Practice Problem 5.13

```
fun maxIntList nil' = valOf(Int.minInt)
  | maxIntList (cons(x,xs)) = Int.max(x,maxIntList xs)
```

or

```
fun maxIntList (cons(x,nil')) = x
| maxIntList (cons(x,xs)) = Int.max(x,maxIntList xs)
```

The second solution will cause a pattern match nonexhaustive warning. That should be avoided, but is OK in this case. The second solution will raise a pattern match exception if an empty list is given to the function. See the section on exception handling for a better solution to this problem.

5.26.14 Solution to Practice Problem 5.14

The first step in the solution is to determine the number of calls required for values of n. Consulting Fig. 5.16 shows us that the number of calls are 1, 1, 3, 5, 9, 15, 25, etc. The next number in the sequence can be found by adding together two previous plus one more for the initial call.

The solution is that for $n \geq 3$ the function 1.5^n bounds the number of calls on the lower side while 2^n bounds it on the upper side. Therefore, the number of calls increases exponentially.

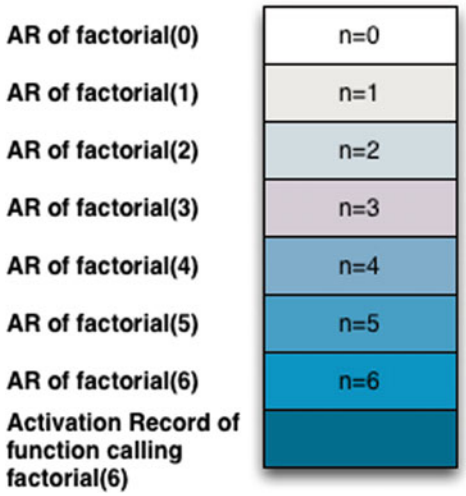

Fig. 5.27 The run-time stack when factorial(6) is called at its deepest point

5.26.15 Solution to Practice Problem 5.15

The cons operation is called n times where n is the length of the first list when append is called. When reverse is called it calls append with $n-1$ elements in the first list the first time. The first recursive call to reverse calls append with $n-2$ elements in the first list. The second recursive call to reverse calls append with $n-3$ elements in the first list. If we add up $n-1+n-2+n-3+...$ we end up with $\sum_{i=1}^{n-1} i = ((n-1)n)/2$. Multiplying this out leads to an n^2 term and the overall complexity of reverse is $O(n^2)$.

5.26.16 Solution to Practice Problem 5.16, see Fig. 5.27

5.26.17 Solution to Practice Problem 5.17

This solution uses the accumulator pattern and a helper function to implement a linear time reverse.

```
fun reverse(L) =
    let fun helprev (nil, acc) = acc
          | helprev (h::t, acc) = helprev(t,h::acc)
    in
      helprev(L,[])
    end
```

5.26.18 Solution to Practice Problem 5.18

This solution is surprisingly hard to figure out. In the first, f is certainly an uncurried function (look at how it is applied). The second requires f to be curried.

```
- fun curry f x y = f(x,y)
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

- fun uncurry f (x,y) = f x y
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

5.26.19 Solution to Practice Problem 5.19

The first takes a list of lists of integers and adds one to each integer of each list in the list of lists.

The second function takes a list of functions that all take the same type argument, say *a'*. The function returns a list of functions that all take an *a' list* argument. The example below might help. The list of functions that is returned by *(map map)* is suitable to be used as an argument to the *construction* function discussed earlier in the chapter.

```
- map (map add1);
val it = fn : int list list -> int list list

(map map);
stdIn:63.16-64.10 Warning: type vars not generalized because
of value restriction are instantiated to dummy types
(X1,X2,...)
val it = fn : (?.X1 -> ?.X2) list ->
                      (?.X1 list -> ?.X2 list) list
- fun double x = 2 * x;
val double = fn : int -> int
- val flist = (map map) [add1,double];
val flist = [fn,fn] : (int list -> int list) list
- construction flist [1,2,3];
val it = [[2,3,4],[2,4,6]] : int list list
```

5.26.20 Solution to Practice Problem 5.20

foldl is left-associative and *foldr* is right-associative.

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

5.26.21 Solution to Practice Problem 5.21

```
- List.filter (fn x => x mod 7 = 0) [2,3,7,14,21,25,28];
val it = [7,14,21,28] : int list
- List.filter (fn x => x > 10 orelse x = 0)
           [10, 11, 0, 5, 16, 8];
val it = [11,0,16] : int list
```

5.26.22 Solution to Practice Problem 5.22

```
cpslen [1,2,3] (fn v => v)
= cpslen [2,3] (fn w => ((fn v => v) (1 + w)))
= cpslen [3]
        (fn x => ((fn w => ((fn v => v) (1 + w)))(1 + x)))
= cpslen []
        (fn y => ((fn x => ((fn w => ((fn v => v)
        (1 + w)))(1 + x)))(1 + y)))
= (fn y => ((fn x => ((fn w => ((fn v => v)
        (1 + w)))(1 + x)))(1 + y))) 0
= (fn x => ((fn w => ((fn v => v) (1 + w)))(1 + x))) 1
= (fn w => ((fn v => v) (1 + w))) 2
= (fn v => v) 3
= 3
```

5.26.23 Solution to Practice Problem 5.23

```
datatype bintree = termnode of int
       | binnode of int * bintree * bintree;

val tree = (binnode(5,binnode(3,termnode(4),binnode(8,
            termnode(5),termnode(4))), termnode(4)));

fun depth (termnode _) = 0
  | depth (binnode(_,t1,t2)) = Int.max(depth(t1),depth(t2))+1

fun cpsdepth (termnode _) k = k 0
  | cpsdepth (binnode(_,t1,t2)) k =
      Int.max(cpsdepth t1 (fn v => (k (1 + v))),
              cpsdepth t2 (fn v => (k (1 + v))))
```

5.26.24 Solution to Practice Problem 5.24

```
fun card (Set L) = List.length L;

fun intersect (Set L1) S2 =
      Set ((List.filter (fn x => member x S2) L1))
```