Computer Science 54

A Brief Introduction to SML

Contents

Introduction

SML stands for "standard meta-language." The language was originally designed for writing programs to carry out logical deduction, but like other programming languages based on carefully chosen principles, it found many other uses. SML and its functional relatives are used for controlling telecommunications systems, robots, and astronomical telescopes; for designing and validating circuits and low-level software; and for sophisticated financial calculations. Google uses a patented method called MapReduce, inspired by functional programming ideas, to distribute huge computational tasks across thousands of computers. Microsoft has developed a language called F# for its .NET framework that is closely related to SML. Your work in this course will give you a sound foundation in fundamental principles that will transfer to all these languages and paradigms.

We assume that you have previously studied Java. SML differs from Java in several important ways.

- It is a *functional language.* The basic mode of computation in SML is to apply functions to arguments, yielding results to which other functions may be applied. Here, we are using the word "function" in the same way as you may have seen it used in a mathematics course. One advantage of functional programming is that it is easy to reason about programs; one can prove rigorously that the functions behave as expected. Another advantage is that opportunities for parallel computation are presented clearly and usefully.

- Data values in SML are *immutable;* once created, they cannot be changed. We make progress in a calculation by applying functions that create new values. In contrast, Java objects contain an internal state—the values of the instance variables—which may change over the course of the computation. There are no variables and no assignment statements in SML. (We do give names to values for convenience, but once named a value does not change.) The *lack of side effects* in SML is what makes it possible to reason rigorously about programs.

- Like Java, SML is a *strongly-typed language.* It is impossible to mistake an integer for another kind of object. Programs written in strongly-typed languages are less likely to have errors and are therefore more reliable. Unlike Java in which one has to declare types, SML has a system of *type inference* which deduces—and enforces—the types of the values. Some programmers find the type restrictions annoying at first, but most people soon welcome the discipline that strong typing brings.

- Functions in SML are *first-class entities,* right up there with integers and strings and boolean values. This feature is jarring to Java programmers who have learned to make a sharp distinction between variables and methods. In SML, a function may be passed as an argument to another function, and the result of a function application may be another function. The concept of functions allows us to work at a higher level of abstraction, benefiting both programmer productivity and program correctness.

- SML has a powerful system for defining *abstract data types.* Functions may be *polymorphic,* meaning that they can take arguments of many different types.
- SML and its associated data structures support and encourage *recursion.* Recursion is a fundamental concept in computer science, and recursive formulations are often clearer and easier to understand. We will use recursion extensively in our work in the course. One feature of SML is *pattern matching* that allows us to create data types and functions in an easy and intuitive way.

We chose SML for this course partly because it encourages clear thinking about data structures and computations and partly because it is new to you and is a useful paradigm. Even if you never write a program in SML after this course, you will use the concepts, ideas, and skills that you develop here.

Using This Document

Learning a programming language is complicated. You must learn the details of the language's syntax—where the semicolons go. You must learn the software that implements the language. You must learn the fundamental concepts underlying the language. And you must learn the idioms associated with writing programs in the language. All of these aspects of a programming language are interrelated, and it is not possible to learn one thing at a time.

This document is divided into four parts which reflect the differing aspects of learning a language. It is unlikely that you will read any one part from start to finish. Rather, you will move back and forth between the parts and make some progress through each one in a single sitting.

Part I, Using the SML System introduces you to the software that you will use in this course, shows you how to submit your assignments, and describes how to interpret SML error messages.

Part II, The SML Language is an informal reference manual for the the language.

Part III, An SML Style Guide discusses how to present your code. It covers indenting, capitalization conventions, commenting, and consistent style.

Part IV, Idioms and Examples presents the fundamental ideas underlying SML—functions, recursion, and lists—and illustrates how they are expressed in the language. There are many examples and practice problems. It is likely that you will spend the largest portion of your time in this part, referring back to the other sections as needed.

Part II

The SML Language

Many introductions to programming languages use a “bottom-up” approach. They start with simple data types and language primitives and show you how to use them to construct larger programs. That may not be the best approach to learning a new language, but it helps to collect the basic facts about a language in one place. Here, we present an informal reference manual for the parts of SML that are important in the course.

1 Data Types

The type `int` The integer data type `int` has constants like `0` and `47`. It has the usual operations `+`, `-`, and `*` for addition, subtraction, and multiplication, respectively. One unusual feature of SML is that it has a special operator for negation; the symbol `~` is used as the *unary minus sign.* One writes `~47` instead of `-47`.

The integer division operations are `div` and `mod`. The former computes the integer quotient and discards any fractional part. The latter gives the remainder; it is the analog of the `%` operator in Java. Be aware that SML, like most programming languages, implements integer division by rounding toward $-\infty$. For negative quotients, that is different from the mathematically more natural notion of rounding toward zero. In all cases, the remainder is computed so that the quotient and remainder satisfy the expected identity.

```
    v * (u div v) + (u mod v) = u
```

The arithmetic operators follow the usual rules of precedence. Unary minus is done first, then multiplication and division, and finally addition and subtraction.

SML values of type `int` admit the usual comparison operators: `=`, `<>`, `<`, `<=`, `>`, and `>=`. Notice that the “equals” operator is a single equals sign, and the “not equals” operator is `<>`. These are *operators* in the strict sense of the word; each one takes two integer arguments and returns a boolean value.

The type `bool` There are two values of type `bool`, `false` and `true`. There is a unary operator, `not`, and two binary operators, `andalso` and `orelse`. The latter operators employ “short-circuit evaluation,” just like their counterparts in Java. If the operator’s value of can be determined from the first argument, the second argument is not evaluated.

The order of precedence for boolean operations is that `not` is done first, then `andalso`, and finally `orelse`.

Boolean values in SML are simply values of type `bool`. There is no cosmic significance to the word `true`. Avoid expressions like `b=true`; the simpler expression `b` is sufficient.

The type `real` Real numbers, called *floating point numbers* in computerspeak, exist in SML, but they are distinct from the integers. Constant expressions *must* have a digit to the right of the decimal point. Thus, `47` is of type `int`, while `47.0` is of type `real`. Values of type `real` have the usual operators `+`, `-`, `~`, `*`, and `/`. They follow the usual rules of precedence, just as for integers.

The real division operator `/` applies only to values of type `real`; it is an error to offer it arguments of type `int`. There are explicit functions `real` and `round` that convert values between `real` and `int`.

Values of type `real` admit the inequality operators `<`, `<=`, `>`, and `>=`, but not `=` and `<>`. The reason for the omission of the equality and inequality operators is that computations of floating point numbers are inexact. Two values that are mathematically identical may not be recognized as being equal by the operator `=`.

The type `char` Individual character constants are written in a strange way. The lowercase version of the last letter of the alphabet is `#"z"`. All the characters are arranged in an ordered sequence. The exact arrangement is usually not important to us, except for the facts that the digits `0` through `9` are adjacent to one another in their usual order. Analogous facts hold for the uppercase letters and the lowercase letters.

The six comparison operators will show the relation of the characters according to the underlying character set. There is a function `ord` that maps characters to their integer position in the character set, starting with zero. There is a function `chr` which takes an integer and returns the character in that place in the character ordering. The standard way to transform a digit into the corresponding `int` value is to use an expression like this: `ord(c) - ord(#"0")`.

The type `string` Strings are essentially sequences of characters and are used more frequently than characters. String constants are written with double quotation marks, as in `"SML"`. The empty string is `""`.

The symbol `^` is used for the string concatenation operator. The six comparison operators operate on strings using the usual lexicographic order inherited from the underlying character set. There are functions `explode` and `implode` that convert a string to a list of characters and vice versa.

2 Constructing New Types

The basic types may be used to create new, sometimes complicated, types. In this section, we review the ways to specify list, tuple, and function types.

List types Lists in SML are recursive. A list is either empty or it has a first element. The elements in a non-empty list after the first element form another list—the *rest* of the list.

The empty list is denoted `[]` or `nil`. The list constructor `::` takes an element and a list and creates a new list with the given element as the first element. The list whose only element is the integer zero is `0::nil`.

There is a convenient notation using square brackets for lists. As we have seen, `[]` is the empty list. The list whose only element is zero can be written `[0]`, which is more convenient than `0::nil`. The expression `[0,1,2]` denotes a three-element list which might otherwise be written `0::1::2::nil`.

Lists in SML are *homogeneous,* meaning that all the elements of a list must be of the same type. The type for lists of integers is written `int list`. More generally, the type `'a list` consists of all the lists whose elements are of some (as yet unspecified) type `'a`.

The operator `@` appends one list to another. Among the built-in list functions are `length`, `rev`, and `map`. We shall see many more list functions in this course.

Tuples Lists in SML are homogeneous—all the elements must be of the same type—and they have variable lengths. Tuples, on the other hand, can have components of different types, but they have a fixed length. Tuples are constructed with parentheses.

The most common kind of tuple is the *ordered pair.* You encountered ordered pairs when you studied analytic geometry. The pair $(4.3, -2.4)$ represents the coordinates of a point in the Euclidean plane. In SML, it would be written `(4.3,~2.4)` and is of type `real * real`. The operator `*`, when used in type specifications, specifies the *Cartesian product* of two types. The general specification `'a * 'b` is the type whose values are the ordered pairs in which the first component is of type `'a` and the second component is of type `'b`.

When we write an expression like `f(x,y)`, we are asserting that `f` is a function whose single argument is an ordered pair. We will also have occasion to write functions whose result is an ordered pair.

There are higher-order tuples as well. An example of a three-tuple is `(47, false, nil)`; its type is `int * bool * 'a list`.

There is no distinct type of one-tuples; we consider the values of a type to be the 1-tuples of that type. There is, however, a zero-tuple. It is written with parentheses, `()`, and is the only value of the type `unit`. One example of the use of the `unit` type is as the return value of the function `use`. That function takes one argument, a file name, and returns `()`. In contrast to most other functions, we execute `use` for the side effect of reading a file into the SML system; we are not interested in the value that `use` returns. We shall see similar uses of the `unit` type when we study lazy structures.

Rarely, it is useful to have a function that extracts one component from a tuple. The function `#1` takes a tuple and returns the first component. Similarly, `#5` would return the fifth component, assuming that the tuple is large enough. Most of the time it is cleaner to use pattern-matching to extract a component from a tuple.

A tuple is actually a special case of a more general type construction called a *record.* We will not encounter records in this course.

Function types We have already mentioned that functions in SML are "first class objects." For example, the `length` function takes a list and returns an integer, namely the number of elements in the list. As an entity, `length` is a function of type `'a list -> int`.

The function `length` is *polymorphic,* meaning that it applies to many different types. In this case, the argument to `length` can be any kind of list. That is the reason for the "placeholder type" `'a`.

The operator `->` specifies a function type. The operator `->` *associates to the right,* meaning that

`'a -> 'b -> 'c` is interpreted as `'a -> ('b -> 'c)`.

A function of that type takes a value of type `'a` and returns a function from `'b` to `'c`. In contrast, the signature `('a -> 'b) -> 'c` specifies an entirely different type; it takes a function from `'a` to `'b` and returns an element of type `'c`.

3 Expressions and Values

The syntax of SML is simple and natural, and it emphasizes the underlying meaning of the code. The fundamental syntactic entity is the *expression,* which specifies a *value.*

Expressions An expression is composed of identifiers, constants, and operators. Parentheses may be used for grouping. Each of the components of an expression has a type, and the operators must be used in ways that are consistent with the types of their operators. Later in the course, we will study a more precise way of specifying the syntax of expressions. Here are some legal expressions; the corresponding values are obvious.

```
50-3
(2+2)*10+(9-2)
round(141.0/2.0)
round(141.0) div 2
implode [chr 52, chr 55]
```

Each expression specifies a computation which produces a *value.* In a sense, that value is the meaning of the expression. The value of an expression has an unambiguous type. The notation for that type is the value's *type signature.* Type signatures are constructed from the basic types, like `int` and `string`, using the type constructors, like `*` and `list` and `->`. Here are some sample type signatures.

```
bool
int * bool
real * real -> int -> real
int list -> int
```

Conditional expressions A conditional expression is constructed from the keywords `if`, `then`, and `else`. The structure is as follows.

```
if ⟨boolean-expression⟩
   then ⟨expression⟩
   else ⟨expression⟩
```

In SML, the `else`-part is *not* optional, and the `then`- and `else`-expressions must be of the same type. The result of the SML conditional expression is a *value,* not an action as it is in Java and other languages. The Java analog to the SML conditional expression is the ternary operator (`? :`), not Java's `if-then-else`.

Value declarations A value declaration begins with the keyword `val`. It is followed by an identifier, an equals sign, an expression, and finally a semicolon. (Strictly speaking, the semicolon is optional in many circumstances, but it does not hurt to include it.)

```
val pi = 3.14159;
```

After a value declaration, the identifier is associated with the value that has just been computed. The identifier is a name for the newly-computed value.

Value declarations are different from assignment statements in Java and other languages. For example, consider the following block of SML code. After the block has been executed, what is the value of `addY 7`?

```
val y = 3;
fun addY x = x + y;
val y = 0;
```

If value declarations were like assignment statements, the result of `addY 7` would be `7`. But in SML, the result is `10`. When the function `addY` is declared, it picks up the *value* of `y`, not the *variable* `y`.

Value declarations are actually more general than just described. A more complicated structure may appear in place of the identifier. Value declarations use the same kind of pattern-matching as function declarations.

```
- val x::xs = [1,2,3];
val x = 1 : int
val xs = [2,3] : int list
```

Function declarations As previously mentioned, a function is merely a particular kind of value. However, creating a function with a value declaration is convoluted. (We shall see how to do it later.) Fortunately, the designers of SML have given us a more convenient and natural syntax.

A function declaration begins with the keyword `fun`, has one or more cases separated by vertical bars, and is terminated with a semicolon. Each case consists of the function name, an argument list, equals sign, and an expression. Across the cases, the function names must be the same; the argument lists must have the same number of arguments, and the types must match; and the expressions must all have the same type. Here is the declaration of a function `append` which joins two lists in the same way as the operator `@`. The function is recursive, and its declaration follows the usual pattern for list recursion.

```
fun append nil     v = v
  | append (u::us) v = u::(append us v);
```

4 Information Hiding

The object-oriented paradigm of Java makes it easy to combine related values into a “package” and to hide some of the internal workings. There are similar facilities in SML. In this section, we look at two forms of packaging and some more advanced topics relating to functions.

Temporary values Sometimes it is convenient to give a name to a value just to use it in an expression. The `let` construction was designed to do just that.

```
let
    ⟨declarations⟩
in
    ⟨expression⟩
end
```

Any number of value and function declarations may appear, and their identifiers may be used in the expression. The type of the `let` expression is the type of the expression between `in` and `end`. Here is a function that we will encounter in Section 4 of Part IV.

```
fun ourRev1 lst =
    let
        fun revAux acc nil     = acc
          | revAux acc (x::xs) = revAux (x::acc) xs;
    in
        revAux nil lst
    end;
```

Controlling visibility Another way to package details of an implementation is with the *local*-construction.

```
local
    ⟨declarations⟩
in
    ⟨declarations⟩
end;
```

The declarations in the first part can be used in the declarations of the second part, but only the declarations of the second part are visible in the rest of the program. (Think of private methods in Java.) Here is the `local` version of our list-reversing function.

```
local
    fun revAux acc nil     = acc
      | revAux acc (x::xs) = revAux (x::acc) xs;
in
    fun ourRev1 lst = revAux nil lst;
end;
```

The choice between `let`- and `local`-constructions is largely a matter of taste. The `let`-construction is an expression and can be used inside other expressions, and for that reason, it will be more common in our work. On the other hand, the `local`-construction allows several declarations to share the same "local" information.

5 More on Functions

There are some convenient (some people might say "essential") constructions for creating and manipulating functions.

Anonymous functions A function is, as we have seen, a kind of value. And like other values, a function need not have a name. There are situations in which we want to define a function—usually as an argument to another function—without naming it. The syntax for doing that uses the keyword `fn`. Here is an expression for a function which squares integers.

```
fn n => n * n
```

The value of this expression is the function that takes an integer `n` and produces its square. If we wanted to square every element of a list of integers, we could use the expression as an argument to the `map` function.

```
map (fn n => n * n) someList
```

Anonymous functions are usually simple, but they can be defined with multiple cases. Our function declaration is really an abbreviation for a value declaration in which the resulting value is a function. For example, here are two equivalent definitions of the function `square`.

```
fun square n = n * n;
val square = fn n => n * n;
```

It is possible to define recursive functions with multiple curried arguments as value declarations, but we leave the details for another time.

Composition You probably encountered function composition in a mathematics course. If f and g are functions, then $f \circ g$ is another function, defined by

$$(f \circ g)(x) = f(g(x)).$$

In SML, we might write the following function declaration.

```
fun compose f g = fn x => f(g x);
```

Actually, the composition operator is built into SML; it is `o`, the lowercase letter "oh."

6 The Basis Library

Like most modern programming languages, SML has a large collection of pre-declared datatypes and functions, organized into components called *structures.* You may already have seen several of the functions.

```
List.filter
ListPair.zip
Int.max
Int.sign
Math.sqrt
```

In addition to "utility functions" like the examples above, there are functions for accessing services of the underlying operating system, including time and date, files, and the network.

Take some time to browse the documentation at `http://www.standardml.org/Basis/manpages.html`. The structures `Int`, `List`, `ListPair`, `Math`, and `String` may be of the most immediate interest at this point in the course.

7 Exceptions and Options

A large portion of the computer code that has ever been written is "defensive." It is concerned with handling cases in which something has gone wrong: Perhaps the user has entered an incorrect value, the end of a file has been encountered prematurely, or a request for service has timed out. The idea of an *exception* makes this kind of programming easier.

An exception is a signal that something unexpected has happened. You may have encountered exceptions in Java, where exceptions are "thrown" and "caught." In SML, the ideas are much the same but the words are different; exceptions in SML are "raised" and "handled."

Declaring exceptions Some exceptions are built into SML. For example, the exception Div is raised on an attempt to divide by zero.

```
- 3 div 0;
uncaught exception Div [divide by zero]
  raised at: stdIn:20.3-20.6
```

The built-in function hd returns the head of a list. (We have not mentioned it before because pattern-matching is a safer and usually more transparent alternative.) Obviously, hd may be applied only to a non-empty list. If it is applied to an empty list, the exception Empty is raised.

```
- hd nil;
uncaught exception Empty
  raised at: smlnj/init/pervasive.sml:209.19-209.24
```

We can also define our own exceptions using the keyword exception.

```
exception SomethingBad;
exception SomethingWorse of string;
```

In the second case, the exception SomethingWorse carries along with it a string, presumably a message describing what went wrong.

Raising exceptions We signal an exception with the keyword raise. For example, suppose that we want to "pair up" the elements in two lists, and we insist that the two lists have the same length. Encountering lists of different lengths is an exception.

```
exception UnequalLengths;

fun ourZip nil     nil     = nil
  | ourZip (x::xs) (y::ys) = (x,y) :: (ourZip xs ys)
  | ourZip _       _       = raise UnequalLengths;
```

The third case, in which we raise the exception, will occur only when one list is empty and the other is not.

```
- ourZip [1] [2,3];
uncaught exception UnequalLengths
  raised at: stdIn:26.36-26.50
```

(We note in passing that there is a built-in library function ListPair.zip which does almost the same thing as ourZip. The differences are that ListPair.zip is uncurried and takes an ordered pair, and that ListPair.zip ignores any extra elements at the end of one of the lists.)

Handling exceptions In all of our examples of the use of exceptions, the exceptions are “uncaught.” (That is the word that SML uses; it should actually be “unhandled.”) When an exception is raised, it is passed up the chain of pending function calls until there is an expression with the keyword `handle` for that particular exception. If there is no `handle` clause, the exception emerges at the top level, the computation stops, and a message is printed. We will discuss writing explicit handlers in Section 9.

Option types The option type is an alternative to one kind of exception. Suppose that we ask a function to compute a value and there is no such value. It could be that we asked for the square root of a negative number, or we asked for an element of a list that was not present. The keyword `option` gives us a way of saying “no answer.” An `option` value is either `NONE` or `SOME value`.

For example, suppose that we want to know the place where a particular element occurs in a list.

```
fun posOfAux _ _ nil      = NONE
  | posOfAux k e (x::xs) = if e = x
                              then SOME k
                              else posOfAux (k+1) e xs;
fun posOf e lst = posOfAux 0 e lst;
```

With these declarations, we can compute positions.

```
- posOf 3 [1,2,3,4];
val it = SOME 2 : int option
- posOf 0 [1,2,3,4];
val it = NONE : int option
```

One cost of using the option type is that the receiver of an option value must decode it. A robust program will have branches for both `NONE` and `SOME`. Also, if there is a result, it must be removed from the `SOME`-expression. The `case` statement, described in Section 8, provides a convenient way to handle results of type `option`.

One useful built-in function is `valOf` which behaves as if it were declared as follows.

```
fun valOf NONE     = raise Option
  | valOf (SOME k) = k;
```

The use of `valOf` is limited by the fact that one must already know that the argument is not `NONE`.

8 More on Patterns

Here we present a few syntactic constructions that extend and facilitate pattern-matching.

Case expressions We can take advantage of pattern-matching within arbitrary expressions, not just in function declarations. For an example, recall the function `posOf` which we discussed when studying the `option` type. The function `posOf` takes an element and a list and returns a result of type `int option`. That is, it returns either `NONE` or `SOME j`, where j is of type `int`. A case expression gives us a natural way to handle the two kinds of values returned by `posOf`, as the following expression illustrates.

```
case posOf elt someList of
     NONE    => expression for the NONE case
   | SOME v => expression involving v
```

Here is the syntax for the `case` expression.

```
case ⟨expression⟩ of
     ⟨pattern1⟩ => ⟨value1⟩
   | ⟨pattern2⟩ => ⟨value2⟩
   | ...
```

Just as in a function declaration, the patterns in a `case` expression must cover all of the possible forms of the given expression. The result of the `case` expression is the value that corresponds to the first-matched pattern.

The case expression is a very general construct. The `if-then-else` expression is really an abbreviation for a case expression.

```
case booleanExpression of
     false => falseExpression
   | true  => trueExpression
```

Further, pattern matching in functions is reducible to a case expression. Consider this declaration for a function that tells us whether a list is empty. (The function `null` is actually built-in to SML.)

```
fun null aList =
    case aList of
         nil     => true
       | (x::xs) => false;
```

Our former syntax is clearer and easier to use, but it adds no fundamental power to the language.

Syntactical detail: When case expressions are nested, or when case expressions are combined with pattern-matching in function declarations, the vertical bar is sometimes ambiguous—the system does not know to which expression the bar applies. In those cases, it is necessary to enclose the inner case expression in parentheses.

Anonymous variables Sometimes we do not care about naming a variable. Consider the declaration of the `map` function.

```
fun map f nil     = nil
  | map f (x::xs) = (f x) :: (map f xs);
```

The argument `f` in the first line is unnecessary; it is never used. Although we do not have to name it, we do have to reserve a position for it in the argument list. SML uses the underscore character _ as a place holder or *anonymous variable.*

```
fun map _ nil     = nil
  | map f (x::xs) = (f x) :: (map f xs);
```

Anonymous variables are frequently seen in pattern-matching. Here is a variation of the function `null` which we used to illustrate the `case` expression.

```
fun null nil    = true
  | null (_::_) = false;
```

Anonymous variables can also appear in value declarations. Suppose we have a function that returns an ordered pair, but we only care about the first component. We could write the following declaration.

```
val (first,_) = pairProducer arguments;
```

In a strict sense, anonymous variables are unnecessary, but they do make code more readable.

9 More on Expressions

An expression is a sequence of symbols that satisfies certain syntactic specifications. An expression produces a value. Here is an incomplete list of expressions.

- A constant term, like `32` or `true`, is an expression.
- A single identifier, like `myFunction` or `k`, is an expression.
- An expression may be formed by putting two expressions next to one another, to signify function application.

- An expression may be formed by putting an infix operator, like + or `::` or `orelse`, between two expressions.

- Expressions may be formed with the keywords `if-then-else`, `case-of`, `let-in-end`, and `raise`.

- Expressions denoting anonymous functions are formed with the keyword `fn`.

There are, of course, conditions on the component expressions. The two parts of an infix expression, for example, must have values whose types are appropriate for the infix operator. The expression after the keyword `raise` must evaluate to an exception. In a programming languages course you will learn about syntactical specifications and about the orthogonal conditions imposed by the type system.

One kind of expression that is not in the above list involves the keyword `handle`. The syntax is simple; here is an example.

```
hd ⟨list-expression⟩
   handle Empty  => someDefaultValue
        | Div    => anotherDefaultValue
        | Option => aThirdDefaultValue
```

If the ⟨list-expression⟩ evaluates to a non-empty list, then the first element of that list is the result of our expression, and the `handle` part contributes nothing. If the ⟨list-expression⟩ evaluates to `nil`, then the application of `hd` raises the exception `Empty`, and our expression returns `someDefaultValue`. If evaluating ⟨list-expression⟩ raises `Div` or `Option`, then our expression will return one of the other default values.

To pass type-checking, all of the default values (the expressions after the symbol =>) must be of the same type as the expression before `handle`. Any expression of the appropriate type may appear after the symbol =>; one can even raise another expression there.

When an exception is raised, computation stops. If the exception can be handled locally, the expression returns a value to the surrounding environment and computation continues. If the exception cannot be handled locally, it is passed to the next larger expression and is either handled there or passed on further. If the exception ever reaches the outermost level, the computation is aborted and we see the familiar "uncaught exception" message.

The list of exceptions in a `handle` clause may be incomplete. An exception that does not match any element of the list is passed on to the surrounding environment. If we want to handle *all* exceptions, we can use an anonymous variable, which matches any exception.

```
... handle Empty => specificDefaultValue
           _     => genericDefaultValue
```

Exceptions may be declared to carry extra information. For example, the exception Error might contain a string with a description of what went wrong.

```
exception Error of string;

⟨string-expression⟩
    handle Error msg => msg
         | _         => "unknown error"
```

The examples in this section are intended to illustrate how exception-handling operates; they are not examples of good programming practice. In particular, it is dangerous to return a value when an error has occurred unless that value is truly "correct" for the surrounding program.

In contrast to the general principle of separating the "returned value" from a "status message," the type `real` contains two values that signal errors. These values exist in an attempt to avoid raising exceptions in real arithmetic. The values are `inf`, "infinity," and `nan`, "not a number."

```
- 4.7/0.0;
val it = inf : real
- 4.7/0.0*(~1.0);
val it = ~inf : real
- 4.7/0.0*0.0;
val it = nan : real
- 0.0/0.0;
val it = nan : real
- 4.7/0.0-4.7/0.0;
val it = nan : real
```

Part III

An SML Style Guide

Like English prose, computer programs can be presented clearly and elegantly. Part of the course is to develop your judgment and teach you how to write clean, crisp, and correct code. The guidelines here, although by no means complete or authoritative, give you a place to start.

1 Absolutes

- The code in the files you submit must compile without errors or warnings other than the warning “calling polyEqual”. No partial credit will be given for code that does not compile.
- Be sure that your code conforms exactly to the specifications in an assignment. In particular, the identifiers must be spelled correctly and the functions must have the correct types.
- Use spaces, not tab characters. (This is to help provide a uniform view of a file, regardless of where it is viewed or printed. The `sml-mode` in `emacs` on the Computer Science systems will insert spaces when you press the tab key.)
- No line may exceed 80 characters.

If for some reason you develop programs on another platform (in particular, if you import files from MS Windows), then you should explicitly convert the file to the proper format. The utility program

```
/common/cs/cs052/bin/formatCheck
```

will verify (and, with the `-f` option, attempt to repair) the format of a file. Be aware that long lines will be broken to obey the 80-character limit, which may cause your program to fail. Given the `-h` option, `formatCheck` will print a short description of itself.

2 General Points

- Be consistent. You are encouraged to experiment with different formats at different times, but do not do it in a single assignment.

- Pay attention to style from the time you begin to write, and use it as an aid in structuring your code. Contrary to your feelings as time runs short, it is never expedient to write sloppy code and "fix it" later.

- Use blank lines to separate logically separate parts of a file. For example, two different function declarations should be separated by a blank line. Conversely, do not insert blank lines where they would separate closely related parts of your code.

3 Comments

- Always start a file with a comment, indicating what it is, who wrote it, when, and why. See Table 2 for an example.

- Comments should go above the code, or in the case of very short comments, to the right.

- Avoid useless comments, like the one below.

```
    n + 1      (*  add one to n  *)
```

- Avoid over-commenting. We will discuss this topic further in class. There are (at least) two competing philosophies: One, like Java's `javadoc` system, says that every publicly-accessible function should have a comment indicating its type, its semantics, and a description of its use. Another says that most of that information ought to be embedded in the identifier names and the code structure.

- When multi-line comments are necessary, use the format of the comment in Table 2.

- Indent comments by the same amount as the code.

4 Names

- Use descriptive names. For the most part, one character variable names are appropriate only within a function definition or a block.

- Use the standard convention for upper and lowercase letters, as in Table 3. In this course, we may not cover all the language features listed in the table.

```
(*
 *  producer.sml
 *
 *  Rett Bull
 *  August 1, 2003
 *  CS 52, Assignment -3, Version 2
 *
 *  Provides an abstraction for an "input stream." A
 *  producer is an entity that supplies a sequence of
 *  values, using a hidden state--and therefore
 *  violating the functional paradigm. This code was
 *  written as an experiment with SML references.
 *
 *  The following functions are implemented:
 *
 *   - isValid  : 'a producer -> bool, tells if there is
 *                an element at the front of the stream.
 *   - isActive : 'a producer -> bool, tells if the stream
 *                has not terminated.  (Note isActive
 *                being false implies isValid is also
 *                false.)
 *   - current  : 'a producer -> 'a, gives the current
 *                element in the stream.
 *   - advance  : 'a producer -> unit, advances the stream
 *               one element.  Executed only for the
 *               side-effect.
 *   - create   : ('s -> 's option) ->
 *                   ('s -> 'a) -> 's -> 'a producer,
 *               creates a producer from three items:
 *                   + a next-state function,
 *                     's -> 's option, to advance to the
 *                     next state (where NONE signifies
 *                     that there is no next state and
 *                     the producer is finished);
 *                   + a value function, 's -> 'a, to
 *                     extract the next value from the
 *                     state; and
 *                   + an initial state.
 *)
```

Table 2: A sample header comment.

identifier	style	example
Variables	initial lower case letter; embedded upper case for separate words	`nodeCount`
Constructors	initial upper case letter; embedded upper case for separate words	`TreeNode`
Exceptions	just like constructors	`BadInputData`
Types	all lower case, with underscore as a separator	`binary_tree`
Signatures	all upper case, with underscore as a separator	`BINARY_TREE`
Structures	initial upper case letter; embedded upper case for separate words	`BinaryTree`
Functors	like structures, with `Fn` appended	`BinaryTreeFn`

Table 3: Capitalization conventions in SML.

5 Indentation

- Choose a standard number of spaces for indentation and stick to it. Most programmers use between two and eight spaces.
- Professor Bull prefers the following indentation scheme, which emphasizes vertical alignment. The corresponding arguments to a function appear in a single column, and the `then`-expression is directly over the correxponding `else`-expression. See the textbooks and references on the web for other schemes. Again, consistency within a file is important.

```
fun f nil     nil = ...
  | f (x::xs) nil = ...

if bool1
   then expn1
else if bool2
   then expn2
   else expn3

case expn of
     pat1 => ...
   | pat2 => ...

val bigSum = longExpression +
             anotherLongExpression
```

```
fun longFunctionName lotsOfArguments =
    expressionTooLongtoFitAbove

fun myReverse u =
    let
        fun myRev nil     y = y
          | myRev (x::xs) y = myRev xs (x::y)
    in
        myRev u nil
    end
```

6 Parentheses

- Avoid extraneous parentheses. Parentheses are used differently in SML than in Java or other languages. The expression `f x` is usually preferable to `f(x)`.

- Use parentheses to disambiguate nested blocks involving `case` or `if-then-else` constructions. (Almost always, there is a way to avoid deep nesting of `if-then-else`. Look for it.)

- Use parentheses to emphasize the logical structure. Although the parentheses in the example below are not required by the language, they show the structure of the third argument to `f`.

  ```
  f u v (if boolTest then w else x)
  ```

7 Pattern Matching

- Avoid incomplete pattern matchings. These will generate warnings, meaning that your work will get no credit. Use an exception if you believe (if you *really* believe and can prove it!) that you have covered all the cases that will actually occur.

  ```
  exception ThisCannotHappen;

  fun f pattern1 = ...
    | f pattern2 = ...
  ...
    | f _        = raise ThisCannotHappen;
  ```

8 Verbosity and `let` Expressions

- In general, organize your code so that the logical structure is clear and simple, but not fragmented. The `let` expression is one example of a construction that can be used to clarify a block of code—or overused to muddle it.

- Use `let` expressions when it improves efficiency. For example, the function below is inefficient because it makes two recursive calls.

  ```
  fun uniquify nil = nil
    | uniquify (x::xs) = if member x (uniquify xs)
                            then uniquify xs
                            else x::(uniquify xs);
  ```

  Using a `let` expression is much better.

  ```
  fun uniquify nil     = nil
    | uniquify (x::xs) =
      let
          val recResult = uniquify xs
      in
          if member x recResult
             then recResult
             else x::recResult
      end;
  ```

- Use `let` expressions to "protect" auxilliary functions from the surrounding functions, as in the example below.

  ```
  fun myReverse u =
      let
          fun myRev nil     y = y
            | myRev (x::xs) y = myRev xs (x::y)
      in
          myRev u nil
      end;
  ```

- Use `let` expressions to improve readability. Compare the two declarations of `removeNegatives`:

  ```
  fun removeNegatives nil     = nil
    | removeNegatives (x::xs) =
      if x < 0 then removeNegatives xs
               else x::(removeNegatives xs);

  fun removeNegatives nil     = nil
    | removeNegatives (x::xs) =
  ```

```
let
    val isNeg = x < 0;
    val recResult = removeNegatives xs;
in
    if isNeg then recResult else x::recResult
end;
```

The two declarations are operationally identical, and we could argue about which is more readable. Although it is a judgment call in this case, the choice is often clear. As the expressions within the `let` become more complicated, the second form becomes preferable.

9 Verbosity and Booleans

- Remember that `if-then-else` returns a value, and never write

  ```
  if expr then true else false
  ```

  That code can always be replaced simply by `expr`.

- Never write expressions like `if cond = true` or `if cond = false`. They can be replaced by `if cond` and `if not cond`.

- Try to avoid repeating sub-expressions. For example, the expression

  ```
  if boolTest
     then f u v w
     else f u v x
  ```

  can be written more clearly and simply as

  ```
  f u v (if boolTest then w else x)
  ```

  The parentheses are not necessary, but they clarify the structure.

Part IV

Idioms and Examples

The *syntax* of a programming language specifies how the language is written—the order of symbols and where to put braces and semicolons. It is essential that a programming language have an unambiguous syntax; otherwise we could not distinguish between correct and incorrect programs. On the other hand, learning the syntax of a programming language is the easiest and least important part. What we really care about is the *semantics* of the language—what the programs do—and the *idioms* of the language—the patterns that programmers have developed to use the language to its fullest extent.

1 Getting Started

The best way to learn SML is to sit down at a computer and experiment with it. Work through the examples and practice problems of this section. Invent your own variations.

Let us begin by looking at a simple SML function, one that takes a pair of integers and returns their difference.

```
fun difference (a,b) = a - b;
```

The keyword *fun* indicates that we are defining a function. The name of the function is *difference*. It takes a pair as an argument and (after the equals sign) returns the result of a subtraction. The semicolon concludes the declaration.

If we were to type that declaration into the SML system, we would see the response below. (Recall our the convention that input from the user is shown in purple and the system's response is in green. The hyphen is the prompt indicating that SML is ready for input.)

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

The response says that we have created a *value* named `difference` which is a function that takes a pair of integers and produces an integer. We will discuss how the system deduces the types a little later. Notice that the result of our declaration is a *value* which happens to be a function.

Now that we have a function, we can apply it.

```
- difference(94,52);
val it = 42 : int
```

The system applied the function to the argument and obtained a value, indicated by the keyword `val`, named `it` which happens to be the integer `42`.

Had we wanted to, we could have given the resulting integer a name.

```
- val fourtytwo = difference(94,42);
val fourtytwo = 42 : int
```

Everything in SML is a value. A function is simply a special kind of value.

Practice Problem 1. For each expression below, write an equivalent one that is simpler.

a. `a andalso not a`

b. `a orelse (not a andalso b)`

c.
```
(not a orelse b) andalso
(not b orelse c) andalso
(not c orelse not a) andalso
(not c orelse not b)
```

Practice Problem 2. Some of the expressions below contain errors. For each one, diagnose the error, or if there is no error, give the result evaluating the expression.

a. `3+4*11`

b. `3/4.0`

c. `21 div 2 mod 5`

d. `1-2-3-4`

e. `2.0<3.0 orelse 7<=5`

f. `if 2.0<=3.5 then 6+11`

Practice Problem 3. In Section 1 of Part II, we saw how to turn a *digit* (a character between `#"0"` and `#"9"`) into the corresponding *integer*. Write a simple function `intToDigit` that does the opposite.

Practice Problem 4. Write a function `isEven` that returns true exactly when its argument is an even integer.

Practice Problem 5. Write a function `circleArea` to compute the area of a circle of a given radius. Make sure that the radius and the area are of type `real`. The value of π is available as `Math.pi`.

Practice Problem 6. Give the type signatures of the following functions.

a. `fun f(a,b,c) = if a<b then c else false;`

b. `fun g(a,b,c,d) = if a<b then c else d;`

c. `fun h(a,b,c,d,e) = if a<b then c+d else e;`

d. `fun i(a,b,c,d) = if a<b then c+1.0 else d;`

---

2 Curried Functions

A careful reader will have observed that we said that the function `difference` is applied to the "argument," not "arguments." The function `difference` is *not* a function of two variables, as you might naturally have assumed. Rather, it is a function of one variable, a variable that happens to be a pair of type `int * int`.

All functions in SML are functions of one variable, a fact that reflects a very powerful idea. Let us write the difference function in another way.

```
- fun curriedDifference a b = a - b;
val curriedDifference = fn : int -> int -> int
```

The only difference between this function and the original is that we have listed the arguments `a` and `b` separately. If we apply the new function, we obtain the expected result.

```
- curriedDifference 94 52;
val it = 42 : int
```

But what kind of creature is `curriedDifference` with a type signature of `int -> int -> int`? Let us try an experiment.

```
- curriedDifference 94;
val it = fn : int -> int
```

Applying `curriedDifference` to a single argument produced a function—one that is ready to receive another integer argument. The type signature `int -> int -> int` is interpreted as `int -> (int -> int)`. The function `curriedDifference` takes an integer and produces a function of type `int -> int`. The resulting function is ready to take its second argument.

We say that our original difference function that takes a pair is in *uncurried* form, while our new version is *curried*. The name is in honor of the mathematical logician Haskell B. Curry; it has nothing to do with tasty food.

The distinction between uncurried and curried functions may appear at first to be petty and arcane. After all, both functions do the same thing. However, as we will soon see, curried functions are exceedingly useful. We can think of `curried-Difference` as a "factory" that takes an integer like `94` and produces a function that transforms a value `n` into `94 - n`. That resulting function is available to be an argument to still another function.

Practice Problem 7. The following functions are curried versions of the ones you encountered in Practice Problem 6. Give the type signatures for each one.

a. `fun f a b c = if a<b then c else false;`

b. `fun g a b c d = if a<b then c else d;`

c. `fun h a b c d e = if a<b then c+d else e;`

d. `fun i a b c d = if a<b then c+1.0 else d;`

---

3 Lists

Lists are a basic data type in SML. The list `[2,4,6,8]` has four elements, each of which is an integer. In SML, lists are *homogeneous;* all the elements of a list must be of the same type.

```
- [2,4,6,8]
val it = [2,4,6,8] : int list
```

The system's response indicates that the value is a list of integers.

Lists are defined recursively. The empty list is `[]` or `nil`. Any other list has a *first* element; when we remove the first element, we have the *rest* of the list—which is another list. A one-element list has a value as its first element, and the rest of the list is empty. To add a new element to a list, we use the operator `::`.

```
- 0::[2,4,6,8]
val it = [0,2,4,6,8] : int list
```

Notice the asymmetry; the value to the left of `::` is an *element,* while the value to the right is a *list.* The square bracket notation is simply a convenience. The expression `[2,4,6,8]` is an abbreviation for the more cumbersome notation

```
2::(4::(6::(8::nil)))
```

Strictly speaking, the parentheses are unnecessary. We say that the :: operator "associates to the right." Most other operators, like addition and subtraction, associate to the left.

We can write a recursive function to compute the length of a list. The length of the empty list is zero, and the length of a non-empty list is one more than the length of the rest of the list. We named the function ourLength to avoid confusion with the built-in SML function length.

```
- fun ourLength nil     = 0
    | ourLength (x::xs) = 1 + (ourlength xs);
val ourLength = fn : 'a list -> int
```

There are several features of this definition to notice. The first is that it employs *pattern-matching.* There are two cases, or "patterns." In applying the function, the SML system tries to match the actual argument with each pattern, in order from the top down. If the first pattern matches, if the argument is the empty list in this case, the result is 0. Otherwise, the second pattern matches, and the result involves a recursive call.

The type signature 'a list -> int indicates that our length function is *polymorphic.* The expression 'a is a type variable; it can be any type. The function takes any kind of list and produces an integer.

There are several syntactic details to observe in our example. The different patterns are separated by the vertical bar |, and the semicolon does not appear until after the last pattern. Diverging from the usual mathematical notation for functions, we can write ourLength xs without parentheses around the argument. The other parentheses are necessary, however. The parentheses around (x::xs) are needed to show that the pattern is a single argument to the function. The parentheses around (ourLength xs) are needed because the SML system reads from left to right. If the parentheses were omitted the system would complain that we were asking it to add 1 to ourLength, and it is impossible to add an integer to a function.

The use of x::xs is common, primarily because xs is read "excess" and indicates the rest of the list.

4 List Recursion

Lists are defined recursively, and functions on list can follow that definition. It is a pattern that you will use frequently. For example, suppose that we want to square every element in a list of integers.

```
- fun square y = y * y;
```

```
val square = fn : int -> int
- fun squareAll nil     = nil
    | squareAll (x::xs) = (square x) :: (squareAll xs);
val squareAll = fn : int list -> int list
```

The function squareAll squares all the elements in a list of integers. It takes a list of integers and produces a list of integers (of the same length).

The pattern in the declaration of squareAll is the standard one for list recursion. There are two cases, one for the empty list and one for other lists.

Practice Problem 8. Write a function doubleAll that doubles every element in a list of integers.

Practice Problem 9. Write a function dup that takes a list and duplicates each element. For example, dup [0,1,2] returns [0,0,1,1,2,2].

Practice Problem 10. Write a function undup that removes consecutive repetitions from a list. For example, undup [0,0,1,1,1,0] returns [0,1,0]. (Hint: This function illustrates a variation on the standard pattern; there are two base cases.)

The pattern we used to declare squareAll is common, and we can generalize. The idea is to apply a function to every element of a list and obtain a list of the results. In other words, we want to "map" the function across the list. There is a built-in function map that behaves in the same way as our example.

```
- fun ourMap f nil     = nil
    | ourMap f (x::xs) = (f x) :: (ourMap f xs);
val ourMap = fn : ('a -> 'b) -> 'a list -> 'b list
```

With this function we have a new declaration for squareAll.

```
- val squareAll = ourMap square;
val squareAll = fn : int list -> int list
```

The declaration for ourMap matches the pattern for list recursion. The only new element is that another argument, the function to be mapped, is added.

We are now in a position to see the value of curried functions. Consider the declaration below.

```
- val sub94All = ourMap (curriedDifference 94);
val sub94All = fn : int list -> int list
```

Recall that `curriedDifference 94` is a function that takes an integer and subtracts it from 94. The function `sub94All` subtracts every element of a list from 94 and produces a new list.

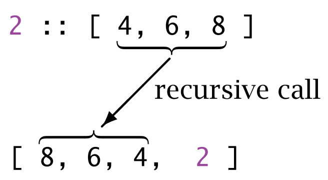
Let us consider another problem, that of reversing a list. We want a function that behaves as follows.

```
- ourRev0 [2,4,6,8];
val it = [8,6,4,2] int list
```

We have used the name `ourRev` to avoid confusion with the built-in function `rev` which does the same thing. The pattern is the same as before, with a base case and a recursive case, as outlined below.

```
fun ourRev0 nil      = ??
  | ourRev0 (x::xs)  = ??
```

We need only fill in the expressions for the two results. The base case is easy; the empty list read backwards is still the empty list. For the recursive step, we make an illustration.



To reverse a non-empty list, we reverse the rest of the list and then put the first element of the original list at the end of it. The idea is captured in the following declaration.

```
fun ourRev0 nil     = nil
  | ourRev0 (x::xs) = (ourRev0 xs) :: x;  (* error! *)
```

Warning! We have intentionally made a common error. Try to diagnose it before reading further.

Remember that the `::` operator takes an element and a list—in that order. In the declaration above, we have used it backwards, giving it a list and an element. We need a way to add an element as the *final* element of a list. There is no operator that will do it directly, but an easy way is to convert the element into a list and then to append the two lists using the append operator `@`.

```
fun ourRev0 nil     = nil
  | ourRev0 (x::xs) = (ourRev0 xs) @ [x];  (* correct! *)
```

This solution provides a clear example of list recursion and is correct, but it is not as efficient as possible. A different list-reversing function puts the recursion in an auxiliary function. The advantage of having a second function is that that function can have an additional variable which "accumulates" the result.

```
fun revAux acc nil     = acc
  | revAux acc (x::xs) = revAux (x::acc) xs;

fun ourRev1 lst = revAux nil lst;
```

The first argument to the function `revAux` is the accumulator. It starts out empty and receives one element at a time from the original list. Each element goes from the front of the original list to the front of the accumulator, so that the first element of the original list is the first element put into the accumulator and ends up at the end of the accumulator. The last element in the original list ends up at the front of the accumulator. When the original list, the second argument to `revAux`, is empty, the function returns the accumulator.

Practice Problem 11. Write a function that uses an accumulator to compute the length of a list.

Practice Problem 12. Write a function `pam` ("map" backwards) that takes a list of functions and applies each of them to a single argument. Give the type signature of your function.

Sometimes we carry out recursion on several arguments simultaneously. Let us represent polynomials as lists of coefficients. For example, $2 + X^3$ is represented as `[2,0,0,1]`. Write a function `polyadd` that computes the sum of two polynomials. (Observe that the representation is not unique. For example, `[0,1]` and `[0,1,0,0]` both represent the polynomial X.) To add two such polynomials, we simply add the components of the two lists. The recursive call is made on two shorter lists.

```
fun polyadd pl      nil     = pl
  | polyadd nil     ql      = ql
  | polyadd (p::ps) (q::qs) = p+q :: (polyadd ps qs);
```

Practice Problem 13. Write a function `polyconstmul` that multiplies a constant with a polynomial.

Practice Problem 14. Write a function `polymul` that multiplies two polynomials. The following mathematical identities will be helpful.

$$0 \cdot Q(X) = 0$$
$$(p_0 + p_1X + \ldots p_nX^n) \cdot Q(X) = p_0 \cdot Q(X) + (p_1 + \ldots p_nX^{n-1}) \cdot Q(X)$$

---

5 The Three Rules of Recursion

There are many kinds of recursion. We have just seen list recursion, and in previous course you saw numerical recursion. There will be other examples as our course progresses.

All types of recursion have the same characteristics. There is a *base case*—or a collection of base cases—for which the solution is immediate. And there is a *recursive step* in which the solution is assembled from the solutions to simpler problems. In the example with the variations on `ourRev`, the base case was to reverse the empty list, and the result was just the empty list. The result of recursive step, applied to a non-empty list, was constructed by making a recursive call to reverse the rest of the list and then attaching the former first element at the end.

In the CS 52 class in the fall of 2010, we formulated three rules of recursion to guide us in writing recursive functions. First of all, whatever kind of recursion you are using, there must be a base case. You have to stop somewhere!

First Rule of Recursion: *Remember the base case(s).*

It is usually, but not always, easy to identify the base cases. In the case of list recursion, the empty list is the base case. With numerical recursion, the values 0 or 1 often correspond to the base case. If you claim to be using recursion but cannot identify the base cases, then you are probably making a mistake.

Second Rule of Recursion: *Be sure the recursive calls are on "simpler" cases.*

The second rule insures that we make progress toward a base case. Each recursive call should be on a "simpler" case. With list recursion, that usually means a shorter list. With numerical recursion, it usually means a smaller number.

Third Rule of Recursion: *Believe . . . in the correctness of the recursive call.*

Students often understand programs by walking through them, one step at a time. While this strategy can be a good one, it fails for recursive programs and functions. The value of recursion is that we do not have to step through it; we can be confident that the recursive call works properly.

The principle of mathematical induction, a close relative of the technique of recursion, justifies the third rule. If a recursive function gives a wrong answer, then there must be a "simplest" argument for which the answer is wrong. For that argument, the recursive calls have even "simpler" arguments and must give correct answers.

The error is therefore in assembling the final result from the recursive results—and not in the recursive calls themselves.

The key to writing recursive functions is to decide what simpler problems to solve and then to create a solution to a given problem from the solutions to simpler problems. There is no need to think back further into the recursion; it will only make the problem more difficult. The third rule is really a statement of confidence in recursion, confidence that you will develop as you work with recursion.

In practice, there are some standard patterns that guide us. Suppose that we want to write a function rec, and we decide that there are three arguments.

```
fun recur a b c = ...
```

Somehow, we determine that the function should be list-recursive on the second argument. Then we have a standard pattern.

```
fun recur a nil     c = ...
  | recur a (y::ys) c = ... (recur b ys c) ...
```

The next steps are to determine the expression for the base case and the expression for the recursive step. In the latter, there must be a recursive call—which acts on a simpler instance of the list. Look back at this example and identify the uses of the three rules of recursion.

The patterns do not provide a mechanical alternative to analysis; rather they guide our thinking toward a correct and effective use of recursion. In some cases, the pattern will illuminate a flaw in our thinking and show us that recursion, or recursion on some particular argument, is not the appropriate method to use.

6 Numerical Recursion

The classic example of recursion is over the natural numbers $\{0, 1, 2, \ldots\}$. The base case is zero, and the induction step reduces a problem about n to a problem about a natural number less than n, usually $n - 1$. There are many variations; sometimes the recursion will start at 1, and occasionally there may be several recursive calls.

One snag to implementing numerical recursion in SML is that we do not have a type for the natural numbers; we have only int which includes negative values. Let us put that aside for now and not worry about negative numbers. They will be easy to add later.

Suppose that we have two functions pred and succ, which compute the predecessor and successor of a natural number. In SML, we could define them directly using + and -, as below, but we want to see how far we can get with *only* these two functions and the constant 0.

```
fun pred n = if n = 0 then 0 else n-1;
fun succ n = n+1;
```

Addition is an iteration of the successor function, so we can define addition.

```
fun add u 0 = u
  | add u v = succ (add u (pred v));
```

We can also define subtraction. Strictly speaking, we are defining *arithmetic subtraction* in which a result that would normally be negative is zero.

```
fun sub u 0 = u
  | sub u v = pred (sub u (pred v));
```

In both cases, we are following the rules of recursion. The recursion is on the second variable. We have a base case; the recursive call is on a simpler case, namely `pred v`; and we believe (We do, right?) that the recursive call gives us the correct answer.

Multiplication is iterated addition, so we can define multiplication as well.

```
fun mul u 0 = 0
  | mul u v = add (mul u (pred v)) v;
```

Let us expand our definitions to include negative integers. Since the recursion is on the second argument, we need be concerned with only that argument.

```
fun add u v = if v < 0
                 then ~(add (~u) (~v))
              else if v = 0
                 then u
                 else succ (add u (pred v));
fun sub u v = if v < 0
                 then ~(sub (~u) (~v))
              else if v = 0
                 then u
                 else pred (sub u (pred v));
fun mul u v = if v < 0
                 then mul (~u) (~v)
              else if v = 0
                 then 0
                 else add (mul u (pred v)) v;
```

Division is no harder except for the danger of dividing by zero. We postpone a discussion of it until we have more facilities for handling errors.

The functions so far are not particularly useful. SML already has built-in operators for arithmetic. The examples were chosen to illustrate numerical recursion and to demonstrate how fundamental recursion is. All of arithmetic can be developed with only `succ`, `pred`, and recursion.

Practice Problem 15. Consider the function below.

```
fun f(x,y) = if x = 0
               then y
               else f(x-1, x*y);
```

a. Give the type signature for `f`.

b. For which arguments does `f` terminate?

c. What is the mathematical meaning of `f(x,y)`?

d. Rewrite `f` so that it terminates on all arguments and its values are consistent with the meaning from part c.

Let us now turn to a different example. Suppose we want to create a list of natural numbers, starting at zero and ending at a given number. For example, `[0,1,2,3,4]` is such a list. Here is one attempt that uses numerical recursion directly.

```
fun interval0 k = if k < 0
                     then nil
                  else if k = 0
                     then [0]
                     else (interval0 (k-1)) @ [k];
```

It is a perfectly good solution, except that it is inefficient. At each recursive step we are appending a list to a singleton, and that list is getting larger and larger. As we will see later, this type of construction leads to an excessively large number of operations. It is usually faster to construct lists with `::` than with `@`. In order to work from the front of a list, we have to allow the first element of the list to vary as well as the last.

```
fun interval1 (j,k) = if k < j
                         then nil
                      else if k = j
                         then [j]
                         else j :: (interval1(j+1,k));
```

We now have a faster interval function, and we can construct a list starting at zero with a call to `interval1(0,k)`. The function `interval1` is certainly recursive—it contains a recursive call. But what about the base cases? Further, it appears that the recursion is on the argument `j`, but that value *increases* in the recursive call. How is that a "simpler case"?

The answer is that we are doing recursion not on `j` or `k`, but on the *difference* between those two values. The base cases occur when $\mathtt{k} - \mathtt{j}$ are less than or equal to zero. The recursive call is on a simpler case because $\mathtt{k} - (\mathtt{j} + 1)$ is smaller than $\mathtt{k} - \mathtt{j}$. With this interpretation, our definition of `interval1` satisfies all three Rules of Recursion: There are base cases, the recursive call is on simpler cases, and we see clearly how we are constructing the current case from the result of the recursive call.

Practice Problem 16. Write a function `sumInterval0` that takes a pair `(m,n)` of integers and evaluates the following expression. You may assume that both `m` and `n` are non-negative.

$$\mathtt{m} + (\mathtt{m} + 1) + (\mathtt{m} + 2) \ldots + (\mathtt{m} + (\mathtt{n} - 1)) + (\mathtt{m} + \mathtt{n})$$

Practice Problem 17. Write a different function `sumInterval1` that takes a pair `(u,v)` of integers and evaluates the expression below. You may assume that $\mathtt{u} \leq \mathtt{v}$.

$$\mathtt{u} + (\mathtt{u} + 1) + (\mathtt{u} + 2) \ldots + (\mathtt{v} - 1) + \mathtt{v}$$

We end the section with an interesting problem that mixes list recursion and numerical recursion. Suppose we want a function `subList` that takes a list and a pair `(j,k)`, and returns a list containing of the `j`-th through the `k`-th elements of the original list. We assume that the elements of a list are numbered starting with index zero. Let us try to make sense out of the many different cases.

- If the given list is empty, then any sublist of it is empty.
- If $\mathtt{k} < \mathtt{j}$, then the resulting list is empty.
- If `j` is zero, then we may take the first element of the given list and add it to the front of the first `k-1` elements of the rest of the list.
- If `j` is positive, then we can discard the first element of the given list and find the sublist corresponding to `(j-1,k-1)` of the rest of the list.
- If `j` is negative, then it is not the index of any element of the given list, and we can find the sublist corresponding to `(j+1,k)` of the given list.

Before reading further, convince yourself that these cases cover all the possibilities.

We can now write an SML function that embodies our discussion.

```
fun subList nil     _     = nil
  | subList (x::xs) (j,k) =
        if k < j
           then nil
        else if j = 0
           then x :: (subList xs (j, k-1))
        else if j < 0
           then subList (x::xs) (j+1, k)
           else subList xs (j-1,k-1);
```

Let us study the function through the three Rules of Recursion. We are doing recursion on a list *and* on the difference k − j. The base cases occur when the list is empty and when k − j is negative.

The three recursive calls are all on simpler cases. The call `subList xs (j, k-1)` is a simpler case because the list is shorter *and* k − j is smaller. The call `subList (x::xs) (j+1, k)` is a simpler case because k − j is smaller. The call `subList xs (j-1,k-1)` is a simpler case because the list is shorter.

Each of the three recursive branches gives the "correct answer." When `j` is zero, we take the first element and attach it to the sublist of size `k-1`. If `j` is negative, we ignore that value of `j` and move on. (We could in fact skip some steps and take the `j`-argument all the way to zero instead of just adding one.) If `j` is positive, then we discard the first element of the list and adjust the indices for the rest of the list.

Finally, we must be sure that our solution is complete, that we have covered all the cases. We have, because the list is either empty or it is not. We certainly have covered the case in which the list is empty. And when it is not empty, we have cases for `j` being negative, zero, and positive. There is no case that will "slip through the cracks."

7 List Recursion Revisited

Recall that the elements in a list are ordered and that an element may appear more than once. Suppose that we want a function that will remove duplicated elements from a list. It is easiest to start with a membership function.

```
fun member e nil     = false
  | member e (x::xs) = e = x orelse member e xs;
```

The function `member` returns a value of type `bool`. Notice the use of the boolean connective `orelse`.

Now we can write one version of the function `uniquify`. We will encounter several other versions later in the course.

```
fun uniquify0 nil     = nil
  | uniquify0 (x::xs) = if member x xs
                          then uniquify0 xs
                          else x :: (uniquify0 xs);
```

Practice Problem 18. Our version of uniquify preserves the order of the elements. Write a version that uses an accumulator variable but does not necessarily preserve order.

Practice Problem 19. Our version of uniquify retains the *last* occurrence of an element. Write a version that preserves order but retains the *first* occurrence of an element.

As a second example, we consider more complicated lists. The elements of a list may be of any type. In particular, we can have a list of lists, ... or even a list of lists of lists, and so on. An example of a function that produces a list of lists is one that produces all the permutations of a given list. Given a list argument, we want to create a list of all the different orderings of the elements in the given list. Given the list [1,2,3], we desire the result below.

```
[ [1,2,3], [1,3,2], [2,1,3],
  [2,3,1], [3,1,2], [3,2,1] ] : int list list
```

For our purposes, the order of the different permutations will not matter; we just want to make sure that we have them all. If the given list has n elements, then there will be $n!$ permutations.

An obvious strategy is to use list recursion. The base case is the empty list, which has 0 elements. The result has $0! = 1$ permutations—in particular, *the result is not empty!* There is one possible permutation of the empty list, namely the empty list itself. Therefore, the result of the base case will be a one-element list whose sole element is the empty list.

```
fun perm nil    = [ nil ]
  | ...
```

We have just made good use of the First Rule of Recursion. Let us move to the Second Rule, the recursive step. In the example with the argument [1,2,3], the recursive call would naturally be on the list [2,3] and give us the result [[2,3],[3,2]]. The crucial question is “How do we incorporate the first element 1 into our recursive result to obtain all six permutations?” Simply putting the element 1 on the front of the recursive result will not work; it will cause a type error. Putting the element 1 on the front of *each element* of the recursive result will not give rise to an error, but it will yield only two of the six permutations. On reflection, we see that we want to insert the element 1 into

- every possible position in
- every list in the recursive call.

There are two steps, and it is easiest to separate them. Let us start by considering the "every possible position" part and write a function that will insert an element into every possible position of a single list.

```
fun insEverywhere e nil     = [[e]]
  | insEverywhere e (y::ys) =
      (e::y::ys) :: (map (fn u => y::u) (insEverywhere e ys));
```

It is worthwhile to spend some time on insEverywhere. If we are to insert an element e into a list of length n, there are $n+1$ places to put it. In particular, if the list is empty, there is exactly one place to put it, and the result of inserting e into the empty list is [e]. The result of insEverywhere is a list of all possibilities, so the result in the base case is a list with one element, namely the only possibility [e].

The recursive step is most easily understood from the inside out. The recursive call insEverywhere e ys produces a list of all the possible results of inserting e into ys. If we add y onto the front of each of those lists, with map, then we obtain some of the ways to insert e into ys. In fact, we obtain all of the ways in which e is inserted *after* the first element y. The only possibility left out is the one in which e comes before y, and that is the one that is added to the front of the result.

Now that we have insEverywhere, let us forget about how it works and use it to complete the construction of the permutation-generating function. We must apply insEverywhere to "every list in the recursive call." It is natural to try to use map.

```
  | perm (x::xs) = map (insEverywhere x) (perm xs);
```

The expression with map gives us all the permutations, but it is of the wrong type. It is a list of lists of lists instead of a list of lists. The function insEverywhere provides a list for each element of the recursive call, so in the case of [1,2,3] we obtain

```
[ [ [1,2,3], [2,1,3], [2,3,1] ],
  [ [1,3,2], [3,1,2], [3,2,1] ] ] : int list list list
```

We need one more function, one that appends all the lists in a list of lists. Fortunately, it is an easy application of list recursion.

```
fun appendAll nil     = nil
  | appendAll (z::zs) = z @ (appendAll zs);
```

We can now write the function perm in all its glory.

```
fun perm nil     = [ nil ]
  | perm (x::xs) =
    appendAll (map (insEverywhere x) (perm xs));
```

Take some time to examine the structure of perm and understand how it creates the final result from the recursive call.

Practice Problem 20. Recall that a list is a *palindrome* if it is its own reverse. The following function purports to detect palindromes. Is it correct? Explain.

```
fun isPalindrome xl =
    let
        fun isPalAux acc nil = null acc
          | isPalAux acc (x::xs) = acc = x::xs orelse
                                   acc = xs orelse
                                   isPalAux (x::acc) xs;
    in
        isPalAux nil xl
    end;
```

---

8 An Aside: When *Not* to Use Recursion

Recursion is a powerful technique, and the patterns that we have presented are appealing. However, not every function is defined recursively, and in those cases there is no need to use a recursive pattern.

Consider the example of creating a "list palindrome." Given [1,2,3], we can create the list [1,2,3,3,2,1] which is the same when read forward or backward. The strategy is to append the original list with its reversal. (Astute readers will observe that we are creating only the list palindromes of even length.) We can accomplish the task with a simple function.

```
- fun listpalindrome xl = xl @ (rev xl);
val listpalindrome = fn : 'a list -> 'a list

- listpalindrome [4,7];
val it = [4,7,7,4] : int list
```

Notice that there is no recursion here, even though one of the component functions, rev, is defined recursively elsewhere.

Sometimes students become so attached to list recursion that they begin to write the function using the standard pattern for list recursion.

```
fun listpalindrome nil     = ...
  | listpalindrome (x::xs) = ...
```

At best, the use of the pattern makes the code too complicated. Often, it leads to an error. When you do use recursion, be clear about why you are doing it. Identify the variable on which you are recursing, and understand how the final result is produced from the recursive call. If you do not have a recursive call, then you are not using recursion.

Practice Problem 21. Consider the parallel problem of creating a palindrome from a *string.* Write a function `stringpalindrome`.

---

9 Numerical Recursion Revisited: Division

Division on the natural numbers gives us a place to exercise our facility with numerical recursion. From one point of view, division is simply repeated subtraction, just as multiplication is repeated addition. Division is a little more complicated, however, because there are two results—a quotient and a remainder.

There is also a potential error—dividing by zero. When that occurs, we will follow the practice of SML raise the exception `Div`. Review exceptions in Section 7 of Part II.

As we did with addition, subtraction, and multiplication in Section 6, we start with the natural numbers. Given a numerator n and a non-zero denominator d, we seek a quotient q and a remainder r satisfying

$$n = d \cdot q + r \text{ and } 0 \le r < d.$$

We can think of having all four variables present. We start with $(n, d, q, r) = (n, d, 0, n)$; the values $q = 0$ and $r = n$ satisfy the equation on the left, above. If in addition $0 \le r < d$, then the pair (q, r) is the desired result. If not, then we can subtract d from r and add 1 to q. The equation is still satisfied (Verify!), and we are closer to a solution because r is smaller. In the sense of the Second Rule of Recursion, a "simpler case" is one with r being closer to zero. This analysis translates immediately to an SML function.

```
fun quoremAux (n,d,q,r) =
        if r < d
           then (q,r)
           else quoremAux(n,d,q+1,r-d);

fun quorem (n,d) = quoremAux(n,d,0,n);
```

Although correct for the natural numbers, this code is painfully slow. But it is important because it illustrates a new variation of recursion.

Later, we will look at other implementations of division, including some that mimic the long division algorithm. For now, it is sufficient to extend our division algorithm to all the integers. For this exercise, we adopt a convention that is used by most mathematicians but is seldom seen on computer hardware: We insist that the remainder be non-negative. There are four cases. In each one, we negate a negative numerator or divisor, and obtain a corresponding quotient and remainder.

- $0 \leq n$ and $0 < d$. Our function `quoremAux` gives us the correct quotient-remainder pair (q, r).
- $n < 0$ and $0 < d$. We divide $-n$ by d to obtain values q and r satisfying $-n = d \cdot q + r$ and $0 \leq r < 0$ and rewrite the equation to obtain $n = d \cdot (-q) - r$. If $r = 0$, the result is the quotient-remainder pair $(-q, r)$. If $r \neq 0$, we must adjust the remainder by adding d and the quotient by subtracting 1; the result is $(-q - 1, r + d)$.
- $0 \leq n$ and $d < 0$. We divide n by $-d$ to obtain values q and r satisfying $n = (-d) \cdot q + r$ and $0 \leq r < 0$. Rewriting gives $n = d \cdot (-q) + r$. The result is the quotient-remainder pair $(-q, r)$.
- $n < 0$ and $d < 0$. We divide $-n$ by $-d$ to obtain values q and r satisfying $-n = (-d) \cdot q + r$ and $0 \leq r < 0$. Rewriting gives $n = d \cdot q - r$. As in the second case, if $r = 0$, the quotient-remainder result is (q, r). If $r \neq 0$, then we adjust the remainder and the result is $(q - 1, r + d)$.

It is routine, but tedious, to translate the analysis into SML code. To make the presentation cleaner, we use the library function `Int.sign` which returns `~1`, `0`, or `1` according to whether its argument is negative, zero, or positive.

```
fun quorem (n,d) =
       case (Int.sign n, Int.sign d) of
            ( _, 0) => raise Div
          | ( 0, _) => (0,0)
          | ( 1, 1) => quoremAux(n,d,0,n)
          | (~1, 1) => let
                           val (q,r) = quoremAux(~n,d,0,~n)
                       in
                           if r=0 then (~q,r) else (~q-1,r+d)
                       end
          | ( 1,~1) => let
                           val (q,r) = quoremAux(n,~d,0,n)
                       in
```

```
                    (~q,r)
                end
    | (~1,~1) => let
                    val (q,r) = quoremAux(~n,~d,0,~n)
                in
                    if r=0 then (q,r) else (q-1,r+d)
                end;
```

Practice Problem 22. Integer division in most computer languages follows the division method of the underlying hardware. The usual method is to round toward $-\infty$, in which case the remainder takes the sign of the divisor. Modify the function `quorem` to satisfy that condition on the remainder.

---

10 Types

We have seen a collection of built-in types and turn now to programmer-defined types. We can construct new types out of old ones, and we have the option of using recursion.

10.1 Naming Types

The keyword `type` is used to define a one-word name for a known type. It is analogous to the `val` declaration for values.

```
- type ilist = int list;
type ilist = int list
```

After the definition, `int list` and `ilist` are names for the *same type.* We can, for example, compare a value that is identified as an `int list` with one identified as an `ilist`.

Type definitions may be parameterized, making them more interesting and more useful.

```
type ('d, 'r) mapping = ('d * 'r) list;
```

A `mapping` is a list of ordered pairs; it is one way to represent a function from one set of values to another. For example, we might want to map words to their definitions using a `(string, string) mapping`.

10.2 Datatypes

Even with parameters, type definitions are of limited use; they simply provide abbreviations. A more powerful construction uses the keyword `datatype`.

```
- datatype bit = Zero | One;
datatype bit = One | Zero
```

Here we have created a brand new type whose name is `bit` and whose two values are `Zero` and `One`. In a subsequent section, we will have use for this type. By itself, the type is not much use; we must create functions that operate on the values of the type. We may use pattern-matching on values of the type.

```
- fun complement Zero = One
=   | complement One  = Zero;
val complement = fn : bit -> bit
```

Practice Problem 23. Pretend that the type `bool` does not exist. Define it and the associated functions `not`, `andalso`, and `orelse`. (For this exercise, assume that `andalso` and `orelse` evaluate both their arguments. SML functions evaluate *all* their arguments, so it is not possible to obtain the short-circuit behavior of these operations with functions.)

Practice Problem 24. With the result of the previous exercise, we try to define a function `ifthenelse` as follows.

```
- fun ifthenelse true  texpn _     = texpn
=   | ifthenelse false _     fexpn = fexpn;
```

Is this a correct definition? Is it an acceptable replacement for the built-in `if-then-else` expression?

10.3 Example: Backward Lists

The words `Zero` and `One` are called *data constructors* in SML. They may be used by themselves, as they are here, or they may be tags for more complicated structures. Consider the case of backward lists. Instead of all of the activity occurring at the head of a list, everything happens at the tail. For ordinary lists, we have the notions of *first, rest,* and *cons.* The last operation, cons, forms a new list from an element and a list; it is the analog of the SML operation `::`.

```
- datatype 'a tsil = Lin | Snoc of 'a tsil * 'a;
datatype 'a tsil = Lin | Snoc of 'a tsil * 'a
```

Here, we have used the SML convention of writing types in lower case and data constructors with a leading capital letter. We can now proceed to define our backward list functions.

```
- exception Ytpme;
exception Ytpme
- fun tsrif Lin          = raise Ytpme
=   | tsrif (Snoc(_,x)) = x;
val tsrif = fn : 'a tsil -> 'a
- fun tser Lin          = raise Ytpme
=   | tser (Snoc(sx,_)) = sx;
val tser = fn : 'a tsil -> 'a tsil
```

Notice how the recursion follows the structure of the datatype. There is a base case involving `Lin` and a recursive step involving `Snoc`.

Practice Problem 25. Write a tsil-reversing function.

Practice Problem 26. Write a function to append two values of type `tsil`.

Practice Problem 27. Pretend that the `option` type is not present in SML and give a parameterized type definition for it. Define the function `valOf`.

10.4 Example: Nested Boxes

Nested boxes provide another example of user-defined types.[1] Nested boxes are colored cubes that fit inside one another. The innermost box contains "nothing"; every other box contains another box. A box that is not "nothing" has a dimension, a color, and a box inside of it. Here are the initial declarations.

```
exception NestedBox;

datatype color = Red   | Orange | Yellow |
                 Green | Blue   | Violet;

datatype nbox = Nothing
              | Cube of real * color * nbox;
```

[1]This example is adapted from Hansen and Rischel, *Introduction to Programming in SML*, section 8.1

The exception is there because we have conditions on type nbox: the dimension must be positive, and an inner box must have a smaller dimension than one containing it.

It may seem strange to have a value Nothing, for it is not really a box. But it is more convenient to consider Nothing to be a cbox than to deal with the special cases that arise when a box may or may not contain something. Here is an example of three nested boxes .

```
Cube(4.0,
     Green,
     Cube(2.5,
          Red,
          Cube(0.9,
               Yellow,
               Nothing)))
```

Suppose that we want to add to our nest a new box of a certain size and color. If the new box is larger than the outer box of our nest, then we simply put the nest in the new box. If the new box is smaller, then we (recursively) place the new box inside the outer box. This strategy is reflected in the following SML function which places a new box of size dim and color col into an existing box. The result is a new nest with one more box.

```
fun insert(dim, col, Nothing) =
       if dim <= 0.0
          then raise NestedBox
          else Cube(dim, col, Nothing)
  | insert(dim, col, Cube(d, c, b)) =
       if dim <= 0.0
          then raise NestedBox
       else if dim < d
          then Cube(d, c, insert(dim, col, b))
       else if dim = d
          then raise NestedBox
          else Cube(dim, col, Cube(d, c, b));
```

As you can see, we have preserved the conditions that the dimensions are positive and that a smaller box cannot contain a larger one. The code is a little cluttered because we must test for a non-positive dimension in each of the branches.

Practice Problem 28. Write a function difflist that computes the sequence of dimension-differences of a nest of type nbox. For example,

`difflist Nothing` returns `[]`,
`difflist(Cube(1.0,Green,Nothing))` returns `[]`, and
`difflist(Cube(1.0,Green,Cube(0.5,Red,Nothing)))` returns `[0.5]`.

Practice Problem 29. Suppose that `nbox` had instead been defined as a list.

```
type nbox = (real * color) list;
```

Write functions `insert` and `difflist` for this version of the type.

Practice Problem 30. [Challenging] Consider a variation `mnbox` of the nested box example in which a box may contain *multiple* boxes. Packing cubes into larger cubes is a difficult mathematical problem, so we make the following simplifying assumptions: A non-empty cube contains cubes *which all have the same size.* Further, the smaller cubes are oriented so that their sides are parallel to the sides of the enclosing cube. This means that a cube of side S may contain as many as $\lfloor S/s \rfloor^3$ cubes of side s.[2]

Write a type declaration and an `insert` function for `mnbox`.

10.5 Example: Trees

Trees are ubiquitous in computer science. You will study them in depth in other courses. Here, we simply see how they are treated in SML. A *binary tree* is either empty or it is a *node* with a value and two *subtrees.* We distinguish the subtrees as *left* and *right.* The value at a node can be a very complicated record; for our purposes it can be as simple as a single integer.

The two subtrees of a node are called the *children* of the node, and the node is the *parent* of its children. A *leaf* is a node whose two subtrees are empty. The *root* of a non-empty tree is the unique node which has no parent. Here is the SML datatype.

```
datatype 'a binary_tree =
             BTEmpty
           | BTNode of 'a binary_tree * 'a * 'a binary_tree;
```

The datatype is recursive, which allows us to write functions that are recursive on binary trees. For example, we can compute the number of nodes.

```
fun nodeCount BTEmpty              = 0
  | nodeCount (BTNode(lc,_,rc)) = 1 + (nodeCount lc)
                                    + (nodeCount rc);
```

[2]The notation $\lfloor x \rfloor$ denotes the *floor* of x, the least integer not exceeding x.

The *height* of a binary tree is the length of the longest chain of nodes from the root to a leaf. By convention, we set the height of an empty tree to -1.

```
fun height BTEmpty                = ~1
  | height (BTNode(left,_,right)) =
       Int.max(height left, height right);
```

We can create a list of all the values in a tree. One strategy is to list a node, then the nodes in its left subtree, and then the nodes in the right subtree. This is called a *preorder* traversal of the tree.

```
fun preorder BTEmpty                      = nil
  | preorder (BTNode(left, value, right)) =
       value :: ((preorder left) @ (preorder right));
```

The type of `preorder` is `'a binary_tree -> 'a list`. There are also an *inorder* traversal which lists the value at a node after the values of the nodes in left subtree and before those in the right subtree. The *postorder* traversal lists the values from the left subtree, then the values from the right subtree, and then the value at the current node. The functions for inorder and postorder traversals are similar to the one for preorder.

Notice that all the functions for binary trees have the same recursive structure. It arises from the recursive construction of the datatype, just as our pattern for list recursion arises from the recursive formulation of lists.

Practice Problem 31. A *binary search tree* is a binary tree in which all the values from the left subtree are less than the value at the node, and the value at the node is less than all the values in the right subtree. Assume that the values in a tree are integers. Write a function that will insert a given value into a given binary search tree. (Philosophical point: In the functional paradigm, everything is a value. We cannot change a value once it is created, so we do not actually "change" a binary tree by inserting something into it. Instead, we create a new tree which is like the old tree except that it has a new value.)

Practice Problem 32. Write a function `reflect` that operates on binary trees and satisfies the following condition.

```
inorder (reflect bt) = rev (inorder bt)
```

---

11 Functional Programming and List Recursion

Functional programming and list recursion are orthogonal ideas, but they tend to occur together in practice. One of the earliest functional programming languages, Lisp, takes the list as its basic data structure. (Some of you may have encountered a modern dialect of Lisp called Scheme.)

One appeal of the functional paradigm is that it is simple. There is only a limited number of actions that we can take with functions. We may apply function to arguments, use functions as arguments to other functions, and define new functions—perhaps using existing functions as building blocks. Sometimes, when we define new functions, we use recursion. And as we have seen, there are some common templates for that recursion which have found their way into higher-order functions, like `map`.

In this section, we study some of those general functions, the ones that computer scientists expect to find in a language described as "functional." Among those functions are the following.

Application is the act of applying a function to arguments to obtain a result. It is present in some form in virtually all programming languages.

Function definition is a facility for obtaining (and in some cases naming) new functions that are created from more basic components. We have already seen many function definitions in SML.

Composition is an operation that creates a new function out of two existing ones. We know that SML uses `o` as the composition operator.

Mapping is the operation that creates a new list by applying a function to each element of a given list. We are familiar with the SML function `map`.

Filtering is the operation that creates a "sublist" of the elements of a given list that satisfy some criterion. We shall see that the SML function `List.filter` encapsulates a recursive template in a way similar to `map`.

Folding is the operation that combines the elements of a list into a single element. The simplest example is computing the sum of the elements in a list of integers. We shall see that folding is in one (not necessarily practical) sense the fundamental operation of list recursion.

All of these functions are already part of SML. We give definitions here to illustrate their connection to functional programming, lists, and list recursion.

11.1 Function Definition

Most frequently, functions are declared with the keyword `fun`. We have seen many examples.

Many functions have names, but the names are not necessary. We can talk about "the function that squares" or "the function that shifts the characters in a string to uppercase" without giving them names. Such *anonymous* functions are handy for one-time applications or for use as arguments to other functions. In SML, we create anonymous functions with the keyword `fn`.

```
fn x => x*x
```

We can also obtain functions from application. We have already seen `map` which has the type signature below.

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

If we apply `map` to a function of type `'a -> 'b`, we obtain a function of type `'a list -> 'b list`. We can leave that function anonymous or give it a name with a value declaration.

```
val squareAll = map (fn x => x*x);
```

The important point to be made here is that functions are simply values. Like lists and other kinds of computational objects, they exist independently of how we name them. In the functional programming paradigm, a computation is a function that takes input to output. We create useful computations by combining simple functions using application and function definition.

11.2 Composition, Revisited

One important operation on functions is composition. The idea is that we create a new function from functions `f` and `g`; the instructions are first apply `g` to obtain a result and then apply `f` to that result.

In SML, the composition of `f` and `g` is written `f o g`, and it behaves as the anonymous function below.

```
fn x => f(g x)
```

If `f` is of type `'b -> 'c` and `g` is of type `'a -> 'b`, then `f o g` is of type `'a -> 'c`.

11.3 Mapping, Revisited

The SML function `map` behaves as if it is declared as follows.

```
fun map _ nil     = nil
  | map f (x::xs) = (f x) :: (map f xs);
```

It displays all the characteristics of a list-recursive function. There is a base case, corresponding to an argument which is an empty list. And there is a recursive step which assembles the final result from a recursive call on a simpler case.

Keep in mind that the `map` function does the recursion for us. More precisely, it encapsulates the recursion. Applying `map` *replaces* an explicit recursion. We emphasize this point because some students are inclined to use *both* the function `map` *and* an explicit recursion. One or the other is sufficient.

11.4 Filtering

It is easy to select the even elements from a list of integers. The result is a potentially-shorter list.

```
fun evenList nil     = nil
  | evenList (x::xs) = if x mod 2 = 0
                          then x :: (evenList xs)
                          else evenList xs;
```

It would be equally easy to select the multiples of three ... or any other number. Having seen the pattern, we can write a function to select the integers with any property.

A function that tells us whether its argument satisfies some condition is a *predicate.* It is a function of type `'a -> bool`. Using the example above as a template, we can write a function that selects from a list the elements that satisfy an arbitrary predicate.

```
fun filter _    nil     = nil
  | filter pred (x::xs) = if pred x
                             then x :: (filter pred xs)
                             else filter pred x;
```

As mentioned earlier, there is no need to write this definition. SML has a built-in function `List.filter`.

Practice Problem 33. SML has a built-in function `List.exists` which has the type signature `('a -> bool) -> 'a list -> bool`. It takes a predicate and a list and returns `true` if an element in the list satisfies the predicate. Write two declarations for a function `exists`, one that is directly recursive and one that uses `filter`.

Practice Problem 34. SML has another built-in function `List.all` which returns `true` if *all* the elements in the list satisfy the predicate. Repeat the previous exercise for a function `all`.

11.5 Folding

By now, writing a function to add all the elements in a list of integers is easy.

```
fun addAll nil      = 0
  | addAll (x::xs) = x + (addAll xs);
```

We can just as easily multiply all the integers in a list, or concatenate all the strings in a list, or append all the lists in a list of lists.

```
fun multAll nil      = 1
  | multAll (x::xs) = x * (multAll xs);

fun concAll nil      = ""
  | concAll (x::xs) = x ^ (concAll xs);

fun appdAll nil      = nil
  | appdAll (x::xs) = x @ (appdAll xs);
```

The only differences among these examples are the function names, the constants in the `nil` case (shown above in purple), and the operations (shown in olive). There is a clear pattern that we can use to create a general function.

```
fun foldr oper base nil      = base
  | foldr oper base (x::xs) =
        oper(x, foldr oper base xs);
```

The function `foldr` has the following type signature. It takes a function, a base value, and a list, and returns some kind of value.

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

The only concession we made in generalizing the examples is in turning the operation into a function on ordered pairs. Fortunately, there is an SML operator `op` that converts a binary operation into a function on pairs. Therefore, we can rewrite all the specific functions above using `foldr`.

```
val addAll  = foldr (op +) 0;
val multAll = foldr (op * ) 1;
val concAll = foldr (op ^) "";
val appdAll = foldr (op @) nil;
```

Syntactical detail: The space after the symbol * in the definition of `multAll` is necessary to prevent SML from seeing a comment-closing token.

The `r` in `foldr` stands for “right.” The function accumulates values from the right:

```
addAll [1,2,3,4] means 1 + (2 + (3 + (4 + 0))).
```

Notice the zero on the right from the base case of the recursion. The "rightness" of the folding function does not matter for addition, but it does make a difference for subtraction or division.

There is also a function `foldl` that accumulates from the left. It uses the argument `base` as an accumulator. For addition, we have the following.

```
fun addAllAux acc nil      = acc
  | addAllAux acc (x::xs) = addAllAux (x + acc) xs;

val addAllLeft = addAllAux 0;
```

Using that as a guide, we can give a general definition of `foldl`. It has the same type signature as `foldr`, but it behaves slightly differently.

```
fun foldl oper base nil     = base
  | foldl oper base (x::xs) =
        foldl oper (oper(x,base)) xs;
```

An alternate declaration of `addAllLeft` uses `foldl`.

```
val addAllLeft = foldl (op +) 0;
```

It is a function that accumulates from the left:

```
addAll [1,2,3,4] means 4 + (3 + (2 + (1 + 0))).
```

Practice Problem 35. By hand, compute the two values

`foldr (op -) 0 [4,7]` and `foldl (op -) 0 [4,7]`.

Practice Problem 36. What do the following functions do?

```
foldr (op ::) nil
foldl (op ::) nil
```

Practice Problem 37. The function `foldr (op ::)` has the type signature

```
'a list -> 'a list -> 'a list.
```

What does the function do?

Practice Problem 38. Write a one-line function using `foldr` that computes the length of a list. Write another version that uses `foldl`.

Practice Problem 39. Consider a function `foldsubl` that takes a non-negative integer `n` and returns the value `foldl (op -) 0 [0,1,...,n]`.

a. Give a complete SML declaration for `foldsubl`. (Hint: Recall the function `interval1` from Section 6.)

b. Use your declaration from part a to compute `map foldsubl [0,1,...,9]`.

c. Generalize the result of part b and give an alternate, non-recursive declaration for `foldsubl`. Prove, by mathematical induction that both declarations give the same values for non-negative arguments.

Practice Problem 40. Repeat Practice Problem 39 with a function `foldsubr` that uses `foldr` instead of `foldl`.

Practice Problem 41. [Challenging] Prove that `foldr f b xl` gives the same result as `foldl f b (rev xl)`.

Practice Problem 42. Show that `map` and `filter` can be defined, without using recursion, in terms of `foldr`.

Practice Problem 43. Show that `foldr` can be defined, without using recursion, in terms of `foldl`.

Practice Problem 44. [Challenging] The built-in SML function `ListPair.zip` is naturally defined by simultaneous recursion on two lists. Can it be defined, without recursion, using only the folding functions?

```
    fun ListPair.zip (_,     nil  ) = nil
      | ListPair.zip (nil,   _    ) = nil
      | ListPair.zip (x::xs, y::ys) = (x,y) :: (ListPair.zip(xs,ys));
```

Practice Problem 45. [Challenging] After solving the previous problems, one might conjecture that *every* use of recursion can be replaced with applications of `foldl`. Prove the conjecture or find a counterexample.

12 An Aside: Efficiency

The primary goal of a programmer is to write correct programs. A secondary goal is to make them run fast. Although it is a little off-topic for this course, we list some things that a programmer might do to increase speed.

Keep in mind that it is rare for the performance of a specific function to be critical. Most of the time—especially in introductory computer science courses—speed is not important at all. In the few cases in which it does matter, one must take the time to identify which functions are taking excessive time and to focus one's efforts on those parts of the program.

12.1 Reducing the Number of Steps

Consider the problem of raising a real number to a power. For the simplest function, it takes $n - 1$ multiplications to compute x^n. Let us assume that `n` is non-negative.

```
fun power0 x n =
    if n = 0
       then 1.0
       else x * (power0 x (n-1));
```

However, if we observe a couple of mathematical identities, we can reduce that number significantly.

$$x^{2n} = (x^n)^2 \quad \text{and} \quad x^{2n+1} = (x^n)^2 x$$

Translating directly gives the following function.

```
fun power1 x n =
    if n = 0
       then 1.0
    else if n mod 2 = 0
       then (power1 x (n div 2)) * (power1 x (n div 2))
       else (power1 x (n div 2)) * (power1 x (n div 2)) * x;
```

If the system actually computes *both* copies of `(power1 x (n div 2))`, then we have gained nothing. Many modern programming language systems will recognize the "common subexpression" and compute it only once. If we want to be absolutely sure, we can write it differently.

```
fun square (x : real) = x * x;
fun power2 x n =
    if n = 0
       then 1.0
    else if n mod 2 = 0
       then square (power2 x (n div 2))
       else square (power2 x (n div 2)) * x;
```

Notice that we had to be explicit in the definition of `square` that the argument is of type `real`. Otherwise, SML would have assumed the type `int`.

Practice Problem 46. Experiment with the three versions of the power function by computing 1.0^n for large values of n. Can you tell if the SML interpreter eliminates common subexpressions? Here is a framework for timing the execution of a function.

```
fun timeIt (f,n) =
    let
        val start = Time.now();
        val result = f n;
        val stop = Time.now();
        val duration = Time.-(stop, start);
    in
        (Time.toMicroseconds duration, result)
    end;
```

The Fibonacci function provides another example. Here is the mathematical definition.

$F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $1 < n$.

The direct translation into SML creates a function that makes exponentially many recursive calls. We can improve it by carrying along two values. Consider a function G satisfying $G(n) = (F(n), F(n-1))$. Then we can compute $G(n+1)$ as $(F(n) + F(n-1), F(n))$, yielding a recursive function.

```
local
   fun increment (u,v) = (u+v,u);
   fun g 0 = (0, 1)
     | g k = increment (g (k-1));
in
    fun fib k = #1(g k);
end;
```

Our new function `g` makes only `k` recursive calls, a major improvement! (There is, however, a closed formula that computes the Fibonacci function. Using it, we need only *one* function call. The point here is that we often can reduce the number of function calls, even when there is no elegant mathematical shortcut.)

12.2 Avoiding Expensive Constructions

We can go a little further in optimizing our calculation of Fibonacci numbers. Think of the function `g` as operating on a triple containing the current value of `k`, `u`, and `v`. Effectively, we are applying the increment function `k` times and using the triple as an accumulator.

```
local
   fun g(k,u,v) = if k=0
                  then u
```

```
                           else g(k-1,u+v,u);
in
   fun fib k = g(k,0,1);
end;
```

We have not further reduced the number of function calls, but we have given the SML system a way to reduce the overhead of those calls. The crucial property of our function `g` is that it is *tail recursive.* That means that the recursive call is the *very last* function to be applied. It allows the SML system to convert recursion into iteration. The process can be described as "keep modifying the triple until you are finished, and then return the second component." Many, but not all, recursive functions can be written in the tail recursive fashion.

Another feature of our function `g` is that it is uncurried. While currying is elegant, useful, and often essential, it does come with a bit of overhead. The SML system must create functions, and not simpler values, as intermediate steps.

When a function is called only a few times, there is little need to worry about the overhead of function calls. In a few rare cases, when the number of calls rises, we can profitably resort to the techniques outlined here. In subsequent courses you will learn more powerful techniques for improving efficiency.

Practice Problem 47. Write a tail recursive version of the factorial function.

Practice Problem 48. Our first examples of numerical recursion in Section 6 were the functions `add`, `sub`, and `mul` defined in terms of the more primitive functions `pred` and `succ`. Give tail recursive versions of `add`, `sub`, and `mul`.

Practice Problem 49. Write a list-reversing function using functions that are tail recursive and uncurried. Start with the discussion at the end of Section 4.

Practice Problem 50. Is `foldr`, as we described it in Section 11.5, tail recursive? What about `foldl`?

---

13 Adventures in Arithmetic: Bits

A value in a computer's memory is simply a string of zeroes and ones—a list of bits. One way to interpret a value in memory is to view it as the binary representation of a natural number.

Consider the elementary school algorithm for addition. It is even easier in binary. Here is a partially-completed addition.

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 10011 \\ +\ 1001 \\ \hline 00 \end{array}$$

In the rightmost column, we added 1 and 1 to obtain a result of 0 and a carry of 1. Then, in the next column, we added 1, 0, and the carry from the previous column to obtain a result of 1 and a carry of 1. The process continues from right to left. The next result bit will be 1, and the carry will be 0.

Using our type `bit` from the Section 10, let us define a type for representations of the natural numbers.

```
datatype bit = Zero | One;
type natnum = bit list;
```

A value of type `natnum` is simply a list of bits. When we interpret it as a binary representation, it is convenient to consider the first element to the the least significant bit. The practice may seem backwards at first, because we read lists starting from left, and we read numbers with the high-order digit on the left. The reason is that we usually start with the low-order bit, and we want the corresponding bits to "line up." Think about the addition example, above.

We first write a function for single-bit addition. The argument is a triple consisting of the carry-in bit and the two bits to be added. The result is a pair consisting of the sum bit and the carry-out bit.

```
fun bitadd (Zero, Zero, v   ) = (v, Zero)
  | bitadd (Zero, One,  Zero) = (One, Zero)
  | bitadd (One,  Zero, Zero) = (One, Zero)
  | bitadd (One,  One,  One ) = (One, One)
  | bitadd (_,    _,    _   ) = (Zero, One);
```

We arranged the cases to minimize complexity. The idea is to first consider the cases in which there are two or more zeroes and then the case of all ones. The remaining cases all have exactly two ones, and the result is the same for all of them.

The function to add list-based natural numbers follows the usual elementary school algorithm. We proceed from the low-order bits to the high-order bits, propagating the carry as we go. To actually add two numbers, we would invoke `nnadd` with the first argument being `Zero`.

```
fun nnadd Zero nil     nil     = nil
  | nnadd One  nil     nil     = [One]
  | nnadd c    nil     vl      = nnadd c [Zero] vl
  | nnadd c    ul      nil     = nnadd c ul [Zero]
  | nnadd c    (u::us) (v::vs) =
       let
           val (sum, carry) = bitadd(c, u, v);
       in
           sum :: (nnadd carry us vs)
       end;
```

Practice Problem 51. Write `bitsub` and `nnsub`. How does one handle potentially negative results?

Converting between list-based natural numbers and ordinary integers is an exercise in list and numerical recursion.

```
fun nnToInt nil        = 0
  | nnToInt (Zero::us) = nnToInt us * 2
  | nnToInt (One::us)  = nnToInt us * 2 + 1;

exception NNNegative;
fun intToNn k =
    case Int.sign k of
         ~1 => raise NNNegative
       |  0 => nil
       |  1 => let
                  val first = case k mod 2 of
                                    0 => Zero
                                  | _ => One;
                  val rest = intToNn (k div 2);
               in
                  first :: rest
               end;
```

Practice Problem 52. Write `nnToInt` in a different form, using an accumulator.

For multiplication, we again follow the elementary school algorithm. It is easy because the times table in binary has only two values. When multiplying by a single bit, we either get zero or the number being multiplied.

```
fun nnmult nil _          = nil
  | nnmult _   nil        = nil
  | nnmult ul  (Zero::vs) = Zero :: (nnmult ul vs)
```

```
    | nnmult ul  (One::vs)  = nnadd ul (Zero :: (nnmult ul vs));
```

This code, although correct, is misleading. If we were using a different representation—say decimal instead of binary—we would have to multiply ul by a single-digit value that may not be zero or one. How do we generalize?

Comparisons are a little involved, but they are made easier by the built-in type order. There are three values, LESS, EQUAL, and GREATER in type order. We begin with a function compare. The idea is to start out assuming that the two values are equal and read the list from right to left. At each step, we compare two bits and pass the result on to the next step.

```
fun compare t  nil        nil        = t
  | compare t  (Zero::us) nil        = compare t us nil
  | compare _  (One::us)  nil        = GREATER
  | compare t  nil        (Zero::vs) = compare t nil vs
  | compare _  nil        (One::vs)  = LESS
  | compare t  (Zero::us) (Zero::vs) = compare t us vs
  | compare t  (Zero::us) (One::vs)  = compare LESS us vs
  | compare t  (One::us)  (Zero::vs) = compare GREATER us vs
  | compare t  (One::us)  (One::vs)  = compare t us vs
```

Having compare, we can easily write the more natural comparison functions.

```
fun nnless u v = compare EQUAL u v = LESS;
fun nnleq  u v = compare EQUAL u v <> GREATER;
```

The other comparison functions, nnequal, nnneq, nngreater, and nngeq are declared in a similar fashion.

We are now ready to turn to division. Long division is relatively easy in binary because there are only two possible quotient bits. Consider long division as a recursive process, where the recursion is done on the dividend, otherwise known as the numerator. To divide the dividend n::ns, we obtain a quotient q and remainder r for the recursive case for the dividend ns. There are two cases as we try to divide the divisor d into n::r.

- If d <= n::r, then the new quotient bit is One and the new remainder is nnsub (n::r) d.
- If n::r < d, then the new quotient bit is Zero and the new remainder is n::r.

These considerations guide us in writing nnquorem, which returns a pair composed of the quotient and the remainder.

```
fun nnquorem nil     d = (nil,nil)
  | nnquorem (n::ns) d =
       let
           val (rquo, rrem) = nnquorem ns d;
       in
           if nnless (n::rrem) d
              then (Zero::rquo, n::rrem)
              else (One::rquo,  nnsub (n::rrem) d)
       end;
```

Practice Problem 53. *Shifting* is a common operation on binary quantities. A *left shift* adds zeroes on the least significant end of the bits. ("Left" refers to the usual way we read numbers, with the least significant bits on the right.) From the point of view of natural numbers, adding a zero corresponds to multiplying by two, just as adding a zero to a decimal number corresponds to multiplying by ten. Write a function `leftShift` that takes a non-negative integer `k` and a list of bits and shifts the bits left by `k`. What ambiguities are there in the specification of `leftShift`?

Practice Problem 54. An *arithmetic right shift* by `k` adds `k` copies of the most significant bit to the most significant end. Write a function `rightShift`.

---

14 Adventures in Arithmetic: Digits

In the previous section, we carried out arithmetic using the binary representation. We now develop a version for the decimal representation, with an eye toward extending our methods to other bases.

A decimal number is a list of *digits*, starting with the least significant one. A digit, of course, is an integral value between 0 and 9. We could define an SML datatype for digits, just as we did for bits, but it ends up being too cumbersome. The function `digitAdd`, for example, would have one hundred cases. Instead, we take a short cut and use ordinary integers for digits and rely on the computer's arithmetic for operations on digits. We must, without help from SML's type system, insure that our lists contain only values between 0 and 9.

The function for decimal addition is a direct generalization of the one for binary.

```
fun digitadd (c, u, v) = let
                             val total = c + u + v;
                         in
                             (total mod 10, total div 10)
                         end;
fun decadd 0 nil     nil     = nil
```

```
  | decadd c nil     nil     = [c]
  | decadd c nil     vl      = decadd c [Zero] vl
  | decadd c ul      nil     = decadd c ul [Zero]
  | decadd c (u::us) (v::vs) =
      let
          val (sum, carry) = digitadd(c, u, v);
      in
          sum :: (decadd carry us vs)
      end;
```

Practice Problem 55. Write digitSub and decsub.

Practice Problem 56. Write the conversion functions intToDec and decToInt.

As mentioned earlier, multiplication becomes a little more difficult when we leave binary representations. We must consider multiplying by a single-digit that is not zero or one. There are three functions. The first multiplies two single digits (and adds a carry digit), the second multiplies a single digit across a list (and propagates a carry digit), and the third collects the results of the list and single-digit products. Notice that the recursion in the second function is done on the lefthand argument, while the recursion in the third function is done on the righthand argument.

```
fun digitmul (c, u, v) = let
                             val prod = c + u * v;
                         in
                             (prod mod 10, prod div 10)
                         end;

fun decmulAux c nil     dig = if c = 0 then nil else [c]
  | decmulAux c (u::us) dig =
      let
          val (prod, carry) = digitmul(c, u, dig);
      in
          prod :: (decmulAux carry, us, dig)
      end;

fun decmul _   nil    = nil
  | decmul ul (v::vs) =
      let
          val partialprod = decmulAux c ul v;
      in
```

```
          decdd (decmulAux 0 ul v) (0 :: (decmul ul vs))
      end;
```

Except for division, which is covered in the next section, we have all the arithmetic functions for natural numbers represented as lists of digits, it is straightforward to generalize the operations to other representations. One commonly used representation in computer science is *hexadecimal,* base 16. The "digits" are the numbers 0 through 15, which are usually written as 0 through 9 followed by A through F.

Practice Problem 57. Write comparison functions for decimal representations. You may use the built-in function `Int.compare` to compare single digits.

Practice Problem 58. Write the hexadecimal functions `hexadd`, `hexsub`, `intToHex`, `hexToInt`, `hexCompare`, and `hexmul`.

One advantage of using lists to represent integers is that we avoid an upper bound on integers. In SML, the largest available `int` is $2^{30} - 1 = 1073741823$. In Java, it is $2^{31} - 1 = 2147483647$. While those values may seem large, modern cryptography requires integers that are *much* larger. During the course, we will develop a program that represents integers using a base of 2^{28}.

15 Adventures in Arithmetic: Long Division

In this section, we spend some time translating the elementary school algorithm for long division to our representation of natural numbers as lists of digits. In contrast to the division function in Section 9 which used iterated subtraction, the method here is more efficient and closer to the one that computers actually use.

You probably did not recognize it in elementary school, but the method of long division is recursive. Consider the following partially-completed calculation.

$$\begin{array}{r|l}
\multicolumn{1}{r}{} & 015 \\ \cline{2-2}
34 & 5214 \\
\multicolumn{1}{r}{} & 34 \\ \cline{2-2}
\multicolumn{1}{r}{} & 181 \\
\multicolumn{1}{r}{} & 170 \\ \cline{2-2}
\multicolumn{1}{r}{} & 11
\end{array}$$

At this point, we have divided 34 into 521, obtaining a quotient of 15 and a remainder of 11. It has taken us three cycles to get here, one for each digit in the quotient—including the leading 0. In the next cycle, we will append the digit 4 to the remainder to obtain 114 and divide that number by 34. That process will give us a new digit in the quotient and a new remainder.

We know that the next digit in our example is 3 and the new remainder is 12, but let us take some time to think about how we obtain those values. We will talk about dividing the *divisor* into the *dividend* and obtain a *quotient* and a *remainder*. In our example above, we have divided 34 into 521 to obtain a quotient of 015 and a remainder of 11. The process is to append the next digit 4 of the dividend to the remainder 11, obtaining 114. We then divide 34 into 114 to obtain the next digit 3 of the quotient and the new remainder of 12.

$$\begin{array}{r|r}
\multicolumn{1}{r}{} & 0153 \\ \hline
34 & 5214 \\
\multicolumn{1}{r}{} & \ddots\;\; \\ \cline{2-2}
\multicolumn{1}{r}{} & 114 \\
\multicolumn{1}{r}{} & 102 \\ \cline{2-2}
\multicolumn{1}{r}{} & 12
\end{array}$$

When we divide 114 by 34 and discover that the new digit is 3, the new remainder is given by

$$114 - 3 \times 34 = 12.$$

More generally, the *current-dividend* is obtained by appending the digit 4 to the current remainder of 11. If the next digit is d, then the new remainder is

$$\text{remainder} = \text{current-dividend} - d \times \text{divisor}.$$

The digit d is chosen to be as large as possible while the remainder is still non-negative. In other words, we want to find the largest digit d with the property that

$$\text{divisor} \times d \leq \text{current-dividend}.$$

The new remainder can be computed with just a multiplication and a subtraction. The hard part is finding the new digit in the quotient. Let us leave that problem aside for a moment and fit the rest into our list based structure. We are doing induction on the dividend, represented as a list. As previously, our function returns a pair with the quotient and the remainder.

```
fun decquorem nil     dl = (nil, nil)
  | decquorem (n::ns) dl =
        let
            val (ql, rl) = decquorem ns dl;
            val qd = newQuotientDigit (n::rl) dl;
            val newr = decsub (n::rl) (decmul dl [qd]);
        in
```

```
            (qd::ql, newr)
        end;
```

Before reading further, be sure that you understand the connection between the long division example and the SML code.

Now, what about `newQuotientDigit`? We know that d is a single digit. Our choices are 0 through 9, so it would not be terrible if we just tried all ten values. But because we want to apply our technique to representations other than decimal, we adopt a more general approach—binary search. The idea is that we start out knowing that our desired digit is at least 0 and less than 10. We take the midpoint 5 of those two values and see if the desired digit is either at least 0 and less than 5, or else at least 5 and less than 10. We have then cut the number of possibilities in half, and we can repeat the process until we have zeroed in on the answer.

Suppose that we know that the desired digit d satisfies low $\leq d <$ high, and that mid = (low + high) div 2. Then,

- if divisor $\times$ mid $\leq$ current-dividend, then mid $\leq d <$ high, and
- if divisor $\times$ mid $>$ current-dividend, then low $\leq d <$ mid.

With all this preparation, it is not difficult to translate our reasoning into SML code.

```
fun newQuotientDigit low high dl nl =
    if low + 1 = high
       then low
       else let
                 val mid = (low + high) div 2;
             in
                 if decleq (decmul dl [mid]) nl
                    then newQuotientDigit mid high dl nl
                    else newQuotientDigit low mid  dl nl
             end;
```

The function `newQuotientDigit` is recursive on the value `high` $-$ `low`. We know that `low` $\leq$ `high` always holds, and when $1 <$ `high` $-$ `low`, we know that `low` $<$ `mid` $<$ `high`. (Why?) This verifies that our recursive calls are always made on a simpler case. The base case occurs when `high` $-$ `low` $= 1$ and there is only one possible value.

Our division function has to be modified a tiny bit because the function `newQuotientDigit` must carry along the values `low` and `high`.

```
fun decquorem nil     dl = (nil, nil)
  | decquorem (n::ns) dl =
        let
        val (ql, rl) = decquorem ns dl;
        val qd = newQuotientDigit 0 10 (n::rl) dl;
        val newr = decsub (n::rl) (decmul dl [qd]);
    in
        (qd::ql, newr)
    end;
```

Practice Problem 59. Explain how we know that the result of `newQuotientDigit` is always a single digit.

Practice Problem 60. The *hexadecimal* representation uses base 16. The "digits" are 0 through 9, followed by A through F. Write the function `hexquorem`.

---

16 Lazy Datatypes

SML is an *eager* language, in the sense that it evaluates all the arguments to a function before evaluating the function itself. In some cases, an argument may not be necessary and the time spend evaluating it is wasted. This section discusses a technique to delay computation until a value is needed. (Actually, SML has sophisticated facilities for creating lazy datatypes and functions. The simple explanation here is designed to convey the general idea. See Practice Problem 65 for a pointer to the "industrial strength laziness" built into our implementation of SML.)

The basic idea is that instead of creating a value, we construct a function that computes the argument and invoke the function only when necessary. The domain of the function is the one-element type `unit`.

Lists are among the simplest structures in SML, so we look at a lazy version of lists. If lists were not already present in SML, here is how we might add them.

```
exception ListEmpty;

datatype 'a list = Nil | Cons of 'a * 'a list;

fun first Nil          = raise ListEmpty
  | first (Cons(x,_)) = x;

fun rest Nil           = raise ListEmpty
  | rest (Cons(_,xs)) = xs;
```

Notice that we do not have to define a function `cons` for constructing lists; it comes along with the datatype. The word `Cons` (with an uppercase C) is a *constructor* for the datatype and can be used in pattern-matching.

For our lazy lists, we want to avoid computing the rest of a list until we absolutely have to, and so we make the "rest" a function whose domain is `unit`.

```
exception LazyEmpty;

datatype 'a lazy_list =
    LazyNil
  | LazyCons of 'a * (unit -> 'a lazy_list);

fun lazyFirst LazyNil          = raise LazyEmpty
  | lazyFirst (LazyCons(x, _)) = x;

fun lazyRest LazyNil            = raise LazyEmpty
  | lazyRest (LazyCons(_, xs)) = xs();
```

Notice that the only real difference, outside of the type declaration, is that we must evaluate `xs` in `rest`. We could, if we wanted, make the "first" element of the list lazy as well, but the biggest gain comes from being able to avoid computing elements in a long list. If we wanted to be exquisitely lazy, we could even make the entire structure lazy—not just its components.

Here is an example of a lazy list. It is the list of natural numbers—all of them! Because we can wait to compute the "rest of the list," we can have infinitely long lists.

```
fun natRest k () = LazyCons(k, natRest(k + 1));
val naturals = natRest 0 ();
```

All of the familiar list functions have lazy analogs. For example, we can create a lazy version of `map`.

```
fun lazyMap _ LazyNil           = LazyNil
  | lazyMap f (LazyCons(x, xs)) = LazyCons(f x, ???);
```

The only question is how to handle the "rest" part of the result, indicated above with question marks. In analogy with the normal `map`, we would write

```
lazyMap f xs
```

but `xs` is a function and not a lazy list. We could make it a lazy list by evaluating it

```
lazyMap f (xs())
```

but that would defeat the purpose of laziness. It would also give us a problem with the type of the second argument to LazyCons, which must be a function. Our third try is to make it a function.

```
fn () => lazyMap f (xs())
```

That works, and we notice that the result is simply the composition of the two functions lazyMap f and xs. Here is our final version of lazyMap.

```
fun lazyMap _ LazyNil            = LazyNil
  | lazyMap f (LazyCons(x, xs)) =
       LazyCons(f x, (lazyMap f) o xs);
```

We apply the same reasoning to filter and obtain a lazy version of that function.

```
fun lazyFilter _    LazyNil           = LazyNil
  | lazyFilter pred (LazyCons(x, xs)) =
       if pred x
          then LazyCons(x, (lazyFilter pred) o xs)
          else lazyFilter pred (xs());
```

Notice that when pred x is false, we are forced to evaluate xs. With lazyFilter, we run the risk that we will never find an element for which the predicate is true, and the function will run forever. Laziness has its costs!

Here are some functions that convert lazy lists to ordinary ones. The first may run forever, so we introduce the second one with an explicit bound on the number of elements.

```
fun lazyToList LazyNil           = nil
  | lazyToList (LazyCons(x, xs)) = x::(lazyToList(xs()));

fun lazyFirstN _ LazyNil           = nil
  | lazyFirstN n (LazyCons(x, xs)) =
       if n <= 0
          then nil
          else x::(lazyFirstN (n-1) (xs()));
```

Practice Problem 61. Write an expression for the (infinite) lazy list of Fibonacci numbers.

The sieve of Eratosthenes provides an illuminating example. One creates a list of primes by starting with a list of all the natural numbers from 2 on out. We mark each element of the list as a prime by circling it or as a non-prime by crossing it out. At each stage, the first unmarked number on the list is a prime; we circle it. Then, we cross out all the multiples of that number, for they are not primes. Starting with a lazy list of numbers from 2, we use a filter-like construction to remove the multiples.

```
fun natRest k () = LazyCons(k, natRest(k + 1));

fun notMult u v = v mod u <> 0;

fun sieve LazyNil          = LazyNil
  | sieve (LazyCons(u,us)) =
        LazyCons(u, fn () => sieve (lazyFilter (notMult u) (us())));

val primes = sieve (natRest 2 ());
```

The sieve example shows the power of lazy lists, but from a practical point of view, it is inefficient because of the way the filter functions stack up. It runs surprisingly fast at first but then slows down. In one trial, it got all the primes up through one million in less than ten minutes but then took over four hours to get the primes through five million.

Practice Problem 62. Write an SML function `listTolazy` that converts ordinary lists to lazy lists.

Practice Problem 63. The *Newton-Raphson method* for computing the square root of a starts with the approximation $x_0 = 1.0$. It improves an approximation x_n by computing $x_{n+1} = (x_n/a + x_n)/2.0$. The method stops with value x_n when $|x_n - x_{n-1}|$ is less than some preset tolerance.

a. Declare an SML function `iterate` that has the type signature

```
('a -> 'a) -> 'a -> 'a lazy_list
```

and creates the lazy list corresponding to the sequence $x, f(x), f(f(x)), \ldots.$

b. Declare an SML function `inTolerance` that takes a real value `epsilon` and a lazy list corresponding to the sequence $x_0, x_1, x_2, \ldots$, and returns the first element x_n satisfying $|x_n - x_{n-1}| <$ `epsilon`.

c. Using `iterate` and `inTolerance`, define a function that takes two reals, `epsilon` and `a`, and produces the square root of `a` to within tolerance `epsilon`.

Practice Problem 64. Students in calculus courses learn that the exponential function can be approximated with Taylor's series.

$$e^x = \frac{1.0}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots$$

a. Declare a function that takes a value `x` and creates a lazy list of the individual terms in the series.

b. Declare another function that takes a lazy list of terms and produces a lazy list of partial sums of the series.

c. Declare a function that computes the exponential function to within a given tolerance. Use the `toTolerance` function from the previous problem.

Practice Problem 65. Read about the lazy extension to Standard ML of New Jersey in Chapter 15 of Harper's book, *Programming in Standard ML*. Translate the examples of this section into that framework.

17 Mutual Recursion

In SML, an identifier must be declared before it is used. That makes it difficult to define two functions which rely on each other.

Consider, for example, the problem of writing two functions `take` and `skip`.[3] The function `take` operates on a list and returns a list containing every other element of the given list, starting with the first. The function `skip` returns the elements that `take` omits; that is, `skip` returns a list with every other element, starting with the second. It is natural to want to declare them simultaneously.

```
fun take nil     = nil
  | take (x::xs) = x :: (skip xs);
fun skip nil     = nil
  | skip (x::xs) = take xs;
```

But these declarations will lead to an "unbound variable" error because `skip` is used in `take` before it is declared. Reversing the order of the two functions does not help.

Fortunately, the designers of SML have given us a way out of the predicament. The keyword `and` allows the declarations to be made at the same time. (Do not confuse `and` with the boolean connective `andalso`.) The following example is *one* declaration of *two* functions. Notice that there is only one semicolon.

```
fun take nil     = nil
  | take (x::xs) = x :: (skip xs)
and skip nil     = nil
  | skip (x::xs) = take xs;
```

[3]This example is taken from Ullman, *Elements of ML Programming,* second edition, pages 60–61.

Mutually recursive functions (and datatypes; SML allows those too!) are common in computer science. Most grammars that describe the syntactic structure of programming languages give rise to mutually recursive functions. We shall see an example toward the end of the course.

Practice Problem 66. There is another way to solve the take-skip problem, without using mutual recursion. What is it?

Practice Problem 67. Declare the predicates `even` and `odd` in a mutually recursive fashion.

Practice Problem 68. In *Gödel, Escher, Bach: An Eternal Golden Braid,* Douglas Hofstadter presents female and male sequences, defined mathematically on the natural numbers as follows.

$$\begin{array}{llll} F(0) & = 1 & \text{and} & F(n) = n - M(F(n-1)) \\ M(0) & = 0 & \text{and} & M(n) = n - F(M(n-1)) \end{array}$$

Write mutually recursive SML functions to compute the two functions.
