15-150 Lecture 2: SML Basics

Lecture by Dan Licata

January 19, 2012

I'd like to start off by talking about someone named Alfred North Whitehead. With someone named Bertrand Russell, Whitehead wrote *Principia Mathematica*, the first volume of which was published in 1910. This book was the first major attempt at doing mathematics in a fully formal way, inside of logic. On page 362, they even prove that $1 + 1 = 2$. (I bring this up because in lab someone asked what level of detail of proofs we expect. I'm not serious. Well, maybe a little serious.)

Fast-forwarding a century, this book is important because, if you want to do math on a computer, you need to do it in a formal logic. For example, Tom Hales over at Pitt has a proof of the Kepler conjecture (the way oranges are arranged at the grocery store is the most space-efficient way to arrange them), but the mathematical community doesn't believe the proof, because it involves some computer calculations, and is really long. He is formalizing the proof using a tool called HOL Light, which will allow computers to verify the proof. This is very much in the spirit of Russell and Whitehead's program. And do you know what language you use to interact with HOL Light? Not SML, but a close relative called CAML.[1]

Whitehead is also one of my favorite philosophers of education, and in particular I want to talk about *The Rhythm of Education*, 1922.[2] The main idea is that there is a cycle of *romance* (exploring a new idea, being confused) and precision (mastering the details) which repeats itself, at various levels of granularity, as you learn. Last lecture and lab were romance; today we get to some precision about ML programming.

The main ideas of today's lecture are:

1. expressions versus values
2. type checking and evaluation
3. errors
4. declarations
5. functions

1 Expressions, Types, Values

The basic unit of an ML program is an *expression*. Here are some simple expressions:

[1] which has better libraries

[2] http://www.archive.org/details/rhythmofeducatio00whitiala

```
2
1 + 1
(1 + 2) * (3 + 4)
"I am"
"I am" ^ " the walrus"
intToString 5
"the walrus" + 1
5 div 0
```

Note that parens are used for grouping, and that `(1 + 2) * (3 + 4)` is different than `1 + 2 * 3 + 4`, which by convention is the same as `1 + (2 * 3) + 4`.

1.1 Computing by calculation

The way we run an ML program is to calculate it down to a *value*, which is the result of a computation. Simple values include numeric constants (written like `2`) and string constants (written like `"I am"`).

We write `<exp> ==> <val>` to mean "the expression `<exp>` calculates to the value `<val>` (note that `==>` is *not* part of the syntax of ML; it is notation we use to talk about an ML program).

For example,

```
2 ==> 2
1 + 1 ==> 2
(1 + 2) * (3 + 4) ==> 21
"I am" ==> "I am"
"I am" ^ " the walrus" ==> "I am the walrus"
intToString 5 ==> "5"
"the walrus" + 1     no value
5 div 0              no value
```

The value of an expression is determined by calculating. Each value has no calculation left to do; it is done. For each operation, we calculate the values of its subexpressions, and then apply the operation to the result.

The operations on `int` and `float` and `string` are all primitives that "do the expected thing" in one step of calculation. We write `|->` for one step of calculation.

For example,

```
    (1 + 2) * (3 + 4)
|-> 3 * (3 + 4)          (because 1 + 2 |-> 3)
|-> 3 * 7                (because 3 + 4 |-> 7)
|-> 21
```

We write `(1 + 2) * (3 + 4) ==> 21` to mean that the expression calculates to the value in an arbitrary number of steps.

The above execution trace is for *sequential* calculation, where we only calculate one operation in each time-step.

We can also do a parallel calculation, where we are allowed to calculate as many independent operations as we want in one timestep:

```
    (1 + 2) * (3 + 4)
|=> 3 * 7                (because 1 + 2 |=> 3 AND 3 + 4 |=> 7)
|=> 21
```

If the expression were `(1 + 2) * (3 + 4) * (2 + 2)`, we could do 3 additions in parallel in step 2, etc.
Exercise. In pairs, do both sequential and parallel evaluation for the expression
`("I am " ^ "he ") ^ ("as you are " ^ "she")`.

1.2 Types

What about `"the walrus" + 1`? What do we do about expressions that don't evaluate and are not values? We open up *Principia Mathematica* to page 37 and read about *types.*

The world of ML expressions is divided up into types. The type of an expression is a prediction about the value it will yield, should it yield a value at all. For example, if an expression has type `int`, then its value will be a numeral (if it has a value at all). An expression in *well-typed* if it has at least one type;[3] otherwise it is *ill-typed.* The *type checker* determines whether or not an expression is well-typed, and rejects ill-typed programs at *compile-time* (when you're writing the program). This helps you catch mistakes at compile-time, which is better than finding them at *run-time* (when someone runs the program). E.g. it's much better if *you* find the bug at compile-time, than if *the program's user* finds the bug after you ship it to them.

We write `<exp> : <type>` to mean that the expression `<exp>` has type `<type>`.

For example:

```
2 : int
1 + 1 : int
(1 + 2) * (3 + 4) : int
"I am" : string
"I am" ^ " the walrus"  : string
intToString 5 : string
"the walrus" + 1     is ill-typed
5 div 0  : int
```

In general, a type is specified by a collection of *values*, which are the possible results of an expression of that type, as well as a collection of *operations*, which are how you use things of that type.

For example:

- The type `int` has
  - Values: `0`, `27`, `~82`, ...
  - Operations: `+`,`*`,`-`, `intToString`, ...

Note that negatives are written like `~3`, while subtraction is written as `-`.

[3]When can an expression have more than one type? This is called *polymorphism.* We saw an example in lab: the `cons` function was used to construct both rows (sequences of integers) and classrooms (sequences of rows). We'll see more examples in a few weeks.

- The type `string` has
  - Values: `"I am"`, `"the walrus"`, ...
  - Operations: `^`, `size`, ...
- The type `real` (read: float) has
  - Values: `0.0`, `3.14`, `2.17`, ...
  - Operations: `+`, `*`, `-`, `/`, ... (note that these reuse the same names as for `int`; they are disambiguated based on context (if the context provides insufficient info, they default to `int`).

1.3 Type checking

The type of an expression is determined *compositionally* by looking at the types of the expressions inside it ("subexpressions"):

- each of the values listed above has the type indicated. E.g.

  ```
  27 : int
  "I am" : string
  ```

  These are *axioms*, which are unconditionally true.

- each of the operations is well-typed if its subexpressions have the "right" types. For example:
  - `<exp1> + <exp2> : int` if `<exp1> : int` and `<exp1> : int`
  - `<exp1> ^ <exp2> : string` if `<exp1> : string` and `<exp2> : string`
  - `intToString <exp> : string` if `<exp> : int`

  These rules can be used to derive the type of a compound expression:

  `(3 + 7) * 5 : int` because

  ```
      3 + 7 : int because
          3 : int is an axiom
          7 : int is an axiom
      5 : int is an axiom
  ```

Some expressions have no type:
`"the walrus" + 1` would have type `int` if

`"the walrus"` has type `int` (but it doesn't!)

`1` has type `int` (check)

Exercise. In pairs, derive a type for `"I am " ^ (intToString 5)`.

1.4 Exceptions

What about `5 div 0`? `div` means integer division. This expression is well-typed, but can't have a value, because division by 0 is undefined. So, it signals an error at *run-time*, when you're running the program. This is called *raising an exception*:

```
    5 div 0
|-> raise Div
```

Exceptions is that the propagate up to the top of your program. So if some expression somewhere in your program errors, that error will be the final result of the computation. For example:

```
    (5 div 0) + 1
|-> (raise Div) + 1
|-> raise Div
```

Later in the semester, we will talk about recovering from exceptions; for now, you should only raise them in cases where you want that to be the final result of your program.

Raising an exception is considered different that returning a value. An expression is *valuable* iff there exists some value that it evaluates to: `e` is valuable iff $\exists$ a value `v` such that `e ==> v`. So a valuable expression doesn't raise an exception. For example, `5 div 1` is valuable, but `5 div 0` is not.

You might wonder why `5 div 0` isn't a type error: why wait until run-time to signal a problem? The reason is that, to check whether `5 div e` is permissible, you'd need to know whether `e` evaluates to `0` or not. In general, you can't write an algorithm to decide this, because of something called the *halting problem*. There are fancier type systems than ML's, in which you can rule out such programs at compile-time; we'll talk more about this later in the semester.

1.5 Classes of Expressions

We can summarize this by identifying several classes of expressions, each of which is a strict superset of the next. For each, we give an example expressions that is in that class, but not the next one down.

- Nonsense.
- Syntactically correct expressions. These expressions make some basic level of sense; all of the above expressions are syntactically correct.
- Well-typed expressions. These expressions pass the type checker.
- Valuable expressions. These expressions compute to a value.
- Values. These expressions already are values.

For example:

- `(1+2` is not even syntactically correct, so if you say

  ```
  - (1+2;
  ```

you will get a *syntax error* at compile-time.

- `"the walrus"+1` is syntactically correct, but not well-typed, so you will get a *type error* at compile-time.
- `5 div 0` is well-typed, but not valuable.
- `5 div 1` is valuable, because it evaluates to `5`. But it is not already a value.
- `5` is a value.

2 Declarations

The top level of an ML program is a sequence of *declarations*.

For now, we will consider two kinds of declarations:

2.1 Val Bindings

The first is a *val* binding, such as

```
val x : int = 2 + 3
```

This means that the *variable* `x` stands for the *val*ue of the expression `2 + 3` (which must have type `int`) in the subsequent program. The general form of a `val` binding is `val <var> : <type> = <exp>`. These declarations are used to name intermediate steps in a program.

Typing The declaration is well-typed iff `<exp> : <type>`. In the scope of the declaration ("below", modulo the caveat mentioned soon), the variable `<var>` can be used in an expression with type `<type>`. E.g. in the scope of the above `val` binding, `x` can be used in an expression with type `int`.

Evaluation To evaluate a sequence of `val` declarations, you evaluate the first expression, and then *substitute* its value in for the variable in the subsequent declarations (replace occurrences of the variable with the value).

For example, consider

```
val x : int = 2 + 3
val y : int = x + 1
val z : int = x + y
```

We first calculate `(2 + 3) ==> 5`, and then proceed as if the program were

```
val x : int = 5
val y : int = 5 + 1
val z : int = 5 + y
```

Here we have substituted the one occurrence of `x` in the expression `x + 1` with the value `5`, to get the expression `5 + 1`, and the occurrence in `x + y` to get `5 + y`. Next, we calculate `5 + 1 ==> 6` and proceed with the program

```
val x : int = 5
val y : int = 6
val z : int = 5 + 6
```

which in one more step evaluates to

```
val x : int = 5
val y : int = 6
val z : int = 11
```

To summarize, the value of a sequence of declarations is a sequence of declarations of values.

Shadowing What happens when you have two different `val` bindings with the same variable?

```
val x : int = 5
val x : int = 3
```

The right way to think about this is that these are *two different variables that happened to be spelled with the same ASCII string.* The second is *not* an assignment that updates `x`, like in an imperative language. Variables in ML are *not* like "variables" in C or other languages: variables don't vary! Once a value has been bound to a variable, it is bound for life. There is no possibility of changing the value of a variable once it has been bound. Variables are like variables in math: placeholders that can be plugged in for. This difference will become sharper later, when we can declare variables inside of functions; at that point, we can construct an example to illustrate it.

So, when you evaluate

```
val x : int = 5
val x : int = 3
```

you do **not** substitute `5` for the `x` on the second line. The `x` on the second line is an entirely independent variable, not an occurrence of the first `x`.

Similarly, if you have

```
val x : int = 5
val y : int = x + 1
val x : int = 3
val z : int = x + 1
```

then the `x` in the second line is an occurrence of the variable bound in the first line, where the `x` in the fourth line is an occurrence of the variable bound in the third line. This is because, by convention, a variable refers to to the *nearest enclosing declaration.* So the value of this sequence of declarations is

```
val x : int = 5
val y : int = 6
val x : int = 3
val z : int = 4
```

We can make this apparent by *consistently renaming* the second `x` to `x'` in the declaration and at all occurrences.

```
val x : int = 5
val y : int = x + 1
val x' : int = 3
val z : int = x' + 1
```

This program has the same value as before, modulo the fact that the second `x` in the result is now called `x'`.

2.2 Type Definitions

The second kind of declaration is a *type definition.* For example, last lecture, we saw

```
type row = int sequence
```

This declaration means that the *type variable* `row` stands for the type `int sequence` in the subsequent program. The general form of a type declaration is `type <tyvar> = <type>`. In the scope of a type definition, `<type>` can be used as a type.

The scoping rules (shadowing) for type variables are the same as for value variables.

3 Functions

Thus far, we have some basic values and operations, and the ability to name intermediate computations. However, a `val` binding introduces a variable that stands for the value of one specific expression. Thus far, we have no way to capture *repeated patterns of computation.* That is where *functions* come in.

To a first approximation, functions in ML are like functions in math. For example,

$$f(x) = 2x + 6$$

can be rendered as

```
fun f(x : real) : real = (2.0 * x) + 6.0
```

This is a *fun* declaration, which introduces a function named `f`, with argument `x` of type `real`, and result, also of type `real`, given by the body expression `(2.0 * x) + 6.0`.

The main operation on functions is *function application.* For example, we can write `f 3.0` to apply the function `f` to the argument `3.0`. A function application calculates by substitution: you plug in the value of the argument for the variable. In this case

```
    f 3.0
|-> (2.0 * 3.0) + 6.0
|-> 6.0 + 6.0
|-> 12.0
```

Call-by-value It's important to know that you *calculate a function's argument down to a value before plugging in.* This is called *call-by-value evaluation.* E.g.

```
    f (1.0 + 2.0)
|-> f (3.0)
|-> (2.0 * 3.0) + 6.0
```

Another choice would be to plug in the whole expression:

```
    f (1.0 + 2.0)
|-> (2.0 * (1.0 + 2.0)) + 6.0
```

but this is **not** what ML does. How can you tell? Consider `f (5 div 0)` where `f` doesn't use its argument.

Call-by-value makes it easy to to predict when an expressions is evaluated. This is helpful for time analysis and for reasoning about non-valuable expressions (like an expression that raises an exception).

Scoping Function bodies can of course refer to variables that are in scope, including other functions. Example:

```
fun g(x : real) : real = f (f x)
```

Function arguments take precedence over bindings further out:

```
val x : real = 4.0
fun g(x : real) : real = f (f x)
```

Here `x` still refers to the function argument, not to the `val` binding above it.

Function arguments are *not* in scope below the function declaration:

```
fun g(a : real) : real = f (f a)
val y : real = a
```

Here `a` is unbound. This is the only thing that makes sense: the `a` doesn't stand for any particular value at this point, but for any number of possible values that we may later choose to apply `f` to.

Functions are values of function type! Of course, we can have functions on types other than `real`:

```
fun reapeatThreeTimes(s : string) : string = s ^ s ^ s
```

Exercise. Calculate the value of `repeatThreeTimes "hi"`.

Moreover, functions are not some special class of things, but regular old values of a type, just like everything else in ML. The type of `f` is written `real -> real` and the type of `repeatThreeTimes` is written `string -> string`. In general, we have

- The type `<type1> -> <type2>` has
    - Values: functions introduced by `fun` bindings

- Operations: function application, written `f a`. Here `f a` has type `<type2>` if `a` has type `<type1>`.

This means that functions can be passed as arguments to other functions, and returned from functions as results. We won't exploit this much for a few weeks, but we've already seen a couple of examples: `map` and `reduce` in the previous lecture and lab.

Functions can be recursive Functions can be defined by recursion (calling themselves). For example, let's write a recursive function to double a natural number:

```
(* Purpose: double the natural number n

   Examples:
   double 0 ==> 0
   double 2 ==> 4
*)
fun double (n : int) : int =
    case n of
        0 => 0
      | _ => 2 + (double (n - 1))
```

Let's gloss this a little: the first bit a comment that tells you the purpose of the function, and shows some examples. Next, we have a `fun` declaration for a function named `double` that takes an integer argument and produces an integer result. In the body of this function, we do a *case analysis* on the number `n`. The definition tells us that `n` is either `0` or `1 + k` for some `k`. Thus, we have *two cases*, one for `0`, and one for `1 + k`. The vertical bar `|` separates the two cases. In the first, the function's result is `0`. In the second, the `_` means that we're not placing any restrictions on which numbers match this case. In this case, the answer we give back is `2 + (double (n - 1))`.

In essence, what we're doing is this: we peel off `1 +`'s until we hit a `0`, and then add `2` back on for each `1 +` that we peeled off. This doubles the number.

When `n` is `1+k`, the *recursive call* `double (n - 1)` continues the process of peeling off ones on `k`. Note that this recursive call happens in exactly the same place as the self-reference in the definition of the natural numbers.

`case` is a new operation on natural numbers. It lets you distinguish whether the number is `0` or not. On the first line, you give the result for when `n` is `0`; on the second, you give the result for when it is not—the `_` means "any other number falls into this clause." It calculates as follows:

```
case 0 of 0 => <branch1> | _ => <branch2>   |->  <branch1>
case n of 0 => <branch1> | _ => <branch2>   |->  <branch2>   if n is not 0
```

For example, we can calculate as follows:

```
    double 2
|-> case 2 of 0 => 0 | _ => 2 + (double (2 - 1))
|-> 2 + (double (2 - 1))
|-> 2 + (double 1)
|-> 2 + (case 1 of 0 => 0 | _ => 2 + (double (1 - 1)))
|-> 2 + (2 + (double (1 - 1)))
|-> 2 + (2 + (double 0))
|-> 2 + (2 + (case 0 of 0 => 0 | _ => 2 + (double (0 - 1))))
|-> 2 + (2 + 0)
|-> 2 + 2
|-> 4
```

A `case` is well-typed if both branches have the same type, in which case that is the type of the `case`. E.g. here both branches have type `int`, and so the overall case does, too. We'll go over this in more detail next time.