Programming Languages: An Active Learning Approach ©2008, Springer Publishing

Chapter 5 Functional Programming in Standard ML

Kent D. Lee

Functional Programming in Standard ML

- As you might guess by the title, functional programming has something to do with programming with functions.
- However, what the title, **Functional Programming**, doesn't say is what functional programming languages lack.
- Specifically, pure functional languages lack assignment statements and iteration.
- The primary mode of programming in a functional language is through recursion.
- They allow functions to be passed to functions as parameters.
- We say that these functions are higher-order.
- These two features, lack of variables and higher-order functions, drastically change the way in which you think about programming.

Functional Programming in Standard ML

- But why would we want to get rid of variables in a programming language?
- 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.

Functional Programming in Standard ML

Practice 5.1

Is addition commutative in C++, Pascal, or Java? Will `write(a+b)` always produce the same value as `write(b+a)`? Consider the follow Pascal program (or almost a Pascal program, anyway).

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

What does this program produce? What would it produce if the statement were `write(b+a())`?

View Solution

Imperative vs Functional Programming

- Imperative languages are heavily influenced by the von Neumann architecture of computers that includes a store and an instruction counter; the computation model has control structures that iterate over instructions that make incremental modifications of memory.
- The principal operation is the assignment of values to variables.
- 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.
- Functional program execution consists of the evaluation of an expression, and sequential control is replaced by recursion.

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 1930's, before computers existed.
- It is a very small, functional programming language.

The Lambda Calculus

Example 5.1

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

Normal Form

Example 5.2

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*.

Normal Form

Example 5.3

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

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

Normal Form

👉 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 example 5.3. Be sure to parenthesize your expression first so you are sure that you left-associate redexes.

▸ View Solution

Problems with Applicative Order Reduction

👉 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!

▸ View Solution

Getting Started with Standard ML

- 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**.

Getting Started with Standard ML

- There are several implementations of Standard ML.
- Standard ML of New Jersey and Moscow ML are the most complete and certainly the most popular.
- 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.

Getting Started with Standard ML

- 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";
```

Expressions, Types, Structures, and Functions

Example 5.4

Here are some expression evaluations in SML.

```
 - 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;
stdIn:6.1-6.8 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    5 * 3.0
-
```

Expressions, Types, Structures, and Functions

Example 5.5

Here is some code to multiply an integer and a real, producing a real number.

```
- Real.fromInt(5) * 3.0;
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
val it = 15.0 : real
-
```

Expressions, Types, Structures, and Functions

Example 5.6

The signature of the function `fromInt` in the Real structure is

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

The signature of `fromInt` tells us that it 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 makes a `real` number from an `int`.

Expressions, Types, Structures, and Functions

👉 Practice 5.4

Write expressions that compute the values described below. Consult the basis library in appendix **??** 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*.

▸ View Solution

Recursive Functions

Example 5.7

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 recursive function then looks like this:

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

Recursive Functions

👉 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.

▸ View Solution

Recursive Functions

👉 Practice 5.6

The Fibonacci sequence is a sequence of numbers $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. 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 0^{th} element, the second the 1^{st} element and so on. So, $fib(0) = 0$. After arriving at the definition, write a recursive SML function to find the n^{th} element of the sequence.

▸ View Solution

Characters, Strings, and Lists

Example 5.8

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"]
```

Characters, Strings, and Lists

Example 5.9

The signatures of the list constructor and some list functions are given here.

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

Characters, Strings, and Lists

👉 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
```

▸ View Solution

Characters, Strings, and Lists

Example 5.10

Here is a function called implode that takes a list of characters as an argument and returns a string comprised of those characters.

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

So, `implode([#"H",#"e",#"l",#"l",#"o"])` would yield `"Hello"`.

Characters, Strings, and Lists

👉 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?

▸ View Solution

Characters, Strings, and Lists

Example 5.11

Here are a couple more examples of list functions.

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

Characters, Strings, and Lists

👉 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
```

▸ View Solution

Pattern Matching

Example 5.12

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

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

Pattern Matching

👉 Practice 5.10

Rewrite reverse using pattern-matching.

▸ View Solution

Tuples

Example 5.13

`(5,6)` is a two-tuple of `int * int`.
The three tuple `(5,6,"hi")` is of type `int * int * string`.

Tuples

- 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.

Tuples

Example 5.14

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 it will be applied to. In this case append is applied to a tuple of `'a list * 'a list`.

Let Expressions and Scope

Example 5.15

Consider a function that computes the sum of the first `n` integers.

```
1  fun sumupto(0) = 0
2    | sumupto(n) =
3      let val sum = sumupto(n-1)
4      in
5        n + sum
6      end
```

Let Expressions and Scope

- The identifier called `sum` in the example above 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.
- Binding values to identifiers should not be confused with variable assignment.

Let Expressions and Scope

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.

```
1  let val x = 10
2  in
3      (* 1. Value of x here? *)
4      let val x = x+1
5      in
6        (* 2. Value of x here? *)
7        x
8      end;
9      (* 3. Value of x here? *)
10     x
11 end
```

▸ View Solution

Let Expressions and Scope

Example 5.16

The difference between dynamic and static scope can be observed in the following program.

```
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
```

Let Expressions and Scope

- 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 is evaluated in Emacs Lisp it will return a value of 2 from the code in example 5.16.
- It is actually harder to implement static scope than dynamic scope.
- When the function *b* from example 5.16 is executed in a dynamically scoped language, it simply looks in the current environment for the value of *x*.
- Closures are used to represent function values in statically scoped languages where functions may be returned as results and nested functions may be defined.

Datatypes

Example 5.17

In C/C++ we can create an enumerated type by writing

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

This defines a type called TokenType of eleven values: identifier==0, keyword==1, number==2, etc. You can declare a variable of this type as follows:

```
TokenType t = keyword;
```

However, there is 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 can be assigned an integer. This is because the TokenType type is just another name for the integer type in C++. Assigning `t` to 1 doesn't bother C++ in the least. In fact, assigning `t` to 99 wouldn't bother C++ either. In ML, we can't use integers and datatypes 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

Example 5.18

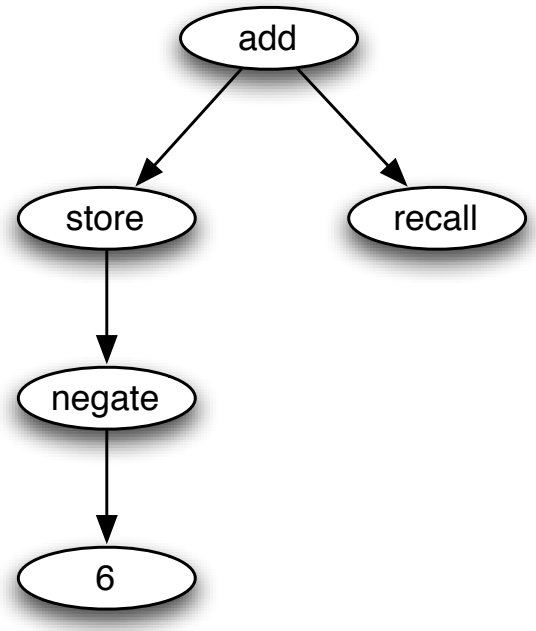
In this datatype the `add'` value can be thought of as a node in an `AST` that has two children, each of which are `AST`s. The datatype is recursive because it is defined in terms of itself.

```
1  datatype
2    AST = add' of AST * AST
3        | sub' of AST * AST
4        | prod' of AST * AST
5        | div' of AST * AST
6        | negate' of AST
7        | integer' of int
8        | store' of AST
9        | recall';
```

Datatypes

Example 5.19

The abstract syntax tree for -6S+R would be as shown below.



Datatypes

Example 5.20

This example illustrates how to use pattern-matching with datatypes and patterns in a `let` construct.

```
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
```

Datatypes

👉 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'`.

▸ View Solution

Datatypes

👉 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 **??** for help with finding the max of two integers.

▸ View Solution

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 section 12.
- If you look at these types in this chapter and in appendix **??** you may notice that there are no functions that modify the existing data.
- All data in Standard ML is immutable.
- Well, almost.
- References may be mutated to enable the programmer to program using the imperative style of programming.

Parameter Passing in Standard ML

- 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.
- Otherwise, the immutability of all data means that how data is passed to a function is irrelevant.

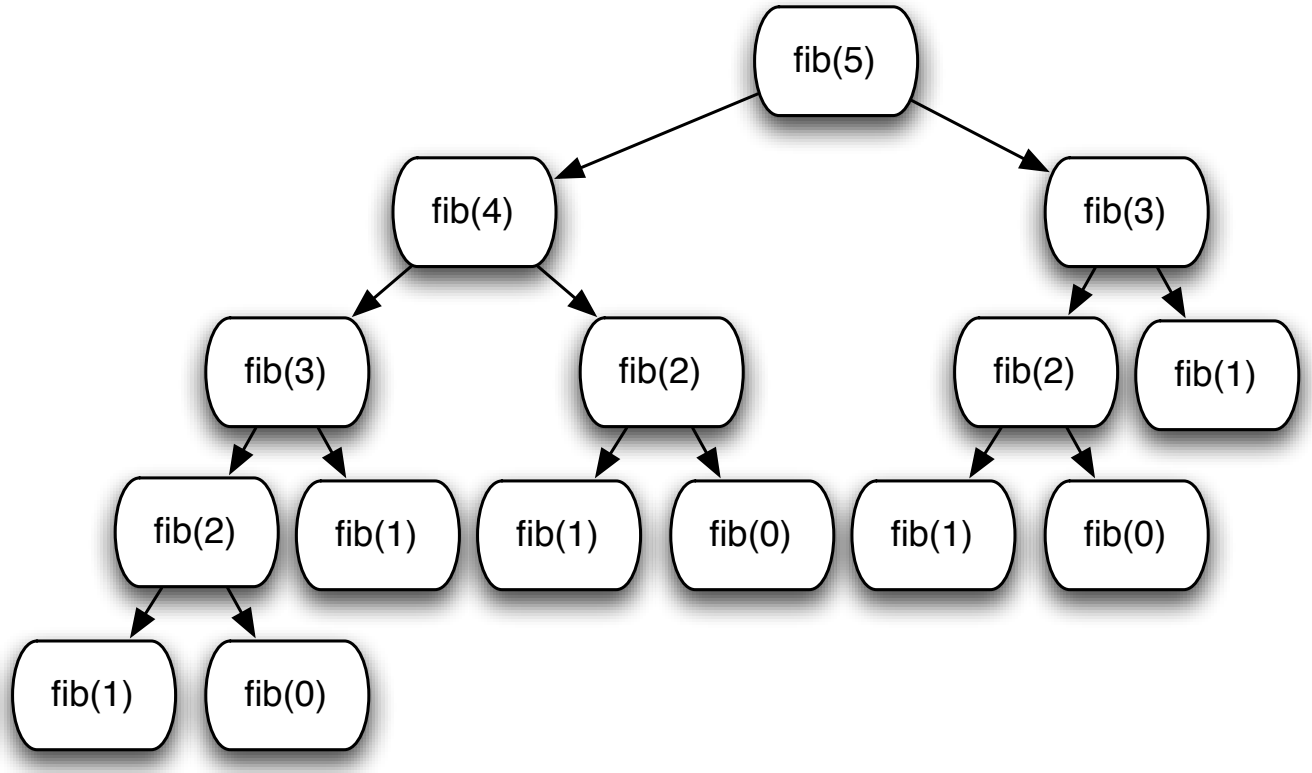
Efficiency of Recursion

Example 5.21

The Fibonacci numbers can be computed with the function definition given below.

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

Calls to calculate fib(5)



Efficiency of Recursion

👉 Practice 5.14

One way of proving that the `fib` function given above 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)`.

▸ View Solution

Efficiency of Recursion

Example 5.22

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.

```
1  fun fib(n) =
2    let fun fibhelper(count,current,previous) =
3        if count = n then previous
4        else fibhelper(count+1,previous+current,current)
5    in
6        fibhelper(0,1,0)
7    end
```

Efficiency of Recursion

👉 Practice 5.15

Consider the reverse function you wrote in practice problem 5.9. 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?

▸ View Solution

Tail Recursion

Example 5.23

This is the `factorial` function.

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

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 above is the multiplication.

Tail Recursion

👉 Practice 5.16

Show the run-time execution stack at the point that factorial 0 is executing when the original call was factorial 6.

▸ View Solution

Tail Recursion

Example 5.24

This is the tail recursive version of the `factorial` function. The tail recursive function is the `tailfac` helper function.

```
1  fun factorial n =
2      let fun tailfac(0,prod) = prod
3            | tailfac(n,prod) = tailfac(n-1,prod*n)
4      in
5        tailfac(n,1)
6      end
```

Tail Recursion

👉 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?

▸ View Solution

Currying

Example 5.25

Here is a function that takes a pair of arguments as its input.

```
- 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. It is applied to 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 takes 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
```

Currying

👉 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.

▸ View Solution

Anonymous Functions

Example 5.26

The anonymous function $\lambda x, y.y^2 + x$ 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
```

Anonymous Functions

Example 5.27

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
```

Higher-Order Functions

Example 5.28

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
```

Higher-Order Functions

Example 5.29

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
```

Composition

Example 5.30

```
- fun compose f g x = f (g x);
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
- fun add1 n = n+1;
- fun sqr n:int = n*n;
- 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 $\circ$ that composes functions. Note that $\circ$ 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
```

Map

Example 5.31

```
- 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
```

`map` can be defined as follows:

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

Map

👉 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

```
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.

▸ View Solution

Reduce or Foldright

Example 5.32

```
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
```

Reduce or Foldright

Example 5.33

This function is sometimes called `foldr`. In this example it is called `reduce`.

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

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

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

```
val sumlist = reduce (op +) 0;
val sumlist = fn : int list -> int
val mullist = reduce 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
```

Reduce or Foldright

Example 5.34

```
foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
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
```

Reduce or Foldright

👉 Practice 5.20

How does `foldl` differ from `foldr`? Determine the difference by looking at the example above. Then, describe the result of these functions invocations.

```
foldr op ::  nil ls
```

```
foldr op @ nil ls
```

▸ View Solution

Filter

Example 5.35

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
```

Filter

👉 Practice 5.21

Use filter to select numbers from a list that are

1. divisible by 7
2. greater than 10 or equal to zero

▸ View Solution

Continuation Passing Style

Example 5.36

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);
```

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.

```
fun cpslen nil k = k 0
  | cpslen (h::t) k = cpslen t (fn v => (k (1 + v)));
```

And here's how cpslen would be called.

```
cpslen [1,2,3] (fn v => v);
```

Continuation Passing Style

👉 Practice 5.22

Trace the execution of cpslen to see how it works and how the continuation is used.

▸ View Solution

Continuation Passing Style

👉 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.

▸ View Solution

Input and Output

Example 5.37

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
```

Input and Output

Example 5.38

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
```

Variable Declarations

Example 5.39

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 a reference points to. This is called the dereference operator. It is the similar to the star (i.e. $*$) in C++.

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

Sequential Execution

Example 5.40

This demonstrates how to write a sequence of expressions.

```
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
```

Evaluating this expression produces the following output.

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

Sequential Execution

Example 5.41

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)
```

Exception Handling

Example 5.42

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.

```
1  exception emptyList;
2
3  fun maxIntList [] = raise emptyList
4    | maxIntList (h::t) = Int.max(h,maxIntList t) handle
5                                  emptyList => h
```

Signatures

Example 5.43

This is the signature of a group of set functions and a set datatype. 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.

```
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
```

Implementing a Signature

Example 5.44

Here is an implementation of the set signature.

```
1  (***** An Implementation of Sets as a SML datatype *****)
2
3  structure Set : SetSig =
4  struct
5
6  exception Choiceset
7  exception Restset
8
9  datatype 'a set = Set of 'a list
10
11 val emptyset = Set []
12
13 fun singleton e = Set [e]
14
15 fun member e (Set [])      = false
16   | member e (Set (h::t)) = (e = h) orelse member e (Set t)
17
18 fun notmember element st = not (member element st)
19
```

```
20  fun union (s1 as Set L1) (s2 as Set L2) =
21      let fun noDup e = notmember e s2
22       in
23         Set ((List.filter noDup L1)@(L2))
24      end
25
26  ...
27  end
```

Implementing a Signature

👉 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.

▸ View Solution

Type Inference

- 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.

Type Inference

- Unification is the process of using type inference rules to bind type variables to values.
- The type inference rules look something like this.
(IF-THEN)

$$\frac{\varepsilon \vdash e_1 : \mathit{bool}, \quad \varepsilon \vdash e_2 : \alpha, \quad \varepsilon \vdash e_3 : \beta, \quad \alpha = \beta}{\varepsilon \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : \alpha}$$

Type Inference

Example 5.45

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.

Type Inference

Example 5.46

In this example the type of the function `g` is inferred.

```
1  fun g h x = if null x then nil
2              else
3                if h (hd x) then g h (tl x)
4                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
```

Exercises

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

Exercises

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.
   2.1 `8 div 3;`
   2.2 `8 mod 3;`
   2.3 `"hi"^"there";`
   2.4 `8 mod 3 = 8 div 3 orelse 4 div 0 = 4;`
   2.5 `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.
   - 9.1 hd
   - 9.2 tl
   - 9.3 explode
   - 9.4 concat
   - 9.5 :: - This is an infix operator. Use the prefix form of `op ::` to get the signature.

10. The greatest common divisor of two numbers, 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 n^{th} 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 `power` 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.

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.

Go Back to Practice Problem

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.

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

▸ Go Back to Practice Problem

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.

Go Back to Practice Problem

Solution to Practice Problem 5.4

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

▸ Go Back to Practice Problem

Solution to Practice Problem 5.5

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

▸ Go Back to Practice Problem

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)
```

▸ Go Back to Practice Problem

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::[]
```

▸ Go Back to Practice Problem

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)))
```

▸ Go Back to Practice Problem

Solution to Practice Problem 5.9

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

▸ Go Back to Practice Problem

Solution to Practice Problem 5.10

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

▸ Go Back to Practice Problem

Solution to Practice Problem 5.11

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

▸ Go Back to Practice Problem

Solution to Practice Problem 5.12

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

▸ Go Back to Practice Problem

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.

▸ Go Back to Practice Problem

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 figure 2 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.

▸ Go Back to Practice Problem

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 $n-1+n-2+n-3+\ldots$ 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 $O(n^2)$.

▸ Go Back to Practice Problem

Solution to Practice Problem 5.16

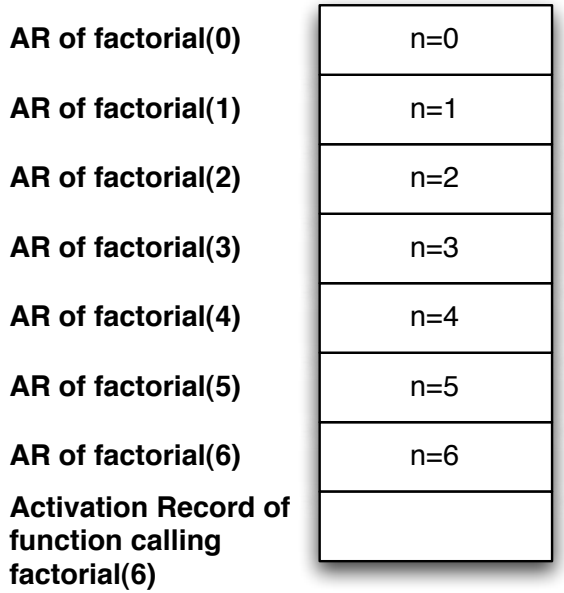


Figure: The run-time stack when factorial(6) is called at its deepest point

▸ Go Back to Practice Problem

Solution to Practice Problem 5.17

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

```
1  fun reverse(L) =
2      let fun helprev (nil, acc) = acc
3            | helprev (h::t, acc) = helprev(t,h::acc)
4      in
5        helprev(L,[])
6      end
```

▸ Go Back to Practice Problem

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
```

▸ Go Back to Practice Problem

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
```

Go Back to Practice Problem

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
```

▸ Go Back to Practice Problem

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
```

▸ Go Back to Practice Problem

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
```

▸ Go Back to Practice Problem

Solution to Practice Problem 5.23

```
1   datatype bintree = termnode of int
2            | binnode of int * bintree * bintree;
3
4   val tree = (binnode(5,binnode(3,termnode(4),binnode(8,
5                termnode(5),termnode(4))), termnode(4)));
6
7   fun depth (termnode _) = 0
8     | depth (binnode(_,t1,t2)) = Int.max(depth(t1),depth(t2))+1
9
10  fun cpsdepth (termnode _) k = k 0
11    | cpsdepth (binnode(_,t1,t2)) k =
12         Int.max(cpsdepth t1 (fn v => (k (1 + v))),
13                 cpsdepth t2 (fn v => (k (1 + v))))
```

▸ Go Back to Practice Problem

Solution to Practice Problem 5.24

```
1  fun card (Set L) = List.length L;
2
3  fun intersect (Set L1) S2 =
4        Set ((List.filter (fn x => member x S2) L1))
```

▸ Go Back to Practice Problem