Tutorial 6

Week of February 19, 2007

1 SML Environment

1. How do I start SML?

   On CDF just type `sml` and you will enter the SML environment:

   ```
   werewolf:\$ sml
   Standard ML of New Jersey, Version 110.0.7,
       September 28, 2000 [CM; autoload enabled]
   -
   ```

   The '-' is the SML prompt. Now you can type in ML code and the interpreter immediately evaluates it and produces an output.

2. How do I quit SML?

   simply press `Ctrl-D` (under Windows press `Ctrl-Z`)

3. How do I load my code from a file?

   The function `use` is defined at top level and will load a file containing SML source code.

   For example, loading your file `myfile.sml`:

   ```
   - use "myfile.sml";
   [opening myfile.sml]
   val it = () : unit
   -
   ```

   What is `val it = () : unit` ? The function `use` returns `()` of type `unit`. We will see what this means later.

   If you are are loading a file that is in a directory different from the one you started sml in, then you need to provide a full path to the file.

   Note the semicolon at the end: to ML, it means "the user is done tying, now I should start working".

4. How do I determine the current directory or change it?

   The function `OS.FileSys.getDir()` returns the current directory.
   The function `OS.FileSys.chDir("path")` sets the current directory to `path`.

5. What do I use for comments?

   Use (* comments are here *). The comment can be multi-line.

6. SML is *case sensitive*:

   So you must match the case exactly (e.g. `getdir` would be an error!)

7. SML Interactive input:

   Input to the top level interpreter (i.e., declarations and expressions) must be terminated by a semicolon (and carriage return) before the system will evaluate it. The system then prints out a response indicating the effect of the evaluation. *Expressions are treated as implicit declarations of a standard variable `it`.* For example,

```
- 3;                        (* user input after prompt *)
val it = 3 : int            (* system response *)
```

   This means that the value of the last top level expression evaluated can be referred to using the variable `it`. For example,

```
- 2.5;
val it = 2.5 : real
- it;
val it = 2.5 : real
-
```

8. Multi-line expressions:

   If you type a long expression, you can enter it in several lines. In this case the prompt for subsequent lines changes to =. Note that you still need to end the expression with a semicolon.

```
  - if 10 > 5
  = then 1
  = else 2;
  val it = 1 : int
```

9. Interrupting compilation or execution:

   Typing `Ctrl-C` should interrupt the interpreter and return you to top level (useful to interrupt infinite recursion).

10. Error messages:

   When compiling files, the error messages include line numbers and character positions within the line. For example:

```
 - if true
 = then 5 true
 = else 6;
 std_in:7.6-7.11 Error: operator is not a function [literal]
     operator: int
     in expression:
          5 true
```

This means: there is an error that starts at line 7, character 6 and ends at line 7 character 11. And the error is: I am trying to apply 5 to true, and I can't.

11. More Error Messages:

    There are a number of different forms of type error message, and it may require some practice before you become adept at interpreting them. The most common form indicates a mismatch between the type of a function (or operator) and its argument (or operand). A representation of the offending expression is usually included, but this is an image of the internal abstract syntax for the expression and may differ significantly from the original source code.

12. A very useful reference for a list of all error messages.

    All the error messages produced by SML/NJ are documented in the SML/NJ Error and Warning Messages page:

    `http://www.smlnj.org/doc/errors.html`

13. Debugging in SML:

    There is no debugger installed with SML/NJ. SML is a strongly typed language and its type inference system is powerful enough to capture a wide variety of errors and ambiguities. Furthermore, there is run-time checking so there's no way you can crash the system (except for an infinite loop).

2 ML basics data types

1. **unit**: this type has only one element ()

   ```
   - ();
   val it = () : unit
   ```

2. **bool**: this type has two elements: `true` and `false`

   Operations on bools: not, andalso, orelse.

   For example:

   ```
   - if (not true andalso false) orelse true
   = then "foo"
   = else "bar";
   val it = "foo" : string
   ```

3. **integer**: {... ˜ 2, ˜ 1, 0 , 1 , 2, ... }

   Operations: $+$, $-$, $*$, `div`, `mod`, $<$, $>$, $=$, $<=$, $>=$, $<>$

   For example:

   ```
    - 5 + 6 * 2 - 3 div 2;
    val it = 16 : int

    - 5 mod 2 >= 6 mod 2;
    val it = true : bool
   ```

   Note the use of ˜ for negation. Here's why: ”-” is a binary operator while ”˜ ” is a unary operator. For example:

   ```
   - -5;
   stdIn:27.1 Error: expression or pattern begins with infix identifier "-"
   stdIn:27.1-27.3 Error: operator and operand don't agree [literal]
     operator domain: 'Z * 'Z
     operand:         int
     in expression:
       - 5
   - ~5;
   val it = ~5 : int
   ```

   In the first case, SML complains that “-” is a binary operator and we only provided one argument.

4. **real**: 1.0, 3.14159, 11.7, ...

   Operations: +, −, ∗, <, >, =, <=, >=

   Note that = and <> are not defined for reals.
   Instead, we can use `a <= b andalso b >= a`.

   Note: You **cannot** mix reals and integers in one expression. For example:

```
- 2 - 1.5;
stdIn:43.1-43.8 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    2 - 1.5
```

   This means: there is an error between line 43, character 1 and line 43, character 8. The error is: operator and operand don't agree, meaning the type of the operand provided to an operator is not what the operator expects. In this case: operator domain is two integers, but we gave it an integer and a real.

   Now, but wasn't there an operator "-" defined for reals as well? Yes, that's right. But for mathematical operators, if the type is not specified (or inferred), then type integer is assumed. For example:

```
- 2;
val it = 2 : int

- 2.0;
val it = 2.0 : real

- real(2);
val it = 2.0 : real

- 2.0 - 1.5;
val it = 0.5 : real

- real(2) - 1.5;
val it = 0.5 : real
```

Note that it inspects the leftmost argument first. For example:

```
- 1.5 + 2;
stdIn:1.1-2.3 Error: operator and operand don't agree [literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    1.5 + 2
```

Now it expects two reals!

5. **string**: "Hello, world", "Wow!", "CSC324 is fun!"

   Operations: ^ for concatenation

   For example:

```
- "Hello";
val it = "Hello" : string

- "Hello" ^ " " ^ "Jim";
val it = "Hello Jim" : string
```

6. **list**: [1, 5, 2, 3]   [] is called **nil**, it is the empty list

   Operations: ::, @, tl , hd

   For example:

```
- [];
val it = [] : 'a list

- [1, 2, ~3];
val it = [1,2,~3] : int list

- "foo"::["bar"];
val it = ["foo","bar"] : string list

- [1,2]@[3,4];
val it = [1,2,3,4] : int list

- hd [1,2];
val it = 1 : int

- tl [1,2];
val it = [2] : int list
```

Very important: all elements of the list **must** be of the same type. Consider the following:

```
- [1, 2, "Hello"];
stdIn:17.1-17.16 Error: operator and operand don't agree [literal]
  operator domain: int * int list
  operand:         int * string list
  in expression:
    2 :: "Hello" :: nil
```

The error message means that there is an error between line 17, character 1 and line 17, character 16. Then it says that operator and operand don't agree. But did we have any operators??? Yes!!!!

When we typed `[1,2,"Hello"]`, we actually did `1::2::"Hello"::nil`. And the error is: cannot do `2::"Hello"::nil`, because 2 is an integer, and `["Hello"]` is a list of strings!

7. **tuple**: (1, "Test", 2.5)   note that `unit` is an empty tuple

   Operations: accessing component N of a tuple Tuple: `#N Tuple`

```
- (1, "Test", 2.5);
val it = (1,"Test",2.5) : int * string * real

- #2(1, "Test", 2.5);
val it = "Test" : string

- ("a", 1, ~2) = ("a", 2-1, 1-3);
val it = true : bool
```

8. **record**: {lastname="Smith", ID=200, Age=59}

   Note: records are similar to tuple but their components have names, and the order does not matter! In fact, a tuple is a special case of a record, where the names of the fields are 1, 2, 3, ...

   Operations: accessing component Name of record Record: `#Name Record`

```
- {lastname="Smith", ID=200, Age=59};
val it = {Age=59,ID=200,lastname="Smith"} : {Age:int, ID:int, lastname:string}

- #lastname {lastname="Smith", ID=200, Age=59};
val it = "Smith" : string
```

But what if I want my own variable? Not just this `it` that keeps changing? You can declare your own:

```
 - val aNumber = 4*5;
val aNumber = 20 : int

- val aBiggerNumber = aNumber + 10;
val aBiggerNumber = 30 : int

- val Sheila = {job="Prof", course=324, location="StG"};
val Sheila = {course=324,job="Prof",location="StG"}
  : {course:int, job:string, location:string}

- #job Sheila;
val it = "Prof" : string
```

3 Functions

Very important: any ML functions take **exactly 1 argument**. The argument may be a list, a tuple, a record, etc., but it is 1 singe argument.

The syntax for function declaration:

```
 fun <name> <argument> = <body>;
```

Examples:

```
- fun thisCourse () = "csc324";
val thisCourse = fn : unit -> string

- thisCourse ();
val it = "csc324" : string

- fun thisCourse x = 324;
val thisCourse = fn : 'a -> int

- thisCourse "foo";
val it = 324 : int

- thisCourse 333;
val it = 324 : int
```

```
- fun sum(x,y)=x+y;
val sum = fn : int * int -> int

- sum(1,1);
val it = 2 : int

- sum(1, 1.1);
stdIn:25.1-25.12 Error: operator and operand don’t agree [tycon mismatch]
  operator domain: int * int
  operand:         int * real
  in expression:
    sum (1,1.1)

- fun sum(x:real, y:real) = x+y;
val sum = fn : real * real -> real

- sum(1.0, 1.1);
val it = 2.1 : real

- fun sum(x, y:real)=x+y;
val sum = fn : real * real -> real

- fun sum(x,y)=(x+y):real;
val sum = fn : real * real -> real

- fun len L = if null L then 0 else 1 + len (tl L);
val len = fn : ’a list -> int

- len [1,2,3];
val it = 3 : int

- len["csc324","is","fun","!"];
val it = 4 : int

- fun sumlist L = if null L then 0 else hd L + sumlist(tl L);
val sumlist = fn : int list -> int

- sumlist[1,2,3];
val it = 6 : int
```

```
- sumlist[2.3, 4.5, 6.7];
stdIn:36.1-36.23 Error: operator and operand don’t agree [tycon mismatch]
  operator domain: int list
  operand:         real list
  in expression:
    sumlist (2.3 :: 4.5 :: 6.7 :: nil)

- fun switch(x,y) = (y,x);
val switch = fn : ’a * ’b -> ’b * ’a

- fun add_dummy (x, y, z) = (x, y, z, "dummy");
val add_dummy = fn : ’a * ’b * ’c -> ’a * ’b * ’c * string

- fun add_dummy x = "dummy"::x;
val add_dummy = fn : string list -> string list
```

4 Pattern matching

The syntax for function declaration with pattern matching:

```
fun <name> <pattern1> = <exp1>
  | <name> <pattern2> = <exp2>
  .
  .
  | <name> <patternN> = <expN>;
```

This means: the function name is **name**. It takes one argument (as any other ML function). It tries to match the argument to pattern1. If it succeeds, it returns exp1. Otherwise, tries to match the argument to pattern 2. Etc, etc.

For example we can rewrite our `len` to use pattern matching:

```
- fun len [] = 0
  | len (head::tail) = 1 + len tail;
val len = fn : 'a list -> int

- len [1,2,3];
val it = 3 : int

- len["csc324","is","fun","!"];
val it = 4 : int
```

But beware non non-exhaustive patterns:

```
- fun len (head::tail) = 1 + len tail;
stdIn:44.1-44.36 Warning: match nonexhaustive
          head :: tail => ...

val len = fn : 'a list -> int

- len [1,2,3];

uncaught exception nonexhaustive match failure
  raised at: stdIn:44.24
```

More examples:

Define a function `firstlist` which takes as input a list of pairs and gives back the list consisting of the first elements only. For example:

```
   firstlist [] = []
   firstlist [(1,2),(1,3)] = [1,1]
   firstlist [(1,"a"),(2,"b"),(3,"c")] = [1,2,3]
   firstlist [([],"a"),([1],"b"),([1,2],"c")] = [[],[1],[1,2]]
```

```
Solution:

- fun firstlist [] = []
  |   firstlist ((e1, e2)::tail) = e1::(firstlist tail);

val firstlist = fn : ('a * 'b) list -> 'a list
```

Note: no variable can occur twice in each pattern!
For example,

```
- fun eq(x,x)=true
    | eq(x,y)=false;
Error: duplicate variable in pattern(s)

- fun eq(x,y) = (x=y);
val eq = fn : ''a * ''a -> bool

- fun eq(x,y) = x<=y andalso y<=x;
val eq = fn : int * int -> bool

- fun eq(x:real, y) = x<=y andalso y<=x;
val eq = fn : real * real -> bool
```

5 Anonymous functions

The syntax for anonymous functions is:

```
  fn <argument> => <body>;
```

Examples:

```
- fn x => x;
val it = fn : 'a -> 'a

- (fn x => x) 5;
val it = 5 : int

- (fn x => x) "foo";
val it = "foo" : string

- fn (x, y) => [2*y, 2*x];
val it = fn : int * int -> int list

- fn (x,y) => x^y^"!";
val it = fn : string * string -> string
```

More complicated example:

Write a function `double` that takes a list of tuples of integers and strings and doubles the integer in every tuple.

```
fun double [] = []
|   double ((first:int*string)::rest) = (2 * #1first, #2first)::double(rest);

val double = fn : (int * string) list -> (int * string) list

- double [(1, "abc"), (2, "def")];
val it = [(2,"abc"),(4,"def")] : (int * string) list

BUT:

fun double [] = []
|   double (first::rest) = (2 * #1first, #2first)::double(rest);

=> Error: unresolved flex record (need to know the names of ALL the fields
   in this context)
```

6 Functions as parameters

Write a function `applyall` that takes a list of functions and a value and returns the list, the elements of which are the results of applying every function in the input list to the input value, in order.

```
fun applyall ([], _) = []
|   applyall (first::rest, x) = (first x)::applyall(rest,x);

val applyall = fn : ('a -> 'b) list * 'a -> 'b list

fun positive n = n>0;
fun nonnegative n = n>=0;
fun id n = n;
fun double n = 2*n;
fun listoftwo L = length(L)=2;

applyall([positive, nonnegative], 0);
val it = [false,true] : bool list

applyall([id, double], 1);
val it = [1,2] : int list

applyall([null, listoftwo], [1,2]);
val it = [false,true] : bool list
```