This document is intended as an introduction to SML for people with prior programming experience. The specific variant of SML we will be using is SML-NJ (Standard ML of New Jersey), which we assume you already have installed. Emacs (with SML mode) is **highly** recommended for editing your code. We will also assume that you are familiar with emacs—if not, you can find a variety of excellent tutorials and references on the internet.

Contents

1 Getting Started

SML is a strongly typed functional language, with parametric polymorphism. However, unlike many strongly typed langauges that you might be familiar with, SML *infers* the types of all variables—you do not (typically) have to write them down yourself. The type it infers is the most general type—that is, whenever polymorphism is safe, SML will infer a polymorphic type for any variable or function.

SML is a functional language—computation (typically) is comprised of functions applied to values. These functions return new values, which are then passed to other functions. Some of these values might themselves be functions, which can be passed as arguments to, or returned from other functions. However, SML is not *purely* functional: it provides ways for side-effecting computations if you so desire.

SML-NJ has a "read-eval-print-loop" (REPL). When you start SML (either by running `sml` at the command line, or by doing `M-x sml-mode` in emacs), you will get a banner stating the version, and then a prompt that is a single dash (`-`). If you type a valid SML expression at this prompt, SML will evaluate it and print out the value.

The "canonical" first program to write in any language is "Hello World." At the SML prompt, type

```
"Hello World";
```

and hit enter. SML will evaluate this expression, and print the result:

```
val it = "Hello World" : string
```

The string literal Hello World is already a value, so SML simply evaluates it to itself. It then binds it to the variable `it`, since you did not ask for it to be bound to anything else in particular (thats what the `val it =` part is). It also reports the type of its answer, in this case a string.

Note that the semi-colon at the end of the line is here to tell the REPL that we are done typing input—it is not used to end every statement. If you forget to put the semi-colon when writing at the REPL, you will get an `=` prompt, which indicates that SML is expecting you to continue what you were typing. If you simply forgot the semi-colon, you can just type it and hit enter.

At this point you may argue that the canonical first programming excercise is to *print* Hello World, not to simply evaluate the expression. We can do that in SML as well, by using the `print` function. Type the following at your SML prompt:

```
print "Hello World\n";
```

Note that we added a `\n` to the end of the string—as with many programming languages, `\n` is the escape sequence for a newline character. When you evaluate this expression—which passes the string `"Hello World\n"` to the print function, SML will print:

```
Hello World
val it = () : unit
```

The first line of this output is the result of the call to print. The second line of the output is the value that the expression evaluated to. In this case, the value is () (which is pronounced unit) of type unit. The unit type—which has only one value: unit—is the type used to indicate no meaningful value, much like void in C or Java. It is typically used in side-effecting computations (print is a side effecting computation: it has the side effect of outputting a string, but computes no meaningful value).

Note that one subtle detail which may confuse novice-SML programmers: `it` did not change types or values in the above example—there are two *different* variables named `it`. The first one has type string and value `"Hello World"`, the second one has type unit and value unit. This may seem strange and subtle, but it is important. Once you bind a value to a variable, that binding never changes. You can have other variables later of the same name, but cannot change what has already been bound. If you re-bind a name to a new value, subsequent uses of that name will reference the new binding. However, anything between the first and second binding will continue to refer to the original binding.

When writing larger pieces of code, you will not want to type it all in at the REPL—any mistake or changes will be a pain to edit. You can tell SML to load a file with the `use` command: `use "myfile.sml"`. Later on, we will see how to use the Compilation Manager to compile and load a large set of files, but for now, either enter short pieces of code at the REPL, or type longer pieces of code into a file, and use `use`.

2 Basic: Math, Conditionals, Functions, and Tuples

Printing strings is useful, but most computation revolves around math. SML supports integers (of type int) as well as floating point numbers (of type real). If you type `3` as your SML REPL, SML will report that it evaluates to 3, and is of type int. Likewise, if you evaluate 3.14, SML will evaluate it to 3.14, and tell you that it is of type real.

SML has *infix* math operators for addition (+), subtraction (-), and multiplication (*) for both integers and reals. You can experiment with expressions like `2+4`, `3.14 * 6.2`, etc.. and it will evaluate them. These operators all have the standard precedence and associativity that you expect. SML also has division, however, for real numbers it is the / operator, but for integers, the operator is called `div` (it is still infix: `99 div 11`). Integers also support `mod`, which is another infix operator.

Note that the +,-, and * operators are overloaded to either take two ints and return an int, or to take two reals and return a real. This overloading is a special case that is built into the langauge for a handful of mathematical operators. You cannot, in general do your own operator overloading like this. Also, note that you cannot "mix and match" ints and reals with these operators: `3 * 6.2` is ill-typed—you must either explicitly convert the int to a real, or the real to an int before operating on them. We will see how to do such a conversion later.

2.1 Conditionals

SML has a type `bool`, with two values: `true` and `false`. It also has the standard comparison operators: `<`, `<=`, `>`, `>=`, `=` (equal to), and `<>` (not equals). There is no need to have different operators (e.g. `=` vs `==`) to distinguish assignment from comparison, as they cannot appear in the same gramatical contexts. SML also has `not` (which negates a boolean), `andalso` (which performance a logical and), and `orelse` (which performs a logical or).

SML has two ways to make conditional decisions (although, as we shall see briefly, one is actually just short-hand for the other). Both styles are conditional *expressions*, not *statements*—they evaluate to a value. If you are familiar with C or Java, think of the `cond ? expr1 :expr2` construct.

The first way to write a conditional expression is with if-then-else. In SML, since this is a conditional expression, there is *always* an else—anything else is a parse error. The syntax of if-then-else is

```
 if expr then expr else expr
```

where the first expression (conditional test) must have type bool, and the other two expressions must have the same type. For example:

```
- if 3 < 4 then "hello" else "goodbye";
val it = "hello" : string
```

The second way to write a conditional is the case expression. The case expression is much more general than if-than-else, and much more powerful than the switch-case construct of C or Java. The SML case expression has the syntax

```
case expr of
  pattern => expr
| pattern => expr
| pattern => expr
 ...
```

The case expression matches the first expression against each pattern, and evaluates to the expression on the right-hand side of the matching pattern. We will learn more about pattern matching as we introduce other data types/structures throughout the document. For now, we will suffice to say that any constant can be used as a pattern, and that `_` can be used to indicate "match anything." For example, if x is an int, then we could have

```
case x of
   0 => "its zero"
 | 1 => "its one"
 | _ => "something else"
```

Note that all cases must result in the same type.

A bit of trivia: if-then-else is typically converted to case inside the compiler: `if e1 then e2 else e3` is simply

```
case e1 of
   true  => e2
 | false => e3
```

2.2 Functions

Now that we have some basic building blocks, we are ready to start writing some functions. We will start with a very simple example function:

```
fun add3 x = x + 3
```

The keyword `fun` tells SML that we are defining a function. After that comes the name of the function we are defining (add3) followed by the name of its argument (x), then an equal sign, and an expression for the function's body. Note that in SML all functions take *exactly* one argument and return *exactly* one value (this may sound restrictive, but we will see shortly that its really no problem at all).

If we evaluate the above function at the REPL, it will report:

```
val add3 = fn : int -> int
```

Here, SML is telling us that it has bound a value to the name add3. The value is represented as simply `fn`, indicating that the value is a function. The type is reported as `int -> int` (read "int to int"). The arrow indicates a function type. The type on the left side of the arrow is the argument type, and the type on the right side of the arrow is the return type.

SML infered the types of add3 and x for us. It sees that x is being added to 3, so x must be an int. Knowing that x must be an int, and that x is the argument to add3, SML figures out that add3 must take an int as an argument. It figures out that add3 must also return an int because the body (x+3) has type int.

If we want to use the add3 function with an argument of 4, we write:

```
add3 4
```

To a C or Java programmer, this may seem odd: the function call lacks parenthesis. In SML, writing two expressions next to each other means "function call". Parenthesis are only needed for order of operations/binding, or for tuples (which we will discuss momentarily).

We can write more sophisticated and interesting functions using recursion. For example we can write the factorial function:

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

Try entering the factorial function at your REPL, and then try calling it with a few values.

2.3 Tuples and pattern matching

An alternative way to write factorial is with *tail recursion*, however, at first glance, this seems problematic: tail recursive factorial appears to require two function arguments, but SML only allows one argument for any function. This restriction is not problematic, as the argument (and return value) may be *tuples*—pairs, triplets, quadruplets, etc... of data.

A tuple is written in parenthesis with commas separating its elements, for example `(3,5)` or `(7,true,"hello")`. The first of these has type `int * int` (e.g., a pair of integers), and the second of these has type `int * bool * string`—a three tuple with an int, a bool, and a string. In *types* `*` indicates a tuple, in *expressions*, `*` represents multiplicatin. Note that () is actually a tuple with zero elements, and as mentioned before has a special name: unit.

There are functions to access the individual elements of tuples, but they are hardly ever used. Instead, tuples are frequently deconstructed by *pattern matching*—one of the nicest features of SML. The easiest way to explain a pattern match is to start with an example:

```
fun fact_tail t = case t of
                    (0,ans) => ans
                  | (x,ans) => fact_tail (x-1, ans * x)
```

Here, `fact_tail` takes a single argument (t), which is a tuple (specifically, int * int). The body of the function uses `case` to match on t and pull it apart. The first case has a pattern that matches a tuple where the first element is 0. In the spot for the second element, there is a variable name, so this variable is bound to the second element of the tuple. The right hand side of the `=>` is the expression to evaluate to when this case is matched. The second case, started by the `|`, matches any two element tuple—both elements have variables for their patterns. When this pattern is matched, x is bound to the first element of the tuple, and ans is bound to the second element of the tuple.

Functions with pattern matching are so common that SML allows a short hand, where the pattern match is written directly as the function argument:

```
fun fact_tail (0,ans) = ans
  | fact_tail (x,ans) = fact_tail(x-1, ans * x);
```

Once you are used to the pattern matching syntax, you will love it. It is incredibly readable, and easy to write.

2.4 Let

One last annoying feature remains about our tail-recursive implementation of factorial: what we wrote is really a "helper" function. We want to compute the factorial of x, but need to pass in (x,1). It would be very nice if we could package up the helper inside a function that just takes an int, and hide the helper from the rest of the program. Fortunately, we can do this with `let`, an expression which allows us to make local declarations. For our factorial function, we can write this:

```
fun fact x =
    let fun help (0,ans) = ans
  | help (x,ans) = help(x-1, x * ans)
    in
help(x,1)
    end
```

Here the body of the fact function is a let expression. The let expression allows us to bind one or more names (functions with `fun` or values with `val`), and then evaluate an expression in the scope of those bindings. The scope of all of those name bindings is limited to the let expression. Multiple bindings are possible, each of which can see all preceeding bindings in the same let (for those familiar with Scheme, SML's let is like Scheme's let*):

```
fun complexFunction (x,y,z) =
    let val a = f(x,y,32)
        val b = g(z,12,false)
        fun help1(0,ans) = ans
          | help1(n,ans) = help(n-1, if x mod b = 0
                                     then ans + a
                                     else ans = 2 * z)
        val c = help1 (y*z, b)
        fun test n = x < a andalso y > z
    in
        if test b
        then c + 1
        else c - 1
    end
```

The above (contrived) example assumes there are previously defined functions `f` and `g`, which is uses to compute the values for `a` and `b` respectively.

Earlier, we discusses that names can be re-bound, but not changed. Now that we have introduced let, we can see a concrete example of this in action:

```
fun silly x =
  let val a = x
      fun addA b = a + b
      val a = x + 1
  in
     addA 3
  end
```

While this is a rather contrived example, you can play with it at the REPL to see that the addA function sees the original binding to `a` (that is, the value of `x`), not the subsequent re-binding of `a` (to `x+1`).

Let also allows for single-case pattern matching with vals. One use of this would be to pull apart tuples returned by functions (or evaluated from other expressions).

```
fun f x = (x+1, x*x)

fun g y =
    let val a     = y * y + 3
        val (b,c) = f a
    in
        b+c
    end
```

Becoming familiar with `let` will likely help you aclimate to SML much more quickly. It is very convient for writing large, complex functions, and lets you write in a slightly more familiar "assign a value to a variable, then use it to assign a value to the next variable" style. Just remember that once you bind the variables, they never change.

2.5 Mutual Recursion

Mutually recursive functions are quite handy in a variety of circumstances. In SML, they can be defined with the `and` keyword (not to be confused with `andalso`). For mutually recursive functions the functions must be defined consecutively (no other definitions in between them, and the second and later functions in a mutually recursive group use `and` instead of `fun`:

```
fun isEven 0 = true
  | isEven x = not (isOdd (x-1))
and isOdd  0 = false
  | isOdd  x = not (isEven (x-1))
```

3 Data types and data structures

We have already seen one simple way to aggregate data: tuples. Tuples let us combined multiple pieces of data together into one piece of data, with an appropriate type: an int and a string are paired together to form an `int * string` tuple. SML provides other ways to aggregate data, including the ability to create your own (potentially recursive) data types.

3.1 Records

Tuples are great for a few items (2–4), or maybe a few more if they are just being used as function parameters/return values. With very large numbers of fields, tuples can become cumbersome, as it may become difficult to remember which item is which.

SML also supports records with named fields. The syntax for a record is `{name=value, name=value, ...}`. For example:

```
{x=3, y=17, z=4}
```

is an expression which creates a record with three fields (x,y, and z) having values 3,17, and 4 (respectively). The type of this record is

```
{x:int, y:int, z:int}
```

Records can be pulled apart either with pattern matching, or with field selection functions. Field selection functions are named with a `#` and the field name, for example `#x`, `#y`, etc. For example:

```
let val r = {x=3, y=17, z=4}
  in
     #x r
  end
```

Pattern matching with records can be done in a couple ways. In one way, the field names are used as the variables to be bound to:

```
fun f {x,y,z} = x + y * z
```

However, this does not work if one wishes to match on two records with the same field name, or to specify constraints on the field values. One can also write the pattern for each field after an equals sign:

```
fun f {x=0, y=1, z} = z
  | f {x=0, y  , z} = y+z
  | f {x  , y  , z} = x * (y+z)

fun g ({x=a,y=b}, {x=c,y=d}) = a*c + b *d
```

Sometimes for records with many fields, it becomes cumbersome to write down all of the field names in a pattern match, especially if you only care about a couple. SML has the concept of *flex records*, where you are allowed to write ... to mean "there are more fields I am not naming," however, SML must still be able to figure out exactly what type the record is when it runs its type inference algorithm. Typically this means that either another case fully specify the record fields, or the programmer explcitly annotates the type (in at least one case).

For example, in the following function, the first two cases leave out z and q with ..., but the last case specifies all of the record fields, allowing SML to infer the correct type:

```
fun f {x=0,y  ,...} = y
  | f {x  ,y=0,...} = x
  | f {x  ,y  ,z,q} = x*z + y*q
```

For the other approach (naming the record type) to be useful, we need to introduce the fact that you can give types names:

```
type myRec = {x:string,
              y:int,
              z:bool,
              a:int,
              b:int,
              c:int}
fun f ({z=false,...}:myRec) = 0
  | f {a,y, ...} = a + y
```

There are two things to note from the previous example. First, SML does not care about the order of record fields, only their names. a,b is the same as b,a. Second, parenthesis are required around `({z=false,...}:myRec)`, to cause the type annotation to bind to the record, and not the function result type. If you write `f (a:t)` you are saying that and `a` has type `t`. If you say `f a:t` you are saying that `f` returns type `t`.

Tuples are actually a special case of records—they have fields named 1, 2, 3... You can see this with the following code fragment (which appear ill-typed, unless you know that tuples are actually records):

```
let val {1=x,2=y} = (3,4)
in
 x + y
end
```

Using this fact in the above ways is not terribly useful, and rather discouraged (as its counter-intuitive, and thus confusing). However, there is one useful application of this fact: the functions `#1`, `#2`, etc can be used to pull individual fields out of tuples when desired. As mentioned earlier, pattern matching is much more common, and typically preferred, however, this is possible when desired.

3.2 Lists

Tuples and records are great if you want to combine a fixed number of elements. However, frequently, programmers need dynamic data structures to accomodate varying sized storage. The most prevalent data structure in SML is the list. List is a "type constructor"—that is, "list" is not a type by itself, but given another type it constructs a type. For example, "int list" or "string list" are types. Since "int list" is a type, and "list" is a type constructor, "int list list" is also a valid type—a list of lists of ints.

In SML, we can make lists in two ways. First, we can write list literals with square brackets: `[1,2,3]` is a list of three ints (1, 2, and 3 in that order). An example of a function that makes a list using this syntax would be:

```
fun f x =
  let val a = x * 2
      val b = x * 3
   in
     [0, x, a, b, x * 4]
   end
```

This function has type `int -> int list` (that is, it takes an int, and returns a list of ints). Think for a second about what it does, and then try it out at the REPL and see if you understand it.

The other way to construct lists is with the `::` operator. If you are familiar with Scheme or LISP, `::` works much like `cons` does in those langauges. If you are familiar with C or Java, think about linked list "add to front", but remember that we are making a new list, not changing an old one.

We could re-write our previous example using `::` (in fact, this is what the compiler translates our list literal into):

```
fun f x =
  let val a = x * 2
      val b = x * 3
   in
     0::x::a::b::(x*4)::[]
   end
```

SML allows for pattern matching on lists using either syntax. For example, the following function computes the length of a list. The first pattern match uses the [] syntax for the empty list, and the second uses :: to pull match a non-empty list whose first item we do not care about (remember that _ means "dont care") and whose rest is bound to `l`.

```
fun len []     = 0
  | len (_::l) = 1 + len l
```

You can also write things like the following function, which have pattern matches for the empty list, a one item list (binding the single item to `a`, a two item list (binding the two items to `a` and `b`), and any longer list (binding the first to `a` and the rest to `l`.

```
fun f []     = 0
  | f [a]    = a+1
  | f [a,b]  = a * b
  | f (a::l) = a + f l
```

3.3 Polymorphism: flexibility in your types

Consider the following code to operate on a list:

```
fun insLst (x,[]) = [x]
|   insLst (x,a::l) = a::x::l
```

This function takes a pair with a "thing" and a list of "things"—but what type of thing? We could do

```
insLst (4,[1,2,3])
```

or

```
insLst ("x",["y","z","q"])
```

or

```
insLst ([1,2,3],[[4,5,6],[],[9,7,3]])
```

In the first, insLst has type `int * int list -> int list`, in the second `string * string list -> string list`, in the third, `int list * int list list -> int list list`. This seems problematic since SML must assign `insLst` *exactly* one type when it type checks the function, but it seems to have three different types.

What actually happens here is that SML assigns `insLst` a *polymorphic* type—specifically `'a * 'a list -> 'a`. This polymorphic type can be instantiated differently at each use of `insLst`—replacing `'a` with any valid type (int, string, and int list above).

The types of functions may have multiple type variables, for example:

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

Here, `f` has type `'a * 'b -> 'b * 'a` It can be instantiated as `int * string -> string * int`, `int * int -> int * int`, `int list * int -> int * int list` etc. Note that when SML infers the type of a function, the algorithm it uses is guaranteed to find the most general type of that function.

Polymorphic functions and data-types are quite powerful, as we will see shortly.

SML provides one special case of polymorphism: any type, but where equality comparisons are possible. Such types are called "eqtypes" and are represented with two ticks instead of one: `''a`. For example in the contains function:

```
fun contains (_,[]) = false
  | contains (x,a::l) = x = a orelse contains(x,l)
```

It works on any type of list, as long as we can compare an element of that list to something of the same type and see if they are equal. This function has type `''a * ''a list -> bool` (SML/NJ will also warn you that you are "calling polyEqual" which is not a big deal, and you can safely ignore the warning).

This function will work on `int * int list`, since ints can be compared for equality, but will not work int `( (int -> int) * (int -> int) list)` since functions cannot be compared for equality.

3.4 Datatypes : defining your own

SML allows the user to define his/her own `datatypes`, which may be recursive (or mutually recursive) and/or polymorphic.

We have actually already seen one datatype, which is builtin: the list.

Datatypes are defined in terms of their *constructors*, for example:

```
datatype foo =
   X of int
 | Y of string * bool
 | Z
```

Here, we define a new datatype (foo), with three constructors (X, Y, and Z). X has type `int -> foo`, Y has type `string * bool -> foo` and `Z` has type `foo`. foo is now a type just like any other: we can have foo lists, foo * int pairs, etc.

This means that X 3, Y("hello",false), and Z are all expressions of type foo.

Just like any other type, we can pattern match on foos, which is useful to pull them apart:

```
fun myFun (X x) = x
  | myFun (Y(s,b)) = if b then size s else 42
  | myFun Z = 0
```

When we define the datatype this way, SML knows not only that these constructors are the ways to create a foo, it knows that these are the *only* ways to create a foo. This lets SML determine if the match is exhasutive and/or redundant.

We can define the datatypes to be recursive by using the name of the newly defined type in the argument lists of one or more of the constructors:

```
datatype file =
   Directory of string * file list
 | NormalFile of string * int * string
```

Just like functions, datatypes can be made mutually recursive using `and`:

```
datatype a =
   A of int * string
 | B of bool * b
and b =
   C of int * int list
 | D of a list
```

Datatypes may be made polymorphic like this:

```
datatype 'a tree =
   NODE of 'a * 'a tree * 'a tree
 | LEAF
```

You can specify two ticks to require and eqtype:

```
datatype ''a tree =
   NODE of ''a * ''a tree * ''a tree
 | LEAF
```

Note that lists are defined as

```
datatype 'a list =
  :: of 'a * 'a list
| nil
```

and that `::` is made into an infix operator (there is also special syntax for list literals that you cannot create for your own data types).

Another really useful data type is the option type:

```
datatype 'a option =
   SOME of 'a
 | NONE
```

An int option for example, lets you return SOME(an int) to indicate a valid int answer, or NONE to indicate “no answer”.

4 Interlude: Compiler Errors

One major difficulty that novice SML programmers often face is understanding and correcting compiler errors. This difficulty tends to arise from the fact that the compiler works differently in many ways than typical compilers for most other languages like C or Java.

4.1 Parse Errors

One major difference between SML and compilers that you might be used to is the way that SML handles syntax errors. SML's parser uses an error correction algorithm where it attempts to insert, remove, or change lexical tokens to give a correct gramatically correct parse. This scheme results in SML reporting parse error messages in terms of what it ended up changing to get a correct parse. For example, suppse you write this:

```
let val x = [1,2,3
  in
   x
  end
```

SML will report the following error message:

```
temp.sml:2.2 Error: syntax error: inserting  RBRACKET
```

This indicates that the best solution that the parser was able to come up with was to insert a "RBRACKET" token (a right square bracket) to get a correct parse. Note that this is the error message you get if you load the file with `use`. If you type the same code in at the REPL, you get a confusing error message:

```
stdIn:34.2-36.5 Error: syntax error: deleting  IN ID END
```

Where the parser decides to drop `in x end` in the hopes of finding more correct tokens later (*e.g.,* you continue typing `,4]`. If you are getting errors with your code, it is often better to save it in a file and `use` that file, to get better error messages.

Another thing to understand about the error correction algorithm is that the parser chooses a fix that matches the grammar. There may be many options, and it has no way of trying to pick the "right one," even when only one makes sense for the program. Consider the following program, which has a mis-match between an left square bracket and a right parenthesis:

```
let val x = [1,2,3)
 in
 4::x
 end
```

SML's parser will report the following correction:

```
temp.sml:1.13 Error: syntax error: replacing  LBRACKET with  LPAREN
```

That is, it decided to try replacing the left square bracket with a left parenthesis:

```
let val x = (1,2,3)
 in
 4::x
 end
```

This program is gramatically correct, but ill-typed. The parser is not smart enough to realize that this is the wrong choice, and it should replace the RPAREN with an RBRACKET.

When you first start having parse errors, the reference to specific token names may not be terribly information. However, as you write and compile more code, you will become accustomed to them. Here are some common token names to get you started out:

Token Name	Meaning
(any keyword)	(That keyword)
LPAREN/RPAREN	Left and right parenthesis:`()`
LBRACKET/RBRACKET	Left and right square brackets: `[]`
LBRACE/RBRACE	Left and right curly braces:`{}`
ARROW	An arrow, as in a function type: `->`
DARROW	A "double" arrow, as in a case: `=>`
ID	An identifier `x`, `fact`, etc..
WILD	The "wildcard" or "dont care" pattern: `_`
VECTORSTART	The start of a vector literal: `#[`
(other punctuation name)	self-explanatory by name
EOF	End of file

4.2 Type Errors

There are two aspects of SML's type errors which are confusing to new SML programmers. First, SML may infer the incorrect type for a variable from an erroneous use, then report type errors on the correct use. Consider this function:

```
fun f x =
    let val y = "My input value is " ^ x ^ "\n"
val () = print y
val z = x * 3
    in
z + (x div 4)
    end
```

The programmer likely intended for `x` to be an `int`, and the actual error is that `x` is concatenated with strings without a conversion (*i.e.,* `Int.toString x`). However, SML infers the type for `x` as `string`, and then reports errors where user of a string is not valid. In more complex code, it may help to explicitly annotate a type, e.g.:

```
fun f (x:int) =
    let val y = "My input value is " ^ x ^ "\n"
val () = print y
val z = x * 3
    in
z + (x div 4)
    end
```

to help find the actual error. This annotate forces SML to use `int` as the type for `x`, causing it to detect the correct error.

Note that `f x : int` annotates the return type of `f` as an int, and says nothing about `x`.

The other confusion aspect of type errors is the terminology that SML uses. For example, the following error:

```
temp.sml:3.6-3.14 Error: operator and operand don't agree [literal]
  operator domain: string list
  operand:         int
  in expression:
    someFun x
```

sounds somewhat complex, but is really saying "I expected a list of strings (string list), but have a thing that is an int—thats not right".

Here, "operator" is refering to the function hd, and the operand is x. The operator domain is the type that the operator (hd) expects (a list of things) and the operand (x) has type int.

Another term that new SML programmers are unfamiliar with is "tycon mismatch"—this stands for "type constructor mismatch" and happens when SML's type inference algorithm needs a type made from two incompatible type constructors (that is, one thing is used in incompatible ways, which would require a type constructed from two different things). For example, this code:

```
fun f x =
    let val (a,b) = x
        val q = a + b + 3
val [c,d,e,4] = x
    in
q + c + d - e
    end
```

produces the error:

```
temp.sml:4.6-4.19 Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    int list
  expression:    int * int
  in declaration:
    c :: d :: e :: <pat> :: <pat> = x
```

SML infers the type of `x` as `int * int`—a pair of ints, but has run into problems when trying to reconcile that with pulling x apart by a pattern match into a list of ints. Here, SML is complaining that the type for x would have to be made from two different (and incompatible) type constructors: one the one hand it needs to be made from the "*" constructor (int * int) and on the other, from the "list" constructor (int list).

4.3 Other Errors

For help on other errors, see `http://www.smlnj.org/doc/errors.html`

5 Practice Excercises 1

Programming and programming langauges are not something you can learn by reading. You also need to practice. Before continuing, it is useful to stop here and work some practice problems. Write each of these functions, and test them out at the REPL.

1. Write the `min` function which has type `int * int -> int` and returns the smaller of its two arguments.
2. Write the `fib` function which has type `int -> int` and computes the nth Fibonacci number.
3. Write the `isPrime` function which has type `int -> bool` and determines whether or not its argument is prime.
4. Write the function `sumList` which has type `int list ->int` and computes the sum of all of the items in the list.
5. Write the function `squareList` which has type `int list -> int list` and returns a list of the same length as its argument, where each item in the returned list is the square of the corresponding item in the argument list. For example, `squareList [2,5,1,3]` should return `[4,25,1,9]`

6 Higher Order Functions

SML allows functions to be treated just like any other value: they can be passed as arguments to other functions, returned from other functions, put in lists, trees, or other data structures. Pretty much the only thing you cannot do with functions is test them for equality (such a test is undecidable).

Why would we want to do such a thing? One thing we might want to do is have a generic "max" function which allows us to pass in different ways to compare things (especially so that it can take different types:

```
fun max gt =
  let fun lp curr [] = curr
        | lp curr (a::l) = if gt(a,curr)
                           then lp a l
                           else lp curr l
in
   lp
end
```

This `max` function exhibits both taking a function as a parameter (`gt`) and returning another function (`lp`) as its return value.

If we want to do a "normal" max on int lists, we could do:

`val maxInt = max (op >)`.

Note that `op >` is the syntax to take an infix operator (`>`) and get the function value assiocaited with it. Here, `maxInt` has type `int -> int list -> int`. That means it is a function which takes an int, and return an `int list -> int` function. Suppose we know that all our int lists are non-empty and contain positive values. We could do

`val f = maxInt 0` to get an `int list -> int` function that will use the normal "greater than" ordering on lists. We can use `f` on a bunch of different int lists, and it will work fine.

We could also pass our `max` function different orderings on ints, or even comparison functions for different types.

We might improve our `max` function slightly to use options, so that we don't have to pass the start value in from outside:

```
fun max gt [] = NONE
  | max gt (a::l) =
    let fun lp curr [] = curr
          | lp curr (a::l) = if gt(a,curr)
                             then lp a l
                             else lp curr l
   in
     SOME(lp a l)
   end
```

This function is a bit less kludgy, since it handles the corner case of the empty list correctly, no matter what the code that uses it does (and by returning an option, it forces that code to deal with the answer of "there was nothing there").

The function has the type

```
('a * 'a -> bool) ->
   'a list -> 'a option
```

That is, you give it a comparison function, and it gives you back a function which takes a list and returns an option with SOME of the largest element, or NONE for the empty list.

So now, `val maxInt = max (op >)` gives us a `maxInt` function which has type `int list -> int option`.

Note that we can write anonymous functions with the keyword `fn`. Suppose we want to examine a list in such a way that all even numbers are treated as greater than all odd numbers. We could do

```
val weirdMaxInt =
  max (fn (x,y) =>
         if x mod 2 <> y mod 2
         then x mod 2 = 0
         else x > y)
```

Here we make an anonymous comparison function (we don't give it a name) which takes a pair of integer (x,y), and returns a boolean as we want.

Higher order functions and lists are so ubiquitous in SML that it comes with three higher order functions to process lists that you should learn to love: fold, map, and filter. fold comes in two flavors foldl and foldr.

6.1 map

We'll start with map, which has type `('a -> 'b) -> 'a list -> 'b list`. What this says is "you give me a function that transforms 'a s into 'b s, and I will give you a function that transforms lists of 'a s into lists of 'b s".

For example, `map (fn x => x -1)` is a function which takes a list of ints and returns a list of ints where all values are one smaller. map is basically written like this:

```
fun map f [] = []
  | map f (a::l) = (f a)::(map f l)
```

6.2 filter

filter has type `('a -> bool) -> 'a list -> 'a list`. This basically says "give me a function that says to keep something or throw it away, and I'll give you a function that filters the list according to that rule".

For example, `filter (fn x => x mod 2 = 0)` is a function that filters lists keeping all even numbers and discarding all odd numbers. filter basically looks like this:

```
fun filter f [] = []
  | filter f (a::l) = if f a
                      then a::(filter f l)
                      else filter f l
```

6.3 fold

Both versions of fold (foldl and foldr) have type:

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

This type is the most complex, but these functions are also the most powerful. Here, 'a is the type of things in the list, and 'b is the type of the answer at each step. You pass fold a function which takes an item from the list (a 'a) and the current answer-so-far (a 'b) and the function returns the newest answer-so-far. You then pass fold the starting answer-so-far. It returns a function which performs this operation on a list and gives you your answers.

For example, here is a way to write the `sum` function to sum a list of integers: `val sum = foldl (op +) 0` The function to do at each step is + (add: type `int * int -> int\verb`). And we start at 0 as our answer-so-far. The resulting function will sum any list of integers.

We can also re-write our earlier max function:

```
fun max gt [] = NONE
  | max gt (a::l) =
    let fun pick(a,b) = if gt(a,b)
                        then a
                        else b
   in
    SOME(foldl pick a l)
   end
```

Note that once you are accustomed to the functions, using them makes your code imminently readable: you don't have to slog through the basic list processing, you just look at "what are you doing" and "where are you starting".

The two variants of fold (foldl vs foldr) pick the order in which the list is processed. For communative side-effect-free operations like addition, either one works, however, for some things, order matters.

foldl works left-to-right. That is, it applies the function to the starting answer and the first element of the list. Then takes the resulting answer and uses it with the second element of the list, and so on. foldl is basically:

```
fun foldl f ans [] = ans
  | foldl f ans (a::l) = foldl f (f(a,ans)) l
```

foldr works right-to-left. It applies the function to the starting answer and the *last* element of the list, then works backwards. That is, foldr is basically

```
fun foldr f ans [] = ans
  | foldr f ans (a::l) = f(a,foldr f ans l)
```

For those of you who like to think in terms of imperative algorithms, you should think of fold as being a "for loop" over all elements of the list. That is, if you want to write

```
currentState = init;
for ( x=list.begin(); x!= list.end(); x++ ) {
     currentState = f (x, currentState);
  }
return currentState;
```

You should just write `foldl f init list`. If you want to iterate in reverse, use `foldr f init list`.

Note that foldl, foldr, and map are all available directly in the top level environment (you can just use their names and get the right function). filter is not, but is accessible as List.filter (more on structures later).

7 Side-effects

With the exception of printing, the code we have seen so far has been purely functional—nothing is ever modified, instead a new different thing is created. Applying map to a list does not modify the list—it makes a new list with the mapped elements.

7.1 Type constructors: ref + array

There are, however, two type constructors which create side-effectable types: ref and array. The following code shows an example of an int ref in action:

```
let val r = ref 0
    val x = r
    val () = r := 42
in
   !x
end
```

Here, `r` and `x` both have type `int ref`. They both reference the same int (much like two pointers to the same integer). `r := 42` changes the values referenced by r to be 42. This is an expression which has unit type (recall that unit is the name for a zero-element tuple and is the type for things that produce no meaningful value, but are done for side-effect). The body of the `let` uses the `!` operator to de-reference the ref `x`, getting the value it references, which is 42.

Arrays work somewhat similarly, and primarily use the following functions:

```
Array.sub : 'a array * int -> 'a
Array.update : 'a array * int * 'a -> unit
```

See `http://www.standardml.org/Basis/array.html` for more details on arrays if you need them.

7.2 Applications

Note that we can do interesting things with refs and higher-order functions:

```
fun makeCounter initVal =
  let val r = ref initVal
  in
   fn () => let val ans = !r
                val () = r := ans + 1
            in
              ans
            end
  end
```

This function has type `int -> unit -> int`. You give it an initial value, and it returns a function which will count up every time you apply it to unit. This code will result in `a` being 3, `b` being 4, and `c` being 5.

```
val f = makeCounter 3
val a = f ()
val b = f ()
val c = f ()
```

As extra random bonus trivia, note that you can emulate OOP with refs and records:

```
type pt = { getX : unit -> int,
            getY : unit -> int,
            setX : int -> unit,
            setY : int -> unit}
fun makePoint (x,y) =
  let val xr = ref x
      val yr = ref y
in
   {getX = fn () => !xr,
    getY = fn () => !yr,
    setX = fn x' => xr := x',
    setY = fn y' => yr := y'}
end
```

Here, makePoint acts as a constructor (in the OOP sense) for "objects" of type pt, which encapsulate the data and the functions that act on them. We actually hide the data here, and nobody can directly access the refs.

7.3 The value restriction

One caveat of side-effects is that, if done naively, they make polymorphic types unsafe (see `http://mlton.org/ValueRestriction` for an example). The important consequence of this is that ML has a rule called the "value restriction." This may come up sometimes when you use polymorphic functions, especially partial applications of fold (that is, you give fold a few arguments to get the function you want—which is sort of "waiting" for the rest of the arguments to fold"). For example, consider the following use of `foldl` to reverse a list:

```
val reverse = foldl (op ::) []
```

Evaluating this results in

```
stdIn:109.5-109.31 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val reverse = fn : ?.X1 list -> ?.X1 list
```

Attempting to apply the reverse function to any list will result in a type error—to be safe (ensure nobody did anything bad with refs), SML had to make it not polymorphic, and gave it a monomorphic type on "dummy types"—types that aren't meaningful. This made the reverse function, as written here, pretty useless since we can't apply it to much.

However, all is not lost. If we wanted a monomorphic function, we could simply specify the type:

```
val reverse: int list -> int list = foldl (op ::) []
```

but this is not the best option—we can actually get a polymorphic funciton this way, we just have to write it a bit differently:

```
fun reverse x = foldl (op ::) [] x
```

This seems almost exactly the same (and works the same) as the original (potentially problematic) solution—the differences are subtle, and why it guarantees safety is too complex to go into here. The important thing to know is what to do if you get that sort of error message.

8 Practice Excercises 2

Before proceeding, perform the following coding excercises.

1. Using the following datatype declaration:

   ```
   datatype expr =
      NUM of int
    | PLUS of expr * expr
    | MINUS of expr * expr
    | TIMES of expr * expr
    | DIV of expr * expr
    | F of expr list * (int list -> int)
   ```

   Write the function `eval: expr -> int` which evaluates an expression to a value. NUM(x) simply evaluates to x. PLUS, MINUS, TIMES, and DIV should recursively evaluate their sub-expressions, then perform the appropriate math operations. F exprs should recursively evaluate all the exprs in their expr list, then apply their function to the resulting integers. (Hint: You should use `map` in your F case.)

2. Use fold to write the `flatten: 'a list list -> 'a list` function. The flatten function merges together a list of lists into a single list with all of the elements in their original order. For example: `flatten [[1,2,3], [4], [5,6], [], [7]]` should result in the list `[1,2,3,4,5,6,7]`.

3. Use fold to implement your own version of map.
4. Use fold to implement your own version of filter.
5. Use fold to write the function `count: ('a -> bool) -> 'a list -> int` which returns a count of how many items in the list the `'a -> bool` function returned true for.
6. Use fold to write `mapPartial: ('a -> 'b option) -> 'a list -> 'b list` this function is like a combination of map and filter. For each item of the input list, if the function applied to that item returns SOME(b), then b appears in the output list. If NONE is returned, that item is dropped from the output list.

9 The module system

So far, our SML discussion has covered *programming in the small*—writing individual functions, or a few functions to do relatively minor tasks. SML has a rich *module system* to support *programming in the large*. There are three pieces to this module system—structures, signatures, and functors—which we will discuss over the next three sections.

9.1 Structures

An SML *structure* forms a module, grouping together related types, constants, and functions into a coherent package.

Consider the following example of the MyBST structure, which defines a datatype and two operations (add and find) for binary search trees:

```
structure MyBST =
struct
 datatype 'a bst =
  NODE of int * 'a * 'a bst * 'a bst
| NIL
 fun add (k,v,NIL) = NODE(k,v,NIL, NIL)
   | add (k,v,NODE (k', v', l,r)) =  if (k < k')
     then NODE(k', v', add(k,v,l), r)
     else NODE(k', v', l, add(k,v,r))
 fun find (k,NIL) = NONE
   | find (k,NODE(k',v,l,r)) = case Int.compare (k,k') of
   EQUAL => SOME v
 | LESS => find(k,l)
 | GREATER => find(k,r)
end
```

Most of the code in this example should be familar from what we have seen so far: a polymorphic datatype declaration, two functions written with pattern matches, an if expression, a case expression, and a few function calls. There are, however, two new things. The first is the declaration of the structure itself. `structure MyBST =` tells SML that we are declaring a structure (called

MyBST). After this we can either follow it with a new structure definition (`struct .... end`) or with the name of an existing structure.

The other new construct that we see here is the use of the `Int.compare` function. This shows how to use something found in another structure. Here, Int is the name of an existing structure (its built into SML, and provides a variety of functions to do things to integers). The dot gives us access to something inside the Int structure, namely the `compare` function. This feature is useful, as there may be many compare functions in a large program. In fact, just in the built-in SML library, there are several compare functions, for example: String.compare, Int.compare, Real.compare.

As we mentioned above, you can declare a structure to be equal to an existing structure. This is particularly useful to give a structure a shorter name if you frequently reference it. For example, if you are writing another structure that uses MyBST frequently, you might say `structure B=MyBST` at the start of that module. You can then say `B.find` and `B.add` to save typing.

You can also `open` a module—e.g., `open MyBST`, which lets you reference the contents of the module without prefixing it with the name (i.e. just `find` instead of `MyBST.find`). While this also saves typing, it can lead to less readable code: if you open many structures, someone reading the code many have to search through all of them to find where a function is declared.

9.2 Signatures

Our binary search tree example from the previous section is a specific case of a more general construct: the Map abstract data type (where we look up values given a key). We could also envision other implementations of the Map abstract data type: lists, balanced BSTs, Hashtables,etc... Signatures provide a way to define an interface without defining an implementation. They force structures to conform to that interface, and force users of that structure to only use those types and values (including functions) specified in the interface. This is particularly useful for code maintainablilty, as you are always guaranteed you can replace change out one implementation of a structure for another conforming to the same signature with no other code changes.

Consider the following example signature:

```
signature MyMap =
sig
    type 'a map
    val empty: 'a map
    val add:   int * 'a * 'a map -> 'a map
    val find:  int * 'a map -> 'a option
end
```

This signature specifies that a structure conforming to the MyMap signature must contain one type and three values (2 of which are functions because they have an arrow type). However, it says nothing about *how* these functions are implemented.

With a few slight modifications, we can specify that our MyBST structure conforms to the MyMap signature (called *ascribing* the signature to the structure):

```
structure MyBST : MyMap =
struct
 datatype 'a map =
  NODE of int * 'a * 'a map * 'a map
```

```
| NIL
 val empty = NIL

 fun add (k,v,NIL) = NODE(k,v,NIL, NIL)
   | add (k,v,NODE (k', v', l,r)) =  if (k < k')
     then NODE(k', v', add(k,v,l), r)
     else NODE(k', v', l, add(k,v,r))
 fun find (k,NIL) = NONE
   | find (k,NODE(k',v,l,r)) = case Int.compare(k,k') of
   EQUAL => SOME v
 | LESS => find(k,l)
 | GREATER => find(k,r)
end
```

Note that we have made three changes here:

1. The declaration of MyBST now has `: MyMap`, saying that it conforms to the MyMap signature
2. We have changed the type name from bst to map, since the signature requires a type called map (we could also keep the datatype declaration the same, and then add this declaration `type 'a map = 'a bst`).
3. We have added the declaration `val empty = NIL` so that we have the empty value required by the signature.

The compiler will check that the MyBST structure has all the names required by the signature, and that they all have the proper types. The structure may define additional names (types or values), but these effectively become private when the signature is ascribed.

There are actually two types of ascription. The one we just saw (using the `:` syntax) is called *transparent*. The other type (using `:>` instead of `:`) is called opaque. The difference between the two is whether or not the actual types used in the structure are externally visible. Both types of ascription hide all names that are not declared in the signature.

The difference between the two types of ascription are hard to see with the MyBST example, so we will consider the following different implementation of the MyMap signature:

```
structure MyLstMap : MyMap =
struct
 type 'a map = (int * 'a) list
 val empty = []
 fun add(k,v,lst) = (k,v)::lst
 fun find(k,[]) = NONE
   | find(k,(k',v)::l) = if (k=k')
 then SOME v
 else find (k,l)
end
```

This implementation uses lists. With transparent ascription, however, our information hiding is incomplete: the fact that 'a map is a (int * 'a) list is externally visible. This fact allows an external module to write something that "cheats" on the interface to the module like this:

```
(1,2)::MyLstMap.empty : int MyLstMap.map
```

If we do such a thing, then we cannot seamlessly swap out one map implementation for another. If we use opaque ascription (shown below), then the fact that 'a map is a list is not externally visible, and the information is completely hidden.

```
structure MyLstMap :> MyMap =
struct
 type 'a map = (int * 'a) list
 val empty = []
 fun add(k,v,lst) = (k,v)::lst
 fun find(k,[]) = NONE
   | find(k,(k',v)::l) = if (k=k')
 then SOME v
 else find (k,l)
end
```

It is generally best to use opaque ascription, though there are times when transparent is appropriate.

9.3 Built-in Structures

SML comes with a wide set of built-in structures called the "Basis Library." You can find documentation on them here:

`http://www.standardml.org/Basis/overview.html`

Ones of particular note are List (various useful things to do on/with lists), ListPair (things to do to/with pairs of lists), Int (functions to work with ints), String (for string operations/manipulations), Array (as mentioned above: mutable arrays), and Vector (like arrays, but immutable—not like Java Vectors: they do not resize).

You can also find things like TextIO (reading/write text to/from files), OS (and its substructures—for a wide variety of system calls), and many more things.

9.4 Using the Compilation Manager

SML/NJ comes with a utility that is similar to `make`, but customize for SML. This utilitiy is called the "Compilation Manager," or CM. When working on any program of significant size, use of the compilation manager is highly recommended.

Using the compilation manager most efficiently goes hand-in-hand with organizing your source code properly. Place each structure definition in its own `.sml` file (do the same for each functor definition—discussed in the next section). Place each signature definition in its own `.sig` file. Then write a `.cm` file for your program. This `.cm` file should start with the line `Group is`, followed by a blank line. It should then list all of the source files (one per line). Order does not matter. It should also list `$/basis.cm` to include the basis library. Frequently, you will also want `$/smlnj-lib.cm` for the utility library (which contains a lot of algorithms and data structures).

An example `.cm` file might be

```
Group is
```

```
$/basis.cm
$/smlnj-lib.cm
foo.sig
foo.sml
bar.sig
bar.sml
main.sml
```

Once you have your `.cm` file created, you can compile your program, and (if compilation was successful) load the name bindings it introduces into your current repl with `CM.make "filename.cm"` where `filename.cm` is replaced with whatever you named the file. If the compilation was successful the last lines of output (before the new REPL prompt) should be

```
[New bindings added.]
val it = true : bool
```

If `CM.make` returns false, it indicates that a compilation error occured, and you should be able to find that error in the output.

If you are using ml-lex, you should be able to just list `.lex` files in your `.cm` file. CM will know to run ml-lex on these to get sml source, and then compile the resulting SML file. If you are using ml-yacc, you can list `.grm` files in your `.cm` file, but also need to list `$/ml-yacc-lib.cm` to include the ml-yacc library. CM will know to run ml-yacc on the `.grm` file to generate sml, and compile the resulting file.

9.5 Functors

Our previous Map design is a good step in the right direction: we make good use of abstraction (signatures provide an interface that hides the implementation), and our Map is polymorphic in the values it stores. However, our Map has one major downside: the keys must be ints.

We can fix this, but we need some way to parameterize the data structure over the comparison function. This sounds simple enough, as we have already learned that we can pass functions to other functions.

A straw-man approach would be to use this interface:

```
signature MyMap =
sig
    type ('a,'b) map
    val empty: ('a,'b) map
    val add:   ('b * 'b -> order) -> 'b * 'a * ('a,'b) map -> ('a,'b) map
    val find:  ('b * 'b -> order) -> 'b * ('a,'b) map -> 'a option
end
```

Note that this signature makes use of the built-in type `order` which is defined as:

```
datatype order =
   GREATER
 | EQUAL
 | LESS
```

This interface works, but exhibits a design flaw that can come back to bite the programmer. A programming mistake could cause the programmer to `add` with one ordering function and `find` with another, causing mysterious errors where something should be in the map, but is not found.

A better option might be:

```
signature MyMap =
sig
    type ('a,'b) map
    val empty: ('b * 'b -> order) -> ('a,'b) map
    val add:   'b * 'a * ('a,'b) map -> ('a,'b) map
    val find:  'b * ('a,'b) map -> 'a option
end
```

In this interface, we provide the ordering function when an empty map is created, and encapsulate that ordering function in the map data structure (so the map type might be a pair of a function and a tree). This approach solves the main problem in option 1—a programmer can no longer erroneously try to `find` with a different function than was used to `add`. However, we can still do better.

Suppose we wanted to add a `merge` function to our interface:

```
val merge: ('a,'b) map * ('a, 'b) map -> ('a, 'b) map
```

When we add this function, we would like implementations to be able to support their inherent orderings for an efficient merge operation where possible. In the setup we have, this might result in something like

```
fun merge (m1, m2) =
  if (sameOrdering(m1,m2))
  then fastMerge (m1,m2)
  else slowMerge (m1,m2)
```

However, we have a problem for almost any practical implementation: functions cannot be compared for equality, so we cannot check if the orderings are the same! One can imagine a variety of kludges, but `functors` provide an elegant solution: we parameterize the *structure as a whole* over another structure (containing types and functions). Instantiating the functor with a particular structure results in a structure which the type system recognizes as distinct from other instantiations of the same functor.

To see this in action, let us first look at the built-in signature `ORD_KEY`:

```
signature ORD_KEY=
sig
  type ord_key
  val compare : (ord_key * ord_key) -> order
end
```

This does not seem terribly exciting—its just a type and a comparison function over that type—however, this is exactly what we need to parameterize our MyBST Map implementation over. To do this, let us first revisit the MyMap signature:

```
signature MyMap =
sig
  type key
  type 'a map
  val empty: 'a map
  val add: key * 'a  * 'a map -> 'a map
  val find: key * 'a map -> 'a option
end
```

Now our signature says that Maps will define two types: (1) the type of key they use (2) the type for a polymorphic map (which has the key fixed, but can be instantiated for any data type). The signature then provides similar values/functions as the original—it just uses `key` as the type of its key rather than int.

Now, we can make a generic BST map *functor* which is parameterized over a structure (K) conforming to the `ORD_KEY` signature. The resulting structure (which is a BST-based map for a particular key type) will conform to MyMap:

```
functor MyBST (K: ORD_KEY) : MyMap =
struct
 type key = K.ord_key
 datatype 'a map =
  NODE of key * 'a * 'a map * 'a map
| NIL
 val empty = NIL

 fun add (k,v,NIL) = NODE(k,v,NIL, NIL)
   | add (k,v,NODE (k', v', l,r)) =  case K.compare(k,k') of
                                       EQUAL => NODE(k,v,l,r)
                                     | LESS => NODE(k', v', add(k,v,l), r)
                                     | GREATER => NODE(k', v', l, add(k,v,r))
 fun find (k,NIL) = NONE
   | find (k,NODE(k',v,l,r)) = case K.compare(k,k') of
   EQUAL => SOME v
 | LESS => find(k,l)
 | GREATER => find(k,r)
end
```

Once we have this, we can do things like

```
structure IntBST = MyBST(struct type ord_key = int
                                val compare = Int.compare
                         end)
structure StrBST = MyBST(struct type ord_key = string
                                val compare = String.compare
                         end)
```

A few things to note about this example: (1) IntBST.map and StrBST.map are two distinct type constructors. The type `bool IntBST.map` is completely different from `bool StrBST.map`. In fact, if we do this:

```
structure IntBST = MyBST(struct type ord_key = int
                                val compare = Int.compare
                         end)
structure IntBST2 = MyBST(struct type ord_key = int
                                val compare = Int.compare
                         end)
structure IntBST3 = IntBST
```

The type constructors IntBST.map and IntBST2.map are distinct. Note that IntBST3.map is the same as IntBST.map, because IntBST3 is exactly the same structure as IntBST.

If you were paying *very* careful attention, you will notice that I used transparent ascription of the MyMap signature rather than opaque (even though I said opaque is generally better).

```
functor MyBST (K: ORD_KEY) : MyMap =
```

If we change this to opaque ascription:

```
functor MyBST (K: ORD_KEY) :> MyMap =
```

but leave everything else the same, then this works great until we actually try to use it:

```
- IntBST.add(1,"hello",IntBST.empty);
stdIn:42.1-42.35 Error: operator and operand don't agree [literal]
  operator domain: IntBST.key * 'Z * 'Z IntBST.map
  operand:         int * string * 'Y IntBST.map
  in expression:
    IntBST.add (1,"hello",IntBST.empty)
```

The problem here is that opaque ascription has not only hidden the information we want (the innerworkings of our tree data type), but has also hidden information we need: that `key` is `int`!

More generally, what we want our functor to reveal (as part of its interface) is that its `key` type will *always* be the same as K's `ord_key` type—and that its ok for the outside world to know this. We accomplish this with the `where` keyword:

```
functor MyBST (K: ORD_KEY) :> MyMap
where type key = K.ord_key =
struct
 ...
end
```

with this definition of the MyBST functor (as well as the definition of the IntBST structure above), we get an excellent design of our data structure: the BST implemenation of the map can be instantiated for any type we can give an ordering function for. The deatils of the implementation are neatly hidden behind the abstraction boundary enforced by opaque ascription, and we can still write any other MyMap implementation we want (Note that these do not have to be functors at all: we can make our original int list-based map work by just adding `type key=int` to it).

Two more notes about writing functors:

First, if you want a functor to be paramaterized over multiple structures (that is, "take multiple arguments") you can, but you do it in a way that looks like curried arguments:

```
functor F (X: Sig1) (Y: Sig2) =
struct
 ...
end
```

Second, you can use `where` in the functor's argument declaration to contrain the signature a bit more. For example, in the practice excercises you will do shortly, you will want to have a functor parameterized over a structure conforming to the `ORD_MAP` signature, but will want to contrain the key type to strings:

```
functor F(M: ORD_MAP where type Key.ord_key = string)
```

Note that Key is a sub-structore of `ORD_MAP`, so you are requiring that `M.Key.ord_key` be a string.

9.6 Built in Maps and Sets

SML's utility library provides `ORD_MAP` and `ORD_SET` signatures, as well as some efficient implementations of them.

You can read about the interfaces here:

```
http://www.smlnj.org/doc/smlnj-lib/Manual/ord-map.html#ORD_MAP:SIG:SPEC
http://www.smlnj.org/doc/smlnj-lib/Manual/ord-set.html#ORD_SET:SIG:SPEC
```

The utility library provides three implementations of each of these BinaryMapFn/BinarySetFn, SplayMapFn/SplaySetFn, and ListMapFn/ListSetFn.

10 Practice Excercises 3

For this excercise, you will be filling in the contents of the following functor:

```
functor F(M: ORD_MAP where type Key.ord_key = string)
         (S:ORD_SET where type Key.ord_key = string) :>
sig
  val proc: string list -> S.set M.map
end
=
struct
  ...
end
```

The `proc` function takes a list of strings (which are file names), and builds a map of sets. The map maps strings (i.e., words) to the set of files names in which they appear. Words are separated by spaces (or newlines). For example, if you were passed `["a.txt","b.txt"]` and a.txt had this contents:

```
Hello World
test
```

and b.txt had this contents:

```
a test
input
```

Your function should return a map that maps “Hello” and “World” to the set `{a.txt}`, “test” to the set `{a.txt, b.txt}` and “a” and “input” to the set `{b.txt}`.

Checking out the String, and TextIO structures are highly recommended for implementing this function.

You should then instantiate your functor on at least one structure instantiated from one of the built-in `ORD_MAP` and `ORD_SET` functors and test it out.

11 Common Mistakes

Here are a few common mistakes for new SML programers:

1. and vs andalso: and is for mutually recursive declarations. andalso is for the boolean and operation.

2. if then else is an expression, not a statement. The else is required, and both the then and the else must have the same type. If you want to do something for side-effect in one case, but not do something in the other case, use () as the body of the else.

3. dot is for accessing things within structures—these are not objects, they are modules. These modules separate namespaces (that is, String.compare and Int.compare are two different compares), but there is no encapsulation here. You do not do value.function.

4. `fun f a:t` vs `fun f (a:t)`. The first of these says that f returns type t. The second say that the argument to f is of type t.