SML Overview

This handout gives a concise summary of those features of Standard ML (a.k.a. SML) most useful for the first part of CS 131. It is not intended as a first tutorial in functional programming, which you should have received in earlier courses.

1 SML Philosophy

While imperative languages perform computation by creating variables and modifying their values, functional languages like SML are based on the idea of computation simply by evaluating definitions and expressions.

Every expression in SML

- Has a type such as `int` or `bool` or `int*string->int`
- May result in a value (answer) when evaluated
- May cause *side effects* when evaluated

A side effect is anything that happens during evaluation besides computing values, such as assignments, input/output, or raising of exceptions. Not all functional languages allow side effects; such languages are often called "pure" or "purely functional". SML encourages purely functional programming, but also permits side effects in those cases where they are appropriate.

In contrast to many functional languages such as Lisp, Scheme, and Rex, Standard ML is *strongly typed*. Programs are type checked before they are run, and SML is very strict about its error checking. This strictness may seem annoying at first, but is actually extremely useful — error messages nearly always reflect a real bug in your code.

The notation

$$expression \ : \ type$$

is used to state the fact that the given SML expression has the specified SML type. For example,

```
3 : int
```

reflects the fact that in SML, the integer type is written `int`, and `3` is classified as an integer.

2 Numbers

There are two main numeric types in Standard ML, `int` and `real`. The `int` type classifies signed integers[1] such as `42` and `~4`, while the `real` type classifies (double preci-

1. The language standard does not specify how large such integers can be; the SML/NJ implementation represents integers as 31-bit numbers.

sion) floating-point numbers such as `16.0`, `~14.237`, and `~1.2e~14`. Unary negation is written with a tilde, making it syntactically different from the binary subtraction operator.

The ordinary arithmetic operators work on pairs of integers or on pairs of reals, except that `div` is the operator for integer division (which truncates its result to another integer) while `/` is used for floating-point division:

```
(3 + 4 * (~6 - 18 div 4)) mod 15      :   int
6.02e~23 + 4.5 * (~6.0 - 18.3 / 4)    :   real
```

SML *never* implicitly converts values from one type to a different type, so you cannot apply the arithmetic operators to one integer and one real; you have to first explicitly convert the integer to a real (via the function `real`) or else convert the real to an integer (via the `round`, `ceil`, or `floor` functions).

3 Strings and Characters

Strings are written as you might expect, with double quotes, and have type `string`.

```
"Hello, world!\n"   :   string
```

Characters are written as a hash `#` immediately followed by a single-character string.

```
#"a"    :    char
#"\n"   :    char
```

Unlike C, characters are *not* a sort of integer, so you can't do arithmetic with characters.

The function `explode` converts a string to a list of characters, and the function `implode` does the opposite.

4 Booleans

The type `bool` contains exactly two values:

```
true  : bool
false : bool
```

The comparison operations `<`, `<=`, `>=`, and `>` take two integers, two reals, two strings, or two characters, and return a boolean result.

```
3 > 5          :  bool
#"a" <= #"z"   :  bool
```

Inequality is written as <> (rather than !=), and equality is written with a single equals sign, =. [2]

```
3 <> 5          : bool
"a" = "a"       : bool
```

The operators for negation, logical-or, and logical-and are `not`, `andalso`, and `orelse`, respectively.

```
not (3 > 5)           :    bool
(4>2) andalso (2=3)   :    bool
(3<2) orelse (5<>3)   :    bool
```

Both `andalso` and `orelse` are shortcircuit operators. If the first argument to `andalso` is false, the second argument won't be evaluated. Similarly, if the first argument to `orelse` is true, the second won't be evaluated.

Boolean values can be used by `if-then-else` or *conditional* expressions:

```
if ((3>4) orelse (5<2))  then "yes"  else "no"   :   string
(if (4 > 5) then 3 else 5) + 1                   :   int
```

These conditionals are expressions returning a value; they correspond to `... ? ... : ...` in C++ or JAVA instead of `if-then`. An `else` branch is always required, because the expression must always yield a value, even when the condition is false.

The type checker requires that the tests in conditional expressions have type `bool` and that the expressions in the `then` and `else` branches have the same type.

5 Pairs, Tuples, and Records

SML provides several mechanisms for aggregating several values together.

5.1 Pairs and Tuples

An ordered pair whose first component has type t_1 and whose second component has type t_2 has the SML type t_1*t_2. For example:

```
(3, "three")  :   int * string
(4.0, 5.0)    :   real * real
```

This can be extended to ordered triples, ordered quadruples, and so forth:

```
(1,#"3","3")  :   int * char * string
```

2. The operator = does not work for equality between two real numbers. There is a function for comparing floating-point values, `Real.==`, but because of floating-point roundoff and other imprecision, it's almost always a bad idea to directly compare two floating-point numbers for equality anyway.

The following values all have different types, however:

```
(1, 2, 3)    :   int * int * int
((1, 2), 3)  :   (int * int) * int
(1, (2, 3))  :   int * (int * int)
```

The first is a triple, while the last two are pairs (of different types). Parenthesization of types matters.

5.2 Records

If tuples have many components or if many of these components have the same type, it is easy to write code with elements in the wrong order. In such cases, a *record* may be more appropriate. A record is very similar to a tuple, except that each component has a *label* (a name). Record expressions are written with curly braces around a sequence of named components. The types of records are similar, except that they give types for each label in the record. For example, a two-dimensional point could be represented as a record with two real components labeled `x` and `y`:

```
{x = 3.14, y = ~2.0}      :    {x : real, y : real}
```

When writing a record or its type, the components can be listed in any order. Hence,

```
{name = "Pat", age = 20, occupation="student"}     :
               {occupation : string, name : string, age : int}
```

The components of a tuple or record can be accessed via *pattern-matching* (see Section 7).

6 Lists

In their most familiar form, lists are written as in Rex with square brackets and commas between elements. Unlike Rex, lists in Standard ML must be homogeneous: all elements of a list must have the same type. When every element of the list has type t, the type of the entire list is written as t `list`. For example,

```
[1,2,3,4]                      :  int list

[(0,"black"), (1,"brown"),
 (2,"red"), (3,"orange"),
 (4,"yellow"), (5,"green"),
 (6,"blue"), (7,"violet"),
 (8,"grey"), (9,"white")]      :  (int * string) list
```

Recall the inductive definition of lists from CS 60: every list is either empty, or else it consists of a single element (the "head" or "first") and another list (the "tail" or "rest"). SML lists are defined in exactly this way. The empty list is written `nil`, and given an element h and a list t we can construct a new list h `::` t whose first element is h and whose remaining elements are contained in the list t. (The infix operator `::` is pronounced "cons".)

The expression h `::` t has exactly the same meaning as the rex or Prolog expressions `[`h `|` t`]`. Be very careful not to get these notations confused. The code `[`h `::` t`]` means something very different from h `::` t in SML: the list $[h :: t]$ has length *one*, and the single item it contains is the list h `::` t.

Thus the list syntax with square brackets and commas is simply a handy abbreviation for uses of `nil` and `::`. For example,

```
[]      ≡   nil
[1,2]   ≡   (1 :: [2])    ≡   (1 :: (2 :: nil))
```

You can use either syntax, depending on what is convenient.

SML provides some useful built-in functions for lists, including `rev` for reversing lists and the infix operator `@` for appending two lists.

```
rev [1,2,3]          ⇒  [3,2,1]
[1,2,3] @ [4,5,6]    ⇒  [1,2,3,4,5,6]
```

As in most functional languages, `rev` and `@` do not modify their arguments, but instead create *new* lists!

Also, be very careful not to confuse `::` with `@`. The former prepends a *single element* onto the front of a list of such elements, while the latter combines two lists of the same type.

7 Pattern Matching

Pattern matching in SML is very similar to what you saw in Rex and Prolog. Given a value and a compatible pattern, the value may or may not *match* the pattern. SML allows many sorts of patterns:

- A variable is a pattern that matches any value. When an entire pattern matches a value, then each variable occurring in the pattern is bound to (i.e., defined to be) the corresponding part of that value. Note that matching a value against a variable, say `x`, is neither an assignment to an existing variable `x` *nor a test to see whether the value is equal to the contents of an existing variable* `x`, but rather initializes a *new* local variable named `x`. Pattern-matching against a variable always succeeds.

  Unlike Rex and Prolog, patterns in SML must be "linear": no pattern can contain the same variable multiple times.

- The *wildcard* or underscore pattern `_` matches any value, but does not define a variable; it means "don't care". Multiple wildcards may appear within a single pattern, and each can match a different value.
- A constant integer, character, string, or boolean can be used as a pattern; it matches exactly that value.
- A tuple pattern is written simply as a tuple of patterns. It matches a tuple if the components of the pattern match the corresponding components of the tuple. For example, the pattern `(3,_)` matches any pair of values whose first component is the integer `3`. The pattern `(x,y,z)` matches any triple and gives the name `x` to the first component, `y` to the second component, and `z` to the third component. Record patterns can be written similarly as a record of patterns, e.g., {`name=x,age=20,occupation=z`}, which matches records having components with those three names where the `age` field contains the integer `20`.
- List values can be matched with lists of patterns; either with the square bracket syntax as in `[x,y,z]` (which matches any list of length exactly three and binds these three elements to variables) or with the `nil` and `::` syntax, as in `(x::(y::_))`, which matches any list with length ≥ 2 and gives the names `x` and `y` to the first two elements. It is usually a good idea to put parentheses around patterns involving `::`.
- Datatype constructors may appear in patterns. (See the discussion of datatypes below.)
- Any pattern can be annotated with a type, to help guide type inference (see below) in figuring out what sort of value is being matched against. Thus the pattern `(x:int)` can be used to match integers, whereas the pattern `(x:int,y:bool)` can be used to match any `int*bool` pair, and bind the names `x` and `y` to the two components.

Pattern matching comes up in several contexts including definitions and functions (see below), but can be used directly in a `case` expression. This use is analogous to `switch` in C or C++, but the cases are specified using patterns rather than integer constants; the cases don't fall through; and the entire `case` expression returns a value. The first case whose pattern matches the item being analyzed is always chosen. For example, assuming that `lst` is a variable of type `int list`,

```
(case lst of
   [] => "lst is empty"
 | [_] => "lst has length one"
 | [3,_] => "lst has length two and starts with a three"
 | _ => "unknown")                                    :      string
```

It is usually a good idea to put parentheses around case statements as in this example—otherwise, the parser can get confused. Also, note that the vertical bar is used to separate the cases, so there's no vertical bar before the first.

Because SML is strongly typed, the type of the value being matched determines which patterns are allowed. For example, if the value being tested in a `case` expression has type `int`, then only variable, wildcard, or constant-integer patterns can appear in the cases. Similarly, the type checker will reject any `case` expression that tries to do one thing if given a boolean and another thing if given an integer, or which does one thing given a pair and another thing given a triple.

The compiler also checks for redundant matches (those which will never be picked) and inexhaustive matches (where some values might not match any of the cases). At run time, if there is no pattern that matches the value being tested then the `case` expression raises a `Match` exception.

8 Functions

Standard ML functions are *first-class* values in the same way that integers and strings are values; they can be passed as arguments to functions, can be returned from functions, can be the value of a variable, and so on. Every function in SML takes *exactly one* argument and returns *exactly one* result. The type of a function which expects an argument of type t_1 and returns a result of type t_2 is written t_1 `->` t_2.

A function value can be written `fn` *pattern* `=>` *expression*. When applied, the given argument is first matched against the pattern, and then the expression making up the body of the function is evaluated. For example,

```
fn x:int => x+1
```

is a value of type `int -> int`, namely the successor function on integers. Similarly,

```
fn (x:int,y:bool) => (y,x)
```

has type `int*bool -> bool*int` (in types, `*` binds more tightly than `->`). This value is a "swapping" function that takes a pair and returns a new pair with the components in the opposite order. This is still a one-argument function, but that argument is a pair; by using pattern-matching notation we can assign names to the two components of that single pair.

Functions are applied by "juxtaposition"; two expressions written next to each other are interpreted as function application. For example,

```
(fn x:int => x+1) 3
```

is an application of the successor function to the argument `3`; unsurprisingly, the value of this application expression is `4`.

Note that the parentheses around the function value are necessary in this case. If we ommited them and wrote

```
fn x:int => x+1 3
```

SML's parser would see it as equivalent to writing

```
fn x:int => (x + (1 3))
```

which makes no sense (you can't apply the integer `1` to another integer `3`—the SML compiler gives a type error for this kind of mistake).

Like most languages, SML allows you to add parentheses around any expression for grouping, even if those parentheses are, technically, redundant. Thus, our function application example could also be written as

```
(fn x:int => x+1) (3)
```

A more common case of application is when we have a variable, say `f`, whose value is a function. If

```
f : int -> string
```

then we can say `f 3` or `f(3)` or `(f)(3)` or `f (2+1)` to apply this function to the argument `3` and get back a `string`. Application has higher precedence than the other operators, however, so the similar-appearing expression `f 2+1` is understood to mean the ill-typed expression `(f 2) + 1`.

9 Type Inference

Functional languages work not by assigning new values to existing variables, but by allocating new variables to hold new values. With all these variables being defined, it would be relatively painful to have to specify a type for every single variable appearing in definitions or patterns. Fortunately, SML requires that implementations do *type inference*, which is the process of figuring out what the types for all of the variables need to be. Thus, if you write

```
fn x => x+1
```

the computer sees that you are adding the function argument `x` to an integer, concludes that the variable `x` must range over integers, and concludes that this expression has type `int -> int`.

10 Definitions

SML provides two ways to associate identifiers with values. `val` provides a general mechanism define variables, and can also be used to define functions. `fun` provides a more convenient syntax for defining named functions.

10.1 `val`

The general form of variable definition in SML is

`val` *pattern* = *expression*

which evaluates the expression and then matches it against the pattern to define any new variables. (If the pattern fails to match, a `Bind` exception is raised at run-time; the compiler will warn you about a non-exhaustive match if it thinks this might happen.) For example,

```
val twice          = (fn (n:int) => n+n)
val swapint        = fn (x:int,y:int) => (y,x)
val two            = twice 1
val (four, three) = swapint (two+one, twice 2)
```

Remember, these definitions are initializations of *new* variables, rather than assignments!

Because the right-hand-side is evaluated before the pattern-matching and variable definition occurs, the definition

```
val fact : int->int =
      fn n => if (n = 0)
                 then 1
              else
                 n * fact(n-1)
```

doesn't work. The function value on the right-hand-side won't type check because `fact` isn't defined yet. To specify that you want a *recursive* definition that can refer to the variable being defined you need to say `val rec` instead of `val`:

```
val rec fact : int->int =
      fn n => if (n = 0) then
                 1
              else
                 n * fact(n-1)
```

10.2 `fun`

The above definition of factorial is awfully ugly. Fortunately, SML provides an alternative syntax for `val rec`. The first two definitions above can be written

```
fun twice (n:int) = n+n
fun swapint (x:int,y:int) = (y,x)
```

and the factorial function can be written

```
fun fact(n:int) : int =
  if (n = 0) then
     1
  else
     n * fact(n-1)
```

Note that the keyword is `fun` instead of `val`, and the argument pattern appears on the left-hand-side of the equals sign. Note also that if you want to specify types (optional, due to type inference) the type of the argument and the type of the function's result are specified separately.

Definitions using `fun` are just convenient syntax for defining a variable to have a function as its value! The compiler internally converts every such function into equivalent code written using `val rec`.

10.2.1 Clausal Definitions

Even better, definitions of functions can specify multiple rules as in rex, by giving a sequence of cases (called *clauses*) separated by vertical bars. When the function is applied, the first clause whose pattern matches the argument is chosen. For example, yet another way to write factorial would be

```
fun fact 0 = 1
  | fact n = n * fact(n-1)
```

Note that the name of the function must be repeated for each case.

As another example,

```
fun sumlist [] = 0
  | sumlist (n::ns) = n + sumlist ns
```

which defines the variable

```
sumlist : int list -> int
```

as a function which sums lists of integers.

10.2.2 Curried Functions

There are two obvious ways in SML to write a function of two arguments, corresponding to two different "calling conventions". Either we can supply both the arguments together at the same time, or we can supply the arguments separately, first one and then (at some later point) the other. The former calling convention can be implemented as a function that takes a pair; for example,

```
fun add_u (x,y) = x+y
```

The latter can be implemented by taking just one of the arguments, and returning a *function* that takes the second argument and does the actual computation:

```
fun add_c x = (fn y => x+y)
```

Here

```
add_u  : int * int -> int
add_c  : int -> (int -> int)
```

and the following two expressions both result in the value 4:

```
add_u (1,3)    :     int
(add_c 1)(3)   :     int
```

The advantage of the latter form (called the *curried* form) is that we can apply the two arguments at completely different points in the program. For example, we could define

```
val succ : int->int  =   add_c 1
```

and then evaluate `succ 3` to get 4. Because curried functions are very common, SML provides special syntax; `fun` definitions may have multiple patterns separated by whitespace, representing arguments to supplied sequentially. (A very similar syntax was also available in rex.) An equivalent definition of `add_c` is thus

```
fun add_c x y = x+y
```

Curried arguments and multiple clauses can be combined. For example,

```
fun add_c' 0 y = y
  | add_c' x y = x+y
```

which defines `add_c' : int->int->int` to avoid the addition if the first argument is zero. A more interesting example is the built-in function

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

which takes (in successive applications) a function and a list, and returns the results from applying the function to every element of the list. This function could be defined as

```
fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)
```

Arrows in types are right associative, so the type of `map` could also be written with one fewer pair of parentheses as `('a -> 'b) -> ('a list) -> ('b list)`.

Deciding whether a function definition should be curried or not depends on how it is to be used, and is, to a large extent, a matter of taste.

10.2.3 Mutual Recursion

You can write mutually-recursive functions using `val rec` or `fun` by writing the definitions in sequence, using `and` instead of `val rec` or `fun` after the first. For example,

```
fun even(n) =   if (n=0) then true else odd(n-1)
and odd (n) =   if (n=1) then true else even(n-1)
```

or, equivalently,

```
fun even(n) =   (n=0) orelse odd(n-1)
and odd (n) =   (n=1) orelse even(n-1)
```

11 Local Definitions

It is often useful to define variables while inside a larger expression, such as the body of a function. The expression `let` *definitions* `in` *expression* `end` allows the creation of local variables; to evaluate a `let`-expression, first the definitions are evaluated, then the given expression is evaluated using those definitions, and finally these local definitions are discarded and the value of the body is returned. So, the expression

```
2 + let
       val fun thrice(n) = 3*n
       val (y,z) = (thrice 2, thrice 4)
    in
       thrice(y)
    end                          :    int
```

evaluates to 20.

12 Polymorphism

For some definitions, it's not obvious what the type of the defined variable should be. For example, consider

```
fun length [] = 0
  | length (_::rest) = 1 + length rest
```

This function could reasonably be applied to lists of integers, lists of strings, lists of pairs of functions, or any other sort of list. In fact, for any type t, this function could have type t `list -> int`.

SML handles this situation by allowing *type variables* in types. Such variables always have a name starting with a single quote, such as `'a` or `'b`. They are called type variables because they range over types such as `int` or `((char*bool) list) list`. Given the above definition, SML will conclude that

```
length : 'a list -> int
```

That is, for *every* type `'a`, the function `length` can be applied to a value of type `'a list` to obtain an integer. Thus we have

```
length [1,2,3]                          :   int
length [true, false]                    :   int
length [fn x=>x+1, fn x=>x, fn x=>x-1]  :   int
```

and so on.

Similarly, the built-in function `rev` has type `'a list -> 'a list` (meaning that it can be applied to a list of any type, and returns a list of the *same* type). The definition

```
fun swap(x,y) = (y,x)
```

yields

```
swap : 'a*'b -> 'b*'a
```

meaning that for any types `'a` and `'b` (possibly but not necessarily different) the `swap` function can be applied to a value of type `'a*'b` to obtain a value of type `'b*'a`. For example,

```
swap (3,true)      :      bool * int
swap (5.0, 2.0)    :      real * real
```

As one more example, the empty list `nil` has type `'a list`.

Type variables are useful in writing generic functions, but are not intended to be the final type of an evaluated expression. Expressions should, after all, produce evaluate to a value with no remaining unknowns. [3]

13 Type Abbreviations

Sometimes types in SML can get large; it can be convenient to give a short name to a big type, the way that `typedef` is used in C. This is easy to do; for example, the definition

```
type striple = string*string*string
```

says that the name `striple` can be used interchangeably with its definition, so that we can later say

```
val x : striple = ("a","b","c")
```

3. If you get a type error or warning about the *value restriction*, you have an expression, such as `rev nil`, that could be given infinitely many different types, but that isn't allowed—the final value needs to have a specific type. This problem can usually be fixed by explicitly specifying the type you want (e.g., `val lst : int list = rev nil`).

instead of

```
val x : string*string*string = ("a","b","c")
```

Type definitions can also be parameterized. For example, the definition

```
type 'a pair = 'a * 'a
```

allows us to use `int pair` interchangeably with `int*int`, and similarly to write `striple pair` instead of

`(string*string*string)*(string*string*string)`.

Type abbreviations never define new types; they can be thought of as a kind of macro. They may *not* be recursively defined.

14 Datatypes

SML `datatypes` correspond to what some other languages call "tagged unions" or "disjoint unions". The elements of a datatype are values with tags attached. For example, the definition

```
datatype intOrReal = Int  of int
                   | Real of real
```

actually defines three names:

- It defines a *new* type `intOrReal`.
- It specifies that there are two sorts of values of type `intOrReal`: such values either contain the tag `Int` and an integer value, or contain the tag `Real` and a floating-point value. These tags are called *datatype constructors*, or *constructors* for short. We can create values of this type by tagging integers or reals with the appropriate constructor:

```
Int 3                      :   intOrReal
Int (~5)                   :   intOrReal
Real 4.2                   :   intOrReal
[Int 3, Int ~5, Real 4.2]  :   intOrReal list
```

  Constructors can also be used in pattern-matching. Recalling that the function `real` converts integers to floating-point numbers, we can write

```
fun add(Int n, Int m)   =  Int(n+m)
  | add(Int n, Real r)  =  Real(real(n) + r)
  | add(Real r, Int n)  =  Real(r + real(n))
```

```
        | add(Real r, Real s) =  Real (r+s)
```

to define a function `add : intOrReal*intOrReal -> intOrReal`.

A datatype can have arbitrarily many constructors, some of which may not even need to be attached to a value. For example,

```
datatype color = Red
               | Green
               | Blue
               | RGB of {redvalue:int, bluevalue:int, greenvalue:int}
```

defines a type color whose values are either the constructor `Red` by itself, the constructor `Green` by itself, the constructor `Blue` by itself, or a three-component record tagged with the constructor `RGB`. Thus, for example

```
[Red, RGB{redvalue=255, bluevalue=255, greenvalue=0}, Blue]
                                                    :  color list
```

If none of the constructors have associated data, then we get something analogous to an "enumerated type" in other languages. For example,

```
datatype day = Monday | Tuesday | Wednesday | Thursday
             | Friday | Saturday | Sunday
```

Note, however, that the values `Monday` through `Sunday` of this new type `day` cannot be used as integers.

14.1 Recursive Datatypes

Datatypes can be recursively defined. For example, we can define a type mimicking integer lists as

```
datatype myIntList = Empty
                   | NonEmpty of (int * myIntList)
```

which says that a value of type `myIntList` is either the constant `Empty` or is tagged `NonEmpty` and has a pair containing an integer (the head) and another `myIntList` (the tail). Thus, given the definition

```
fun myIntListLength Empty = 0
  | myIntListLength (NonEmpty(_,rest)) = 1 + myIntListLength(rest)
```

that defines `myIntListLength : myIntList -> int` we have

```
myIntListLength (NonEmpty(3, NonEmpty(4, Empty)))  :  int
```

with this expression evaluating to 2.

Datatypes can be used to define many other inductively defined data structures as well (e.g., trees, logical propositions, C++ syntax).

We can write a group of mutually recursive datatype definitions using `and` in much the same way as we did for mutually recursive functions.

```
datatype expr = Plus   of expr * term
              | Minus  of expr * term
              | Term   of term
and term      = Times  of term * factor
              | Divide of term * factor
              | Factor of factor
and factor    = Int    of int
              | Expr   of expr
```

14.2 Parameterized Datatypes

Finally, datatypes can be parameterized, just as type definitions can. A particularly useful definition that is already predefined in SML is

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

Thus we have

```
NONE        :   'a option
SOME 3      :   int option
SOME true   :   bool option
SOME (3,4)  :   (int*int) option
(3, SOME 4) :   int * int option
```

The option type is useful when we may or may not have a value. For example, the function to convert a string to an integer is

```
Int.fromString  : string -> int option
```

This function returns `SOME` n if the given string contains a recognizable integer n and returns `NONE` otherwise.

The built-in type list could have been defined as

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

except that the cons operator, `::`, is written infix, that is, as $h :: t$ instead of `::`(h, t).[4]

4. If you really want to write infix operators as if they were nonfix, you can use the `op` keyword. For example, you can write `op ::`(h, t) in the above example.

15 Exceptions

Exceptions in SML generally behave the same as those you have seen in C++ and JAVA. The definition of new exceptions is reminiscent of the syntax for datatypes. For example,

```
exception Oops
```

or

```
exception BadValue of int
```

An exception is thrown by using `raise` expressions:

```
raise Oops
```

or

```
raise BadValue(~1)
```

Finally, exceptions are caught using `handle` expressions. The code

```
f(17)
   handle Oops => 0
        | BadValue(~3) => 0
        | BadValue n => n
        | _ => 42
```

returns the value of `f(17)` if `f(17)` yields a value, but returns the answer `0` if `f(17)` raises the `Oops` exception or the exception `BadValue(~3)`, and returns the answer `n` if `f(17)` raises the exception `BadValue(n)` for `n` $\neq$ `3`, and returns the answer `42` if any other exception was raised. Of course not every exception handler has to catch every possible exception. If an exception is raised but not caught, control jumps to the next enclosing exception handler (as usual for languages with exceptions).

16 Modules

Standard ML has a powerful modules system, comprised of *structures* (implementations), *signatures* (interface definitions), and *functors* ("functions" that operate at the module level, combining code and producing new code).

16.1 Structures

A `structure` is a collection of definitions (of values, types, other structures, etc.), bracketed by `struct` ... `end`. It is somewhat analogous to a package in JAVA, except that once a structure is defined you cannot go back and add new components. Here's

and example that defines an implementation of sets of integers using lists:

```
struct
  type set = int list
  val empty = []
  fun insert(x,s) = x::s
  fun member(x,[]) = false
    | member(x,h::t) = (x=h) orelse member(x,t)
end
```

16.2 Signatures

A `signature` is a specification for a structure; it contains a sequence of specifications (descriptions of defined names) bracketed by `sig` ... `end`. Here's a signature describing a generic implementation of sets of integers:

```
sig
  type set
  val empty  : set
  val insert : int * set -> set
  val member : int * set -> bool
end
```

A signature is said to "match" a structure if the structure contains definitions satisfying all the specification in the signature. A structure can contain more items than the signature and still match; all that is required is that *at least* the components of the signature be present.

16.3 Module Definitions

Signatures and structures are *not* normal values, so they can't be passed to functions, stored in tuples, and so forth. We can still give them names; we just can't use `val`:

```
structure Set = struct
                   type set = int list
                   ... as above ...
                end
signature SET = sig
                  type set
                  ... as above...
                end
```

Once we have a name for a structure, we can refer to its components by name. For example, the type `Set.set` or the value `Set.empty`.

16.4 Information Hiding

We can use a signature to hide everything about a structure except what is explicitly stated in a signature it matches by using the `:>` operator. Thus, after the definitions

```
signature SET = sig
                  type set
                  val empty  : set
                  val insert : int * set -> set
                  val member : int * set -> bool
                end
structure Set :> SET = struct
                          type set = int list
                          val empty = []
                          val insert(x,s) = x::s
                          fun member(x,[]) = false
                            | member(x,h::t) = (x=h) orelse member(x,t)
                        end
```

the type `Set.set` is treated as *abstract*; we don't get to use the fact that sets are implemented as lists because we can't tell this from the `SET` signature. Thus we know that `Set.empty` has type `Set.set`, and can still write `Set.add(3, Set.empty)` but the expression `3 :: Set.empty` will be rejected by the type checker. This ensures that we could change the implementation of sets (to use ordered binary trees, for example) and know that all code using the `Set` module would still work.

17 The Rest of SML

Features not described in this document include references and assignment, input–output, and functors (parameterized structures).

A Appendix: Some Built-In Functions

Below are some useful functions provided in Standard ML. There are other useful functions in the built-in structures `Int`, `Real`, `Char`, `String`, `List`, `Bool`, and `Math`. (See the course web page for the signatures of these structures and the rest of the *Standard Basis* library.)

A.1 Type Conversions

```
real           :  int -> real
round          :  real -> int
ceil           :  real -> int
trunc          :  real -> int
chr            :  int -> char
ord            :  char -> int
explode        :  string -> char list
implode        :  char list -> string
concat         :  string list -> string
Int.toString   :  int -> string
Real.toString  :  real -> string
Bool.toString  :  bool -> string
Char.toString  :  char -> string
```

A.2 Higher-Order Functions

```
map  :  ('a -> 'b) -> ('a list) -> ('b list)     (curried)
o    :  ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)    function composition (infix)
```

A.3 Miscellaneous

```
size    :  string -> int
^       :  string*string -> string            string append (infix)
length  :  'a list -> int
@       :  'a list * 'a list -> 'a list       list append (infix)
rev     :  'a list -> 'a list
hd      :  'a list -> 'a                      head of a non-empty list
tl      :  'a list -> 'a list                 tail of a non-empty list
print   :  string -> unit
```