Using Standard ML of New Jersey—The Basics
Notes by James W. Gray, III
(Last modified by S.C. Cheung: March 26, 1998)

This document gives some very basic information on using Standard ML of New Jersey (sml) release 110.0.3, at HKUST.

1. **Getting Access:** The pathname of the sml compiler/interpreter is:

   ```
   /usr/local/sml/bin/sml
   ```

   You can update your path by putting the following line in your `.cshrc_user` file (located in your home directory).

   ```
   setenv PATH "${PATH}:/usr/local/sml/bin"
   ```

2. **Interactive Mode:** Beginning with the next time you log in, you will be able to start sml in interactive mode by typing

   ```
   sml
   ```

   at the UNIX prompt.

   When sml starts up in interactive mode, it responds with the following.

   ```
   Standard ML of New Jersey, Version 110.0.3, January 30, 1998 [CM; autoload enabled]
   -
   ```

   At this point, you can type expressions such as the following.

   ```
   - (4*7)+3 - 4;
   val it = 27 : int
   - ~3 * 4;
   val it = ~12 : int
   ```

   Note the following.

   (a) Statements are terminated in ML with a semicolon ("`;`"). If you forget to type (or intentionally leave out) the semicolon, sml will respond with an equals sign, "`=`", indicating that the current statement is being continued. At this point, you can simply type the semicolon and hit return.

   (b) "`val it = 27 : int`" is sml's response to the first input expression above. This response means that the value of the last expression you typed is 27, which is of type `int`. sml refers to the last expression that you typed as "`it`".

   (c) Unary minus is indicated by a tilde (`~`), not a dash (`-`). It may take some time to get used to this, so you may want to try out a few expressions using unary minus.

3. **Exiting:** To exit from an interactive session, type "`<CTRL>-d`", i.e., press and hold down the control key and then press d.

4. **Loading Files:** When you are in interactive mode, you can load ML statements from a file using the function called `use`. For example, if factorial.ml is a file containing the definition of a function called `fact`, you can do the following.

```
use("factorial.ml");
[opening factorial.ml]
val fact = fn : int -> int
val it = () : unit
```

5. **val-declarations:** You can bind a value to an identifier using a *val-declaration*. This is a little like a constant definition (e.g., a "`#define`" in C) and a little like an assignment statement. For good functional programming style, it's best if you think of them as constant definitions.

   The following is an example of a val-declaration and sml's response.

   ```
   - val a = 7;
   val a = 7 : int
   ```

   Note that sml's response is almost identical to what you've typed. The main difference is that sml has included the type information (i.e., `a` is of type `int`). You can include the type information too, but in most cases, you can simply let sml *infer* the type.

   Now, you can use the identifier, `a`, in your expressions. For example, you can do the following.

   ```
   - a+5;
   val it = 12 : int
   ```

6. **ML is Case Sensitive:** In ML, "`a`" is not the same as "`A`". For example,

   ```
   - val a = 7;
   val a = 7 : int
   - A+5;
   std_in:15.1 Error: unbound variable or constructor: A
   ```

   ML is case sensitive for variable names as well as built-in function names, type names, keywords, etc.

7. **Types:** Besides integers, ML provides reals, booleans, strings, tuples, and lists. Here are some examples.

   ```
   - true;
   val it = true : bool
   - val r = 4.8 + 2.1;
   val r = 6.9 : real
   - "strings are easy";
   val it = "strings are easy" : string
   - val L = ["this", "is", "a", "list"];
   val L = ["this","is","a","list"] : string list
   - val t = ("this", "tuple", "has", 5, "elements");
   val t = ("this","tuple","has",5,"elements")
     : string * string * string * int * string
   ```

   Note the following.

   (a) Since ML is case sensitive, `true` and `false` are boolean constants, but `True`, `TRUE`, `FALSE`, etc. are not.

(b) Strings are sequences of characters.

(c) Lists are sequences of any given type—as long as the elements are all of the same type.

(d) Tuples can have elements of various types.

You can get the "head" and the "tail" of a list using the functions `hd` and `tl`, respectively. For example, given the above val-declaration for `L`, we can do the following.

```
- hd(L);
val it = "this" : string
- tl(L);
val it = ["is","a","list"] : string list
- hd(tl(L));
val it = "is" : string
```

You can access the elements of a tuple by using the `#n` function (where `n` is an integer). For example, given the above val-declaration for `t`, we can do the following.

```
- #3(t);
val it = "has" : string
- #4(t);
val it = 5 : int
- #5(t);
val it = "elements" : string
```

8. **Operators:** The operators for type boolean, from highest precedence to lowest, are the following.

   - `not` (logical negation—the same as `!` in C),
   - `andalso` (logical and—the same as `&&` in C),
   - `orelse` (logical or—the same as `||` in C).

   When ML evaluates `A andalso B`, `B` will be evaluated only if `A` evaluates to `true`. The reason is the following. If `A` evaluates to `false`, then `A andalso B` will definitely evaluate to `false`; there is no need to evaluate `B`. Similarly, when ML evaluates `A orelse B`, `B` will be evaluated only if `A` evaluates to `false`. This style of evaluation is often called "short circuit evaluation".

   The operators for type integer, from highest precedence to lowest, are the following.

   - `~` (unary minus),
   - `*`, `div` (integer division, rounding down), `mod` (the remainder after integer division),
   - `+`, and `-`.

   The operators for type real are the following.

   - `~`
   - `*`, `/`,
   - `+`, and `-`.

   The empty list is denoted "`[]`" or "`nil`". The operators for type list are the following.

   - `hd(L)` (returns the first element of `L`)

- `tl(L)` (returns the list formed by removing the head of `L`)

Note that the above two operators result in an error if `L` is empty.

We can perform the reverse of `hd` and `tl` using the "`::`" (pronounced "cons", which stands for constructor) operator.

- `x::xs` (returns the list formed by making `x` the head and `xs` the tail; note that if `x` is of type T, then `xs` should be of type `list of` T.)
- `a@b` (concatenate `a` and `b`; here, both `a` and `b` are of type `list of` T for some T.)

The empty string is denoted "`""`". There is only one string operator.

- `^` (string concatenation).

There is only one operator on tuples.

- `#i` (select the i^{th} component).

9. **Conversion between types:** You can convert an integer to a real number using the function `real`. For example,

```
- real(17);
val it = 17.0 : real
```

You can convert a real number to an integer using any of three functions. `floor` always rounds down, `ceil` always rounds up, and `trunc` rounds toward zero. For example,

```
- floor(3.7);
val it = 3 : int
- floor(~3.7);
val it = ~4 : int
- ceil(3.7);
val it = 4 : int
- ceil(~3.7);
val it = ~3 : int
- trunc(3.7);
val it = 3 : int
- trunc(~3.7);
val it = ~3 : int
```

You can convert characters to their integer ASCII code using the "`ord`" function. For example,

```
- ord(#"q");
val it = 113 : int
- ord(#"Q");
val it = 81 : int
- ord(#"1");
val it = 49 : int
```

You can perform the inverse operation, converting an integer ASCII code to the corresponding character, using the `chr` function. For example,

```
- chr(113);
val it = #"q" : char
```

You can convert a string to a list using `explode`, which takes a string and returns a list of one-character strings formed by taking apart the string, one character at a time. For example,

```
- explode("this");
val it = [#"t",#"h",#"i",#"s"] : char list
- explode("");
val it = [] : char list
```

You can perform the reverse operation using `implode`, which takes a list of strings and returns a single string formed by concatenating all of the strings in the list. The strings in the list can be of any length. For example,

```
- implode([#"t",#"h",#"i",#"s"]);
val it = "this" : string
- implode([]);
val it = "" : string
- concat(["this"," ","is"]);
val it = "this is" : string;
```

10. **Function Definitions:** Suppose we want to define a function that squares a real number. We can define it as follows.

```
- fun square(x:real) = x*x;
val square = fn : real -> real
- square(3.0 + 1.5);
val it = 20.25 : real
```

Note the following.

(a) The response from sml is `val square = fn :  real -> real`. This means "`square`" is a function (`fn`) whose type is `real -> real`. That is, `square` maps real numbers to real numbers. Or in other words, `square` takes a single parameter of type `real` and returns a single value of type `real`.

(b) As in C, ML parameters are passed "by value". For example, when we call "`square(3.0 + 1.5);`" the parameter "$3.0 + 1.5$" is evaluated one time, prior to entering the function to obtain its value (viz, 4.5). This value is used in place of "`x`" in evaluating the value of the function.

You can define functions with more than one parameter. Consider the following.

```
- fun max(x:real, y) = if x > y then x else y;
val max = fn : real * real -> real
```

Note the following.

(a) There is no need to declare the type of `y`. This is because ML can *infer* the type. This is done as follows. First, `x` is declared to be of type `real` so ML knows that. Next, `y` is compared to `x` (using the test "`x > y`"), so ML knows that `y` must have the same type as `x`. Thus, ML knows that `y` must be of type `real`.

(b) ML denotes max's type as `real * real -> real`. Here, `*` denotes the cartesian product. Thus, max is a mapping from pairs (or binary tuples) of reals to reals. In other words, max takes two parameters of type `real` and returns a single value of type `real`.

In ML, a function can return a value of any type. For example, you can define a function to return a list. Consider the following function `compound(x,n)` that returns a list of `n` `x`'s.

```
- fun compound(x,n) = if n = 0 then [] else x::compound(x,n-1);
val compound = fn : 'a * int -> 'a list
- compound(0,3);
val it = [0,0,0] : int list
- compound("hello",4);
val it = ["hello","hello","hello","hello"] : string list
```

Note the following.

(a) The definition of `compound` is recursive.

(b) "`::`" is an operator for constructing lists. It is a *binary infix* operator, meaning that it takes two operands (binary) and that the operator is written in between the two operands (infix). The expression "`e::L`" means put the element, `e`, on the front of the list, `L`.

(c) The first argument to `compound` can be *any* type. This is indicated by the sml's response:

```
val compound = fn :  'a * int -> 'a list
```

Here, `'a` is a type variable. This means for any type, `'a`, `compound` takes two parameters, the first of type `'a` and the second of type `int`, and returns an `'a list`.

11. **Local Variables:** You can introduce a temporary variable within a function definition. This is often useful to avoid evaluating the same expression twice. For example, suppose we want to compute the variance of a list of real numbers, $L = [L_1, L_2, \ldots, L_n]$. The variance can be defined as follows.

$$variance(L) = \left(\frac{\sum_{i=1}^{n}(L_i)^2}{n}\right) - \left(\frac{\sum_{i=1}^{n} L_i}{n}\right)^2$$

Therefore, we might implement this in ML as follows.

```
fun sum_squares(L) =
    if L = nil then 0
    else hd(L)*hd(L) + sum_squares(tl(L));

fun sum_list(L) =
     if L = nil then 0
     else hd(L) + sum_list(tl(L));

fun variance(L) =
     sum_squares(L)/real(length(L)) -
     (sum_list(L)/real(length(L))) * (sum_list(L)/real(length(L)));
```

This will work correctly.[1] However, note that `sum_list(L)` is evaluated twice and `real(length(L))` is evaluated three times. We can avoid this by introducing two local variables into the definition of `variance` as follows.

```
fun variance(L) =
    let
        val n = real(length(L));
        val s = sum_list(L);
    in
        sum_squares(L)/n - (s/n) * (s/n)
    end;
```

Note that you can introduce as many `val` declarations as you want between the `let` and the `in`. Further, a `val` declarations can be defined in terms another `val` declaration appearing earlier in the list. For example, we can further improve the definition of `variance` by defining a local variable for `s/n` as follows.

```
fun variance(L) =
    let
        val n = real(length(L));
        val mean = sum_list(L)/n;
    in
        sum_squares(L)/n - mean * mean
    end;
```

12. **Input and Output:** Concepts such as "read from standard input" and "output to a file" do not exist in pure functional programming. However, they make life a lot easier and so, they are provided in Standard ML of New Jersey.

    First of all, you can print to standard output (e.g., the display) using the function "`print`". This function can take a single argument of type `string`. For example,

    ```
    - print("The Right of Spring\n");
    The Right of Spring
    val it = () : unit
    - print("3");
    3val it = () : unit
    ```

    Note the following.

    (a) The type of the expression `print("The Right of Spring\n");` is `unit`, not `string`. This is because sml treats the output of `print` as a "side effect"; it is not the value returned by `print`. The value is a tuple with zero elements (viz, "`()`"), which is of type "`unit`".

    (b) The output of "`print(3)`" is printed without a newline character. Thus, "`3val it = () : unit`" is the output of `print`, immediately followed by sml's normal response.

    sml does not provide a single function for printing the number "3" followed by a newline. However, we can use a statement list; that is, a list of statements, separated by semicolons (";") and surrounded by parentheses. For example,

[1] Of course, `variance` will fail—due to a division by zero—in the case where the length of the list is zero, but this case does not concern us here.

```
- (print("3"); print("\n"));
3
val it = () : unit
```

Note that the last statement in a statement list is not terminated by a semicolon. Doing so produces an error:

```
- (print("3");print("/n"););
std_in:8.23 Error: syntax error found at RPAREN
```

Semicolons in ML are very much like semicolons in Pascal; they are used as statement separators.

The value returned by a statement list is the value returned by the last statement in it. Thus, the statement list construct is useful only when the previous statements have some side effects (such as printing values).

Suppose there is a file called “fileio.test” in the directory where you are running sml. Also suppose that the contents of “fileio.test” are the following.

```
Knight on Bald Mountain
```

You can read this file as follows.

```
- val infile = open_in("fileio.test");
val infile = - : instream
input(infile,1);
val it = "K" : string
- input(infile,10);
val it = "night on B" : string
- end_of_stream(infile);
val it = false : bool
- input(infile,13);
val it = "ald Mountain\n" : string
- end_of_stream(infile);
val it = true : bool
```

In the above, the function `open_in` opens a file for reading. The value returned by this function is not accessible to the programmer—sml represents the value by a dash (“`-`”)—but you can think of it as the file identifier returned by the UNIX call to open a file. In the above example, `infile` is simply a variable that gets this value in the normal way for `val` declarations.

A call to `input(infile,n)` returns the next n characters (if there are that many) in the file.

A call to `end_of_stream(infile)` tests whether there are any more characters to be read.