Main topics of the week:
- Introduction to ML – syntax, types
- Values and functions

Scheme is an untyped functional language. ML is a typed functional language.
Again, everything is a function. In pure ML, no side effects.
No assignment.
No variables in the sense of mutable values – “variables” are bound at point of declaration.
ML stands for Meta Language – originally developed for theorem proving.
Statically typed – very strong typing (as we’ll see), but also has type inference (as we’ll see)
ML is case sensitive unlike the case insensitivity of Scheme.
ML has several variants; one is sml-nj from Bell Labs.
ML is an interpreted language, so we just invoke sml.

Invoking ML
The system binds an expression to ‘it’ and may infer the type if necessary. Type is everything in
ML. There are some builtin types of int and real (and later we’ll see bool and others). A dialog in
ML goes like the following, where the prompt at which we type is ‘-’. The interpreter evaluates
the statement when it sees the terminating semicolon – if the interpreter expects more input, it
will prompt with ‘=’. We terminate the interpreter session with end-of-file (CTRL-D). The input
we key in is shown in boldface, and the interpreter’s response in italics.

- 1 + 2;
  val it = 3 : int
- val z = it + 1;
  val z = 4 : int

In the response from the interpreter, ‘val’ stands for value, ‘it’ stands for the previous expression.
A colon separates the value of the expression from the type of the expression.

Constants in ML can be int with the usual decimal or hexadecimal format. One important
difference from Java/C++ is that the minus sign is not used in integers or floating point numbers
for negation; rather the minus sign always means subtraction (a binary operator). For negation
(the unary operator), the tilde is used. The floating point numerical type is called real with the
usual decimal point notation, or exponential notation. ML also has a Boolean type bool which
can have the constant values true and false (explicitly lower case). For the string type, ML uses
the usual double quoted strings, with the usual escapes. For individual characters, it uses the
somewhat verbose and clumsy notation of preceding a single character string by a #.

- 123;
  val it = 123 : int
- ~123;
  val it = ~123 : int
- 123.456;
  val it = 123.456 : real
- ~123.456;
  val it = ~123.456 : real
- true;
  val it = true : bool
- false;
  val it = false : bool
- "foobar";
  val it = "foobar" : string
Operators in ML are the usual arithmetic operators (+ - * / div mod ~). “mod” is the modulus operator, and div is integer division with truncation of fractional part. The standard division operator / can only be used for reals. In addition to arithmetic operators, there is also a string operator “^” for string concatenation. Finally, there are comparison operators: the usual < > <= >=. Equality is the single “=”, and inequality uses the symbol “<>”.

```
- "foobar\n";
  val it = "foobar\n" : string
- "foo\n  = \bar";
  val it = "foobar" : string
- #"f";
  val it = #"f" : char
```

```
- 3 + 5;
  val it = 8 : int
- 3 - 5;
  val it = -2 : int
- 3 div 5;
  val it = 0 : int
- 5 div 3;
  val it = 1 : int
- 5.0 / 3.0;
  val it = 1.66666666667 : real
- 5 mod 3;
  val it = 2 : int
- "hello" ^ " " ^ "world";
  val it = "hello world" : string
- "hello" < "world";
  val it = true : bool
- "hello" > "world";
  val it = false : bool
- 5 <> 3;
  val it = true : bool
- 5 = 3;
  val it = false : bool
```

Logical operators to combine expressions are andalso and orelse, and the evaluation of these behaves as in C, with short circuited evaluation. Logical negation is accomplished with the not operator. As another example of user hostile language design, “and” is a keyword with a completely different meaning – it’s used in mutual recursion definitions.

```
- 2 > 3 or else true;
  val it = true : bool
- 2 > 3 andalso true;
  val it = false : bool
```

```
- 2 > 3 and true;
  val it = false : bool
```

```
= ;
Error: syntax error found at AND
- not 2 > 3;
Error: operator and operand don't agree
- not (2>3);
  val it = true : bool
```

ML expressions can be conditional, e.g.,

```
- if true then 1 else 2;
  val it = 1 : int
```

All three keywords are required, so this is not a control flow statement, but rather equivalent to the ternary conditional operator in C or the if function in scheme.
Type Consistency

Types in ML must be consistent.

```ml
- if true then 1 else true;
  Error: types of rules don't agree
```

The type consistency for if-then-else expressions is that the “then” and “else” may be any type, but must be the same type. The “if” expression must be type bool. Likewise, there are other type requirements for expressions: the operands of andalso and orelse must be bool, operands of ^ must be strings, etc.

This consistency requirement is extremely strict. Some arithmetic operators in ML are overloaded, that is, they have different semantics for different types (much as in Java/C++). Thus, ‘+’ between integers is different from ‘+’ between reals. But ML is very strict and there is no coercion between integers and reals:

```ml
- 1 + 2;
  val it = 3 : int
- 1.0 + 2.0;
  val it = 3.0 : real
- 1 + 2.0;
  Error: operator and operand don't agree
```

Similarly, the comparison operators <, >, etc. require both operands to be identical types. ML does provide conversion operators to convert real’s to int’s: floor, ceil, round, trunc. The conversion from int to real is the “constructor” real itself:

```ml
- floor(3.5);
  val it = 3 : int
- ceil(3.5);
  val it = 4 : int
- round(3.5);
  val it = 4 : int
- trunc(3.5);
  val it = 3 : int
- 1 + round(3.5);
  val it = 5 : int
- 3.5 + real(1);
  val it = 4.5 : real
- int(3.5);
  Error: unbound variable or constructor: int
```

We will see more about type consistency when we start defining functions.

Variables and Environments

ML has variables like other languages, i.e., identifiers that are associated with storage (memory). However, the variables are not mutable, that is, they are constant with the stored value being set at the point of declaration and never changed after that. Now this may seem like it can’t be that useful – on the surface such a scheme appears to be equivalent to preprocessor defines in C, or having all variables const in C++, or having all variables final in Java. However, in the functional paradigm, we can think of storage as growing and shrinking (like the stack in a
procedural language). Even though values cannot change, we simply add new copies of variables for a new context (i.e., a function). Think of it as not being able to erase our calculated values during a computation, but rather just going to a new blank portion of the paper and making that our “current” value.

**Identifier Rules**

Alphanumeric – follow the usual C rules of letters, digits, underscore (and also single quote), can’t begin with digit. Variables that begin with single quote (also called tick or prime) are type variables and cannot be bound to the usual numeric, string, bool, or character values we have seen, but could be bound to those types.

Symbolic – Symbolic Identifiers can use symbols like +, -, *, etc – i.e., all the characters except those used in alphanumeric identifiers with the exception of the characters needed for ML’s syntax. The only characters that cannot be used in any identifier are: () {} [] “ . , ; (and white space). Some of these identifiers are builtin, e.g., +, -, *. The point here is that the syntax is not built around these symbols – rather they are like keywords in that the lexical analysis delineates the symbolic identifiers, and some of them happen to be pre-defined. This means that symbolic identifiers can be used as variables – this is not very common and only makes sense if we want to define them as functions so as to have additional symbolic operators that we define.

**Environment**

The ML interpreter begins execution with a top-level environment. Think of this as a table that lists identifiers and their associated value. Various things will be in the top-level environment to start with, e.g., so called builtin functions. When the interpreter encounters a variable declaration of the form:

```
- val foo = 13;
val foo = 13 : int
- val bar = 13.0;
val bar = 13.0 : real
```

The identifier and value is added to the environment. This looks a lot like like an assignment, but it is more precise to think of it as a constructor of the value. In fact, it is very much like the constructor for a constant in C++ or a final in Java- the initial value is the permanent value. However, we can “re-declare” the same identifier. And it can even be of a different type, which tells us that we are not assigning

```
- foo + 2;
val it = 15 : int
- bar + 2.0;
val it = 15.0 : real
- foo + bar;
Error: operator and operand don't agree
- val bar = 2;
val bar = 2 : int
- foo + bar;
val it = 15 : int
```

Of course, this won’t be really interesting until we see how to create blocks or contexts so that we can get back to the old values.

**Type Constructors: Tuples and Lists**

Types in ML can also be tuples and we can access fields of tuple values. Note that parentheses are used to denote tuples:

```
- (2, 3);
```
val it = (2,3) : int * int
- ("if", true);
val it = ("if",true) : string * bool
- val city = ("Eugene", "OR", 97402);
val city = ("Eugene","OR",97402) : string * string * int
- val addr = ("123 1st", city);
val addr = ("123 1st",("Eugene","OR",97402)) : string * (string * string * int)
- val state = #2(city);
val state = "OR" : string

Note that tuple components can be of differing types. However, it is useful to have lists of identically typed values (like arrays in Java/C++). The notation for lists uses square brackets. There are also operators on lists (hd returns the first element, tl returns the list comprising the tail), and a predefined symbol nil for the empty list. Moreover, we can append one list to another (of the same type) with the @ operator, and prepend an element to the beginning with ::

- val L = ["first", "second", "third"]; val L = ["first","second","third"] : string list
- hd(L);
val it = "first" : string
- tl(L);
val it = ["second","third"] : string list
- tl(tl(L));
val it = ["third"] : string list
- hd(tl(tl(L)));
val it = "third" : string
- L@nil;
val it = ["first","second","third"] : string list
- L@L;
val it = ["first","second","third","first","second","third"]
  : string list
- val L2 = L::"fourth";
Error: operator and operand don't agree
- val L2 = "zero"::L;
val L2 = ["zero","first","second","third"] : string list

Function Definition

We have seen a lot about how ML deals with variables, but haven’t yet seen how to program in ML. Because ML is a functional language, all “programming” is done by way of function calls, so we need to see how to define a function in ML.

Functions can be defined in ML with the keyword fun, e.g.,
- fun f x = x + 1;
val f = fn : int -> int
where the function has the name f, a formal parameter x, and a body x+1. This is an example of a single expression definition of a function, where the body is just a single expression. Alternatively, we can bind a variable to an anonymous function:
- val f = fn x => x + 5;
val f = fn : int -> int
- f 7;
val it = 12 : int
Note the use of the keyword fn instead of fun. Fn is used to specify a type, and the => is used to specify the definition. Moreover, when the ML interpreter replies with the type, it uses the fn keyword, and expresses the type of the function as the type(s) of the parameters followed by the type of the expression to which the function evaluates. This is like a lambda expression in Scheme or a function prototype in C++. 
We have seen that declarations of variables in ML actually produce constants since there is no assignment, hence a “variable” value never can change. This means that variables and functions are treated consistently, unlike in C, where functions are constant.

ML does resolve some ambiguity by default, but the second expression shows how to explicitly eliminate ambiguity by explicitly declaring the type of the argument:

```ml
- fun g x = x + x;
val g = fn : int -> int
- fun g x: int = x + x;
val g = fn : int -> int
```

In general, ML will try to infer types, always with the goal of ensuring type consistency, and will only complain if no suitable inference can be made.

### Function Application

Of course once we define a function, we would like to be able to use it.

```ml
- fun square x: real = x*x;
val square = fn : real -> real
- val pi = 3.14159;
val pi = 3.14159 : real
- pi * square 2.0;
val it = 12.56636 : real
- pi * square(2.0);
val it = 12.56636 : real
- pi * square(2);
Error: operator and operand don't agree
```

Some things to note: parentheses are not required around the function argument, although they don’t hurt. ML knows that square is a function, so unlike Java/C++ does not need the parentheses in the syntax to distinguish a function call. Note also that the argument type must match exactly. Since we defined square with a parameter type of real, that type is required and it is an error to try to pass any other type. If we had not explicitly defined square to take a real, then it would have defaulted to int, and it would be an error to pass any other type. In particular, there is no coercion. This is like the treatment in C without function prototypes, except that here the interpreter detects the error rather than causing a runtime error as in C, where the argument is passed, but is interpreted incorrectly by the function body code.

### Multiple arguments to functions

Functions may also have multiple parameters. These can be specified in two ways: without parentheses or with. In the former case, the type of the function is that it takes one parameter and returns a function taking one parameter which returns a real. In the latter case, the function takes a single parameter which is a pair of reals and returns a real.

These appear to amount to the same thing, but notice that parentheses in the call are required if we used the tuple notation, because in that case the function is taking a single argument which is a tuple. In particular all ML functions actually take a single argument, although that argument can be a tuple. Note also that this means functions may return a function, which is then applied to what we are thinking of as the second parameter. So when you have a function that takes a number of arguments, not as a tuple, what you really have is a whole bunch of functions.

Applying the function to the first parameter gives you another function, which you could think of
as a partial instantiation of the “whole” function. Decomposing functions one parameter at a time is known as **Currying**:

```ml
- fun area r = pi * square r;
val area = fn : real -> real
- fun volume r h = h * area r;
val volume = fn : real -> real -> real
- volume 3.0 2.0;
val it = 56.54862 : real
- volume(3.0, 2.0);
Error: operator and operand don't agree
- (volume 3.0) 2.0;
val it = 56.54862 : real
- fun volume(r, h) = h * area r;
val volume = fn : real * real -> real
- volume 3.0 2.0;
Error: operator and operand don't agree
- volume (3.0, 2.0);
val it = 56.54862 : real
```

Note also in these examples that ML is able to infer the type of real from the explicit type of real specified in the square function.

**Another example of Currying**

Consider the following two function definitions:

```ml
- fun f x1 x2 x3 = x1 + x2 + x3;
val f = fn : int -> int -> int -> int
- fun g(x1, x2, x3) = x1 + x2 + x3;
val g = fn : int * int * int -> int
```

What is the difference? Before looking at the signatures, let’s reason about them. f appears to have three arguments and so does g, but actually each of them has only one argument. In the case of g, that argument is a 3-tuple of integers. Simple enough, but what about f? It actually has one single integer argument. This brings up the question of return values. For g, it’s pretty obvious the return value is an integer. For f, that may look like the case as well, but if we look closer, we realize that f actually returns a function, and that function in turn is applied to x2, which also returns a function, which is finally applied to x3, which returns an integer. That is, there are three functions here, the last of which takes a single integer and returns an integer. The first two functions each return functions. Although f and g may seem equivalent, they are not exactly alike. When used with three integer values, they behave the same. The implementation of f may be more expensive (involving the creation of more functions) than g, but that is the compiler’s problem, not ours. We call f the **Curried** form of g, and it is actually syntactic sugar for the following:

```ml
- fun F x = (fn y => (fn z => x + y + z));
val F = fn : int -> int -> int -> int
- F 1 2 3;
val it = 6 : int
- f 1 2 3;
val it = 6 : int
```

To see that f and g are really different, we ask if f and g could be used interchangeably in all expressions. That sounds reasonable, but consider:

```ml
- val a = f 1;
val a = fn : int -> int
- val b = a 2;
val b = fn : int -> int
- val c = b 3;
```
val c = 6 : int
- val d = g 1;

stdIn: Error: operator and operand don't agree

What we see here is that the expression of f with one argument makes sense – it is a function. When we try the same thing with g, we get a type error, because g requires 3 arguments. So from a type point of view, they really are different.

Some functions can be defined without specifying types. These are called **polymorphic** functions, or alpha functions (alpha is written ‘a in ML) and are like templates in C++:

- fun f x = x;

  val f = fn : 'a -> 'a

This seems to be a trivial example, and certainly is simple, but it is the identity function and can actually be useful. Polymorphism is not restricted to parameterizing on a single type. As the following example shows, we can form a function that takes two arguments and forms a tuple:

- fun pair x y = (x, y);

  val pair = fn : 'a -> 'b -> 'a * 'b

Thus, the tuple operator is not type specific and thus does not at all limit the types in the function, while some operators are. In particular, the difference with the multiplication ‘*’ operator is that the operator is just overloaded for some specific types, while a polymorphic function works with any type. That is why we have seen the types be inferred (default of int) for the square function.