SML Notes

Ryan Ordille

1 Basics of SML: Types, Values, and Effects

1.1 Evaluation and Execution

Most "familiar" programming languages are based on an *imperative* model of computation. In other words, sequences of commands examine and modify memory in these languages. Commands are executed for their effects on the memory. There are usually (but not always, in the case of C) distinctions between expressions and commands.

ML emphasizes the evaluation of expressions rather than the execution of commands. Because of this, it is much easier to develop mathematical techniques for reasoning about the behaviour of programs. There is usually little emphasis on memory modification in ML.

1.2 The ML Computation Model

The *type* of an expression is a description of the value it yields, if the expression ultimately yields one. In other words, the type is a "prediction" about the value of the expression. An expression is *well-typed* if it has at least one type. An ill-typed expression will be caught by the type checker.

Expressions can also engender effects, such as modifying memory, printing results, or raising exceptions. These are independent of an expression's type.

ML is sometimes called a *call-by-value* language because the value of an operator is defined by the values of its arguments.

Fully evaluated expressions yeild values, but not all expressions have to yield values. For example, `5 div 0` is an expression of type `int` since both of its arguments are integers, however division by zero is undefined and will raise an exception in ML. Expressions that raise exceptions, go into infinite loops, or diverge to infinity do not yield values.

1.3 Types

ML has some basic types:

- `real`
  - Values: `3.14, 2.17, 0.1E6, ...`
  - Operations: `+, -, *, /, =, <, ...`
- `int`
  - Values: `1, 17, 1000, ...`
  - Operations: `+, -, *, div, =, <, ...`
- `char`
  - Values: `#"a", #"b", ...`
  - Operations: `ord, chr, =, ...`
- `string`
  - Values: `"abc", "1234", ...`
  - Operations: `^, size, =, ...`
- `bool`
  - Values: `true, false`
  - Operations: `if EXPR then EXPR1 else EXPR2`

Unlike Java and other languages, ML does not preform any implicit conversions between types. For example, `3.14 + 2` is ill-formed in ML.

All branches of conditional statements *must* have the same type!

2 Scope and the Declarations of Variables

Variables in ML are closer to variables in mathematics than variables in C or other languages. Once a variable has a value bound to it, the value of the variable cannot be changed. Variable scope is also important - local bindings will “override” global bindings for the code segment that they are valid in. In a code example:

```
val k = 4
let
```

```
val k = 3
in
k*k (* final value is 9, not 16 *)
end;
k; (* now k = 4 *)
```

You can "rename" types using the `type` keyword (e.g. `type count = int`). Keep in mind that the new types are constructed simultaneously, i.e.

```
type count = int and newCount = count
```

will not work since the type constructors cannot refer to one another since they are constructed at the same time.

A value can be bound to a variable name using a value binding. For example,

```
val m : int = 3 + 2 and e : real = 2.17
```

binds the integer 5 to the variable `m` and the real 2.17 to `e`. The same type-checking rules apply. If the expression does not have a value for whatever reason, the binding will not have a value.

Keep in mind that, unlike other non-functional languages, *bindings are not assignments.* A binding of a variable cannot and will not change. For example, binding the value 3.14 to the variable `pi` first, then later writing `val pi = 3.0` will "shadow" the previous binding (i.e. calling `pi;` in the interpretor will print a `3.0`).

2.0.1 Limiting Scope

One can limit the scope of a variable or type constructer by using `let` and `local` declarations.

To limit the scope of bindings to one code block, use a `let` expression of the form `let DEC in EXP end`. One can also limit the scope of a declaration in another declaration by using a `local` expression of the form `local DEC in DEC' end`.

```
val m : int = 7
let
   val m : int = 3
   val n : int = m*m
in
   m*n
end
```

The above expression has the type `int` and the value `27`, since the bindings for `m` and `n` are local to the expression `m*n`. The original binding of `m` is shadowed by the local binding. If one were to call `m` later, the original value of `7` is used.

2.0.2 Typing

One can bind a type to another name by using a `type` binding. For example,

```
type float = real
```

will replace `float` by `real` whenever the program is run.

3 Functions

In algebra, we can have functions that vary over multiple variables and different sets of numbers. For example, the function $f(x) = x^2 + 2xy + y^2$ varies over x with a fixed y, $g(y) = x^2 + 2xy + y^2$ varies over y with a fixed x, and $h(x, y) = x^2 + 2xy + y^2$ varies over both x and y. To be explicit in terms of what values x, y might range over, we might write

$$f : \mathbb{R} \to \mathbb{R} : x \in \mathbb{R} \mapsto x^2 + 2x + 1$$

to show that f is a function over the real numbers that maps the real x to the expression $x^2 + 2x + 1$, which evaluates to a real.

In SML, as is the same in mathematics, a function is a kind of value of the type $typ \to typ'$. typ is the domain type of the function, while typ' is the range type, and typ does not always have to be the same as typ'.

Function expressions, or lambda expressions, are of the form `fn VAR : TYPE => EXP`. We can bind this function expression to argument value `var` just as any other binding: `val var = val`. The expression will evaluate, returning a value `val'`, which will be bound to `var`.

For example, `val inc : int -> int = fn x : int => x + 1` will bind a trivial increment function to the variable `inc`, and calling, say, `inc 16` will compute `16 + 1`.

SML has a special `fun` syntax for defining functions. For our increment function, we could write it as:

```
fun inc (x : int) : int = x + 1
```

Using this syntax is less cumbersome and easier to read, so it's usually preferred.

Note that we can also use `let` expressions inside of function declarations.

Functions can take multiple arguments and return multiple results by using tuples.

4 Recursive Functions

Recursive functions are those which call themselves on a smaller input. In order for a function to call itself, it must have a name by which it can refer to itself. The `fun` syntax makes this very easy:

```
fun factorial 0 = 1
  | factorial (n : int) = n * factorial (n-1)
```

Note that the above function uses pattern matching, which is explained in chapter 6. Pattern matching is almost always preferred to `if-else` blocks, especially in recursive functions.

We can prove the correctness of recursive functions using mathematical induction. Notes on this concept, with examples in SML, are found in the posted lecture notes.

We can also define *mutually recursive* functions that call one another in order to compute a result. For example, consider two functions that wish to compute if a given integer is odd or even:

```
fun even 0 = true
  | even n = odd (n-1)
and odd 0 = false
  | odd n = even (n-1)
```

Neither `even` nor `odd` are definable separately from one another, so we must use the keyword `and` to indicate that they are defined simultaneously by mutual recursion.

5 Concrete Data Types

The `type` keyword, in essence, creates an alias for a type or set of types. To define our own new type, we have to use the `datatype` declaration. Using this declaration, SML introduces one or more new type constructors (which can be mutually recursive) and one or more new value constructors.

Type constructors and value constructors can take zero or more arguments. Datatypes can also shadow old type or value constructors, similar to value bindings.

Datatypes can be recursive or non-recursive in nature. For example,

```
datatype suit = Spades | Hearts | Diamonds | Clubs
```

will introduce a new type `suit` with four nullary value constructors. We can create functions that operate on these new types, similar to any other function dealing with pre-defined types. Pattern matching and "wildcard" values may be used for case analysis.

Datatypes may be parameterized by a type. For example,

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

will introduce the unary type constructor `'a option` with two value constructors. Some values of the type `suit option` include `NONE, SOME Spades, SOME Hearts,` etc..

Option types are pre-defined in SML, and are usually used to handle functions with optional arguments, or to deal with aggregate data structures, where not all fields are relevant to all items of the structure. Optional results can also be represented using the option type.

Datatypes can, again, be recursive in nature. From COMP 250, we know a binary tree can either be empty, or have some value and a left and a right subtree. We can therefore define a tree datatype as follows:

```
datatype 'a tree = Empty | Node of 'a * 'a tree 'a tree
```

We can then, later on in the program, construct trees using the type constructor given in the datatype declaration. For example, for a tree of Strings, we can write the following lines of code into the interpreter:

```
val t1 = Node ( "hello", Empty, Empty );
val t2 = Node ( "goodbye", Node ( "world", Empty, Empty ), Empty );
val t3 = Node ( "ryan", Empty, t1 );
```

We can easily use recursive functions when dealing with recursive datatypes, and prove the correctness of them using inductive reasoning.

```
(* height: 'a tree -> int *)
(* assuming max function is defined *)
fun height Empty = 0
  | height Node (_, Left, Right) = 1 + max (height Left, height Right)
  ;
```

Be wary of redundant clauses (which, thankfully, the SML interpreter will warn you of) when dealing with recursive datatypes! Understanding the difference between a value constructor and a variable will make these warnings less common.

Using lists, we can "step past" binary trees to define trees for a variable number of children. Shadowing the original `tree` declaration,

```
datatype 'a tree = Empty | Node of 'a * 'a tree list
```

will allow us to create a tree with a list of children and subtrees.

Datatypes can, again, be mutually recursive using the `and` keyword - see the trees vs forests example in chapter 10.3.

The type system in SML requires all trees be homogeneous - all the data items must be the same at every node. We cannot have a node with a String value with a child node with an integer value. Although we very rarely need to deal with heterogeneous data structures, we can easily create another datatype to store a wide range of type values, and then use that in place of `'a`.

6 Higher-Order Functions

Remember that SML can see functions as values. Because of this, functions can be passed as arguments to other functions, or returned as results of functions. Functions can also be stored within data structures.

Functions which take functions as arguments or yield functions as results are called *higher-order functions*. They're not as uncommon as you think - for example, the derivative function in calculus that, when given as input a differentiable function on the real line, yields the first derivative of that function as another function on the real line. Outside of mathematical programming, higher-order functions may seem unfamiliar, since many languages have only limited ability to use functions of this type.

Higher-order funcions may be used to abstract patterns of control or to stage computation. We'll see examples of these uses later on.

6.1 Binding and Scope

Recall the concept of scope, and remember that functions can also be values.

As an example, figure out what's being printed out:

```
val x = 2;
fun f y = x + y;
val x = 3;
val z = f 4;
z;
```

If you had no idea how scope worked in SML, you'd assume the interpreter would spit out `7`. In other paradigms, this would be true. However, since with functional programming languages, functions are treated as values, `f` uses the nearest previous binding of `x`, which is `2`. Therefore, when `f` is called in the binding of `z`, the original binding of `x = 2` is used, so `z;` would print out `6`.

The key concept to remember is that binding is not assignment. If we were to view the seconding binding of `x` as an assignment statement, then `z`'s value would be `7`, not `6`. The "shadowed" binding of `x` is therefore not lost, since it's used in any function declarations between that binding and the next binding.

7 Returning Functions

The principle of lexical scope is considerably important when we want to return functions as results of other functions.

Consider the `map'` functions, which applies a given function to every element of a list.

```
fun map' (f, nil) = nil
  | map' (f, h::t) = (f h) :: map' (f, t)
```

For example, the call `map' (fn x => x + 1, [1,2,3,4])` evaluates to `[2,3,4,5]`.

We can create new functions during execution - not just return functions that have been previously defined.

For example, consider the trivial example of the `constantly` function. The function will, when given a value `k`, yield a function that yields `k` whenever it is applied.

`val constantly = fn k => (fn a => k)` or `fun constantly k a = k`

The value of `constantly 3` is a function that always yields `3` when called. This function is created by the `constantly 3` call - we have never explicitly defined earlier in the source fine a function which only returns `3`. `constantly 3` is a *closure* consistently of the function `fn a => k` with the *environment* `val k = 3` attached to it.

Back to the `map'` example, remember that `map'` accepts a function and a list as arguments, and returns a new list as a result. It would be terribly inconvenient to constantly pass the same function to `map'` with different lists, so we can create an instance of a map function specialized to a given function that can be applied to many lists. We can redefine the function as follows:

```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
fun map f nil = nill
  | map f (h::t) = (f h) :: (map f t)
```

The function `map` accepts a function of the type `'a -> 'b` as an argument, then *yields another function* of type `'a list -> 'b list` as a result.

Notice that we have changed a two-argument function into a function that takes two arguments in succession, yielding after the first a function that takes the second argument as its sole argument. This is a process called *currying*, and the passage can be codified as follows:

```
(* curry: ('a * 'b -> 'c) -> ('a -> ('b -> 'c)) *)
fun curry f x y = f (x, y)

fun map f l = curry map' f l
(* equivalently *)
fun map f l = ((curry map') f) l
```

7.1 Abstracting Patterns of Control

Consider the following two functions `add_up` and `mul_up`. What exactly is the similarity between them?

```
fun add_up nil = 0
  | add_up (h::t) = h + add_up t
fun mul_up nil = 1
  | mul_up (h::t) = h * mul_up t
```

We have some binary operation to be applied over some list - addition for `add_up` and multiplication for `mul_up`. The result on the empty list is some unit element - `0` for `add_up` and `1` for `mul_up`. The operation is preformed on the head of the list and the result on the tail. We could also say that the two functions are both defined recursively, i.e. they are defined by induction on the structure of the list argument with a base case for `nil` and an inductive case defined in terms of the tail list.

We can further abstract the pattern to work with any binary function - either those built in or those we define ourselves. We just need to define the basic unit for the base case and pass a function as one of the arguments.

```
(* reduce: 'b * ('a*'b->'b) * 'a list -> 'b *)
fun reduce (unit, operation, nil) = unit
  | reduce (unit, operation, h::t) =
     operation (h, reduce (unit, operation, t))
```

We can re-define our two earlier functions as follows:

```
fun add_up l = reduce (0, op +, l)
fun mul_up l = reduce (1, op *, l)
```

We say that the function `reduce` *abstracts the pattern* of defining a function by induction on the structure of a list.

Can `reduce` be better defined? The arguments `unit` and `operation` are continually passed at every recursive step, but they remained unchanged at every step. The only argument to change is the list argument. Remember that multi-argument functions are really single-argument functions that take a tuple as an argument! Every time `reduce` is called in the recursion, we are constructing a new tuple where two of the three components never change.

```
fun better_reduce (unit, operation, l) =
    let
        fun red nil = unit
          | red (h::t) = operation (h, red t)
    in
        red l
    end
```

The new inner function `red` no longer carries bindings for `unit` and `operation`, saving the extra overhead of creating larger tuples on each iteration of the loop. Also, `red` is bound to a closure consisting of the code for the function together with the environment active at the point of definition.

7.2 Staging

We can also *stage* the computation for `reduce`. Since `unit` and `operation` remain fixed for different lists, we can say they are "early" arguments. The list `l` is said to be a "late" argument. If we stage the computation, we can preform as much computation as possible on the early arguments, yielding a function on the late arguments that is "leaner" and easier to compute.

By staging `reduce`, we can yield a function that may be later applied to many different lists. We build `red` on the basis of `unit` and `operation`.

```
fun staged_reduce (unit, operation) =
    let
        fun red nil = unit
          | red (h::t) = operation (h, red t)
    in
        red
    end
```

The closure bound of `red` occurs as soon as `unit` and `operation` are known, rather than each time the list argument is supplied. Curried functions do not have a computation advantage of over staging. If you unravel the following example, you can see that, while we take two arguments in succession, we are not doing any useful computation in between the arrival of the first argument (the pair of `unit` and `operation`) and the second argument (the list `h::t`).

```
fun curried_reduce (unit, operation) nil = unit
  | curried_reduce (unit, operation) (h::t) =
      operation (h, curried_reduce (unit, operation) t)
```

Notice that, although the staged and the curried versions of `reduce` have the same iterated function type

```
('b * ('a * 'b -> 'b)) -> 'a list -> 'b
```

we can see the computation in the staged version is once, while the curried version does not do any work in between the arrival of its arguments. Some, *but not all*, functions of iterated function type are curried. The staged example above is not curried.

Although the computation time saved by `staged_reduce` is minor, we can gain a significant computation advantage by staging intensive functions. Consider a function that appends lists together. In the following code, `append` only works with two lists, and takes both arguments at once. The curried version of our `append` function does not save us any computation - it simply changes the argument syntax, instead of exploiting the fact that the first argument is fixed for many second arguments. The staged version will traverse the first list just once for all applications to a second argument.

```
fun append (nil, l) = l
  | append (h::t, l) = h :: append(t,l)

fun curried_append nil l = l
  | curried_append (h::t) l = h :: curried_append t l

fun staged_append nil = (fn l => l)
  | staged_append (h::t) =
    let
    val tail_appender = staged_append t
in
    fn l => h :: tail_appender l
    end
```

The `staged_append` function still takes $O(n)$ time, but reduces the amount of constant time operations per call by avoiding the pattern matching required to destructure the original list argument.

8 Exceptions

Remember that expressions in SML have a type, but may or may not have a value and/or an effect. Here, we'll introduce the concept of an effect. Effects are actions resulting from evaluation other than returning a value. Some examples of effects in SML are:

Exceptions

Mutation allocating and modifying storage

Input/Output

Communication

8.1 Exceptions as errors

SML is a safe language in that its exectuion behavior can be understood entirely in terms of the constructs of the language itself. SML's static type discipline rules out expressions that are ill-defined, and SML's dynamic checker rules out other violations, like division by zero. Dynamic violations are signalled by raising exceptions.

8.1.1 Primative exceptions

Expressions such as `3 + "3"` are ill-typed, so SML's type checker will simply halt evaluation. However, expressions such as `3 div 0` are well-typed but will fault at run-time, raising an exception (in this case, `Div`).

Inexhaustive matches are another source of run-time exceptions. In the function `fun hd (h::_) = h`, there is no provision for an empty list. If we try to call `hd` on `nil`, SML will raise the exception `Match`. We are allowed to create functions that do not pattern match on all possible cases so long as we make sure not to use them on a value outside of its domain.

8.1.2 User-defined exceptions

We can also create our own exceptions.

```
exception Factorial
fun checked_factorial n =
    if n < 0 then raise Factorial
    else if n=0 then 1
    else n*checked_factorial (n-1)
```

The above function is not the best - it could be better defined using pattern matching, and it constantly checks the arguments at each recursive call, even though it only has to be done once. We can improve this using an inner function.

8.2 Exception handling

Exceptions can not only be used to terminate a program, but also to affect non-local transfers of control. We may catch a thrown exception and continue evaluation along some other path. If no handler handles a thrown exception, then the uncaught exception is signaled as the final result of evaluation, and the computation is aborted. Exceptions force you, as the programmer, to consider the exceptional case, and allow you to segregate that case from the normal case in your code.

```
exception Change
fun change _ 0 = nil
  | change nil _ = raise Change
  | change (coin::coins) amt =
           if coin > amt then
              change coins amt
           else
              (coin :: change (coin :: coins) (amt-coin))
              handle Change => change coins amt
```

In the above example, if we get "stuck" in the computation, we simply undo the most recent greedy decision and try again on the next coin.

8.3 Value-carrying exceptions

We can attach additional information to exceptions to be handled by the exception handler. For example, we can attach a String value to an exception, explaining what went wrong when an exception is raised.

```
exception SyntaxError of String
```

The above will introduce the exception which carries a String as a value. This is similar to constructors of datatypes, and, in fact, the above is an example of an exception constructor. The type `exn` is an example of an extensible datatype, i.e. one whose constructors are not determined "once and for all" like built-in datatypes, or those declared with the `datatype` keyword. A value of, say, `SyntaxError "Integer expected"` is of type `exn`.

9 References and mutable storage

Expressions in SML always have a type and evaluate to a value, and can also have an *effect*, i.e. an action resulting from evaluation of an expression other than returning a value. Here, we'll be looking into mutable storage, which can be allocated, updated, and referenced outside of the normal evaluation scheme.

Recall let-expressions – these expressions let us bind local variables that only exist within a certain scope. In the beginning of the course, when we were describing the scope rules of SML, we used the "area" example:

```
let
   val pi = 3.14
   fun area r = pi * r * r
   val a2 = area (2.0)
   val pi = 6.0
in
   area 2.0
end
```

The value of `a2` and the latter `area 2.0` are exactly the same – `12.56`, even though we've overshadowed `pi`. This overshadowing does not affect the binding of `pi` in the function declaration. To allow us to change that value, we have to use *references*.

9.1 Reference cells

Reference cells allow us to support mutable storage. We can read from these cells and replace their values as we would expect to be able to do in other programming paradigms. Before, when writing code in SML, we were dealing with an effect-free environment; now we have to worry about previous and future values of a cell.

To create a reference cell, we must allocate memory for it by using the constructor `ref`. The reference constructed will be of type `'a ref`, with `'a` being whatever type you wish to reference to.

```
val r : int ref = ref 0
val s : int ref = ref 0
```

Both `r` and `s` are of type `int ref` and point to different cells in memory, which just happen to contain the same value `0`. Evaluating the expression `r = s` will return false, since they do not point to the same cell.

Using the `!` operator, we can read the content of a reference cell. `!r` will yield the current content of the reference `r`, which is `0`. In other words, `val k = !r` will bind the integer `0` to `k`.

Using the `:=` operator (also known as the *assignment* operator), we can change the content of a cell. The effect of this is similar to the `=` operator in most imperative languages, like C. This is an infix operator, and will destroy the old contents of a cell before replacing it. The expression `val _ = r := 3` will change the value of `r` to `3`. Keep types in mind, too, since we cannot put a value of type real into a integer cell, a list into a real cell, and so on.

We can also use aliasing to point two or more variables to the same reference cell. If we compare these two variables in, say, an if-statement, then the result will be `true`.

We also have a shorthand operator to sequentially compose expressions. Instead of

```
let
   val _ = exp1
in
   exp2
end
```

we can use the expression `exp1 ; exp2` to first evaluate `exp1` for its effect, then evaluate `exp2`.

Back to the original area example, we can rewrite this using reference cells:

```
let
   val pi : real ref = ref 3.14
   val area = fn (r : real) => (!pi) * r * r
   val a2 = area 2.0
   val _ = pi := 6.0
   val a3 = area 2.0
in
   print ("a2 = " ^ Real.toString a2 ^ "\n");
   print ("a3 = " ^ Real.toString a3 ^ "\n");
end
```

`a2` will still be bound to `12.56` like the original example, and `a3` will use the new value of `pi` and will have the value `24.0`. Note that the old value of `pi` will be destroyed when the new assignment is computed, since this is a destructive update. We cannot get `3.14` back without editing the source code.

In imperative languages such as C, it's often harder to reason about programs since variables in these languages are always implicitely bound to reference cells and not values themselves, so variables are always implicitely derefernced whenever they are used. These imperative variables always stand for their current contents. In SML and other functional languages, references are used quite sparingly, and it's usually much simplier to reason about programs written in these languages.

9.2 Programming style with references

If we really wanted to, we could completely mimic imperative languages using SML's built-in references. We could define an imperative-style factorial function:

```
fun imperative_fact n =
    let
       val result = ref 1
       val i = ref 0
       fun loop () = if !i = n then ()
                  else (i := !i + 1; result := !result * !i; loop ())
     in
        loop (); !result
 end
```

This `loop` function is pretty much just a while-loop, which repeatedly executes its body n times. We would never use this function in the real world, however, since the factorial function just computes a simple function on natural numbers, so there's no reason to maintain memory state.

9.3 Mutable data structures with references

With our previous data structures such as lists, it is impossible to change the contents of these structures without creating an entire new structure and copying elements over. Operations performed on these structures do not change or destroy the original structure.

Mutable data structures allow us to maintain some sort of structure that can be changed without completely copying it every time we wish to perform an operation on it. We can create mutable data structures in SML using references.

Consider a datatype for a simple mutable reference list:

```
datatype 'a rlist = Empty | RCons of 'a * ( ('a rlist) ref )
```

We can create new reference lists using the `RCons` constructor, and we can point to other reference lists (or even the same list) using the second argument for the constructor. For example:

```
(* node with value 4, next node is Empty *)
val l1 = ref (RCons (4, ref Empty))
(* node with value 5, next node is l1 (value is 4, next node is Empty) *)
val l2 = ref (RCons (5, l1))
(* create a circular list *)
val _ = l1 := !l2
```

Using simple list diagrams, we can model exactly how the lists look at each stage.

We can create a function that appends two reference lists. To do this, we can just traverse to the end of a list, and change the last pointer to the head of the second list:

```
type 'a refList = ('a rlist) ref
(* rapp: 'a refList * 'a refList -> unit *)
fun rapp (r1 as ref Empty, r2) = r1 := (!r2)
  | rapp (r1 as (ref (RCons(h,t))), r2) = rapp (t, r2)
```

Note that, in the above function, nothing is copied over - the first list is simply updated. We can also destructively modify a reference list – consider a reverse function:

```
(* rev: 'a refList -> unit *)
fun rev l =
    let
       val r = ref Empty
       fun rev' (ref Empty) = (l := !r)
         | rev' (ref (RCons (h, t)) = (r := RCons(h, ref(!r)); rev' t)
    in rev' l
    end
```

9.4 Objects in SML

Using references and closures, we can implement objects in Standard ML. Consider a counter object, which can increment (or tick) and reset itself. The tick function takes in unit and returns an int, consisting of what the new, incremented clock is displaying, and the reset counter takes in unit and returns unit, having the side effect of setting the counter back to zero. We can create instances of this counter object that are all independent of each other using a generator function:

```
(* newCounter: unit -> {tick: unit -> int, reset: unit -> unit} *)
```

```
fun newCounter () =
    let
       val counter = ref 0
       fun tick () = (counter := !counter +1; !counter)
       fun reset () = (counter := 0)
    in
       {tick = tick, reset = reset}
    end
```

The closure syntax is a bit unusual, but it makes sense when you make the analogy with object-oriented programming - `newCounter` can be thought of as a constructor for a class of counter objects, and each instance of the object has private instance variable `counter` and two "methods" `tick` and `reset`.

10 Lazy evaluation

SML, and many other languages, evaluate expressions by a *call-by-value* discipline. Variables are bound to values of fully evaluated expressions. In a let-expression, the variables declared under `let` are fully evaluated, then the expression under `in` is evaluated, even if some or none of the variables are actually used in the latter expression. Using this strategy:

```
let val x = horribleComp(345)
in 5
end
```

The value for `x` is computed, even though it is not needed. This style of evaluation is also known as *eagar evaluation*. We can try a different strategy, called *call-by-name*, where variables are bound to unevaluated expressions, i.e. computation is suspended until it is needed. In the above example, `x` is bound to `horribleComp(345)`, not the value of that expression. Since `x` is never used, it is never evaluated.

This strategy is a step above the call-by-value discipline, but it still fails in situations where a variable is called multiple times. In this case, since the variable is bound to an unevaluated expression, not a value itself, every time the value of a variable is needed, the value must be re-evaluated. For example:

```
let val x = horribleComp(345)
in x+x
end
```

Here, in a purely call-by-name strategy, the value of `x` is evaluated twice. This time-saving strategy has actually cost us computation time.

In practice, *lazy* (i.e. not eagar) languages are a bit "smarter". Lazy languages instead adopt a *call-by-need* principle, which acts as a "hybrid" between call-by-name and call-by-value. Variables are bounded to unevaluated expressions, which are evaluated once the value is needed. Instead of throwing away the value, the value is *memoized.* In other words, the variable is bound to an unevaluated expression until that expression's value is needed, in which case the variable is bound to the value instead.

In the above example, `x` is bound to the expression `horribleComp(345)` until it is needed. In the expression `x+x`, the expression `horribleComp(345)` is evaluated to some value `v`, which then bound to `x`. Then, `x+x` can be thought of as `v+v` instead of `horribleComp(345) + horribleComp(3`

Lazy evaluation allows us to delay computation of a value until that value is actually needed. Using this *demand-driven* computation style, we can deal with infinite and interative data structures, which cannot be dealt with using eagar evaluation strategies.

Remember, however, that SML is an eagar language. Therefore, in order to support lazy evaluation, we have to model it with tools such as functions and datatypes. We have to "manually" suspend computation – SML will not automatically do it for us. We need to create a datatype that will "shield" some expression by wrapping the expression in a function.

```
datatype 'a susp = Susp of (unit -> 'a)
(* delay: (unit -> 'a) -> 'a susp *)
fun delay f = Susp (f)
(* force: 'a susp -> 'a *)
fun force (Susp (f)) = f ()
```

11 Formal Syntax and Evaluation

At this point, we have seen many key concepts about programming in different paradigms. However, we would like to have a more theoretical foundation – we would like to be able to answer more broad questions about the programs we are writing, such as:

- What is the meaning of a program?
- How will it execute and what is its behavior?
- How can we reason about its execution?
- What are the legal expressions that I can write?

- What is our concept of a variable?
- What are well-typed expressions?
- How does SML infer the type of a given expression?
- How is the program going to be executed?

Understanding these theoretical aspects will allow us to write better programs that are well-structured, more likely to be correct, and easier to debug.

A programming language has three "parts", i.e. three things that need to be defined in order to define the language.

1. Grammar of the language
2. Operational dynamic semantics
3. Static semantics

We'll design a language called Nano-ML which is a subset of Standard ML in order to better understand these theoretical principles. Much of what we'll define is relevant in other languages and paradigms.

11.1 Inductive definitions of expression

First, what is an expression in Nano-ML? In the Backus-Naur Form (BNF):

```
Operations op ::= + | - | * | < | =
Expressions e ::= n | e1 op e2 | true | false | if e then e1 else e2
```

`3+2, true + 2, if true then 5+3 else false` are all well-formed expressions, while `if true then else` and `+23` are ill-formed. Well-formed expressions are not necessarily well-typed, but we'll get to that in a bit. Notice that `-4` may be semantically sensible, but it does not fit into our grammar and thus will be considered well-formed, since the `op -` requires two arguments.

11.2 Operational Semantics

Now we should describe how a given expression is executed. We'll need to define a high-level operational semantics for Nano-ML. The judgement $e \Downarrow v$ (or, in verbatim-mode, `e evalsto v`) means "the expression e evaluates to a final value v".

So far in Nono-ML, we have two possible values – some integer n or the choice of the booleans `true` or `false`. More formally:

`Values v ::= n | true | false`

If we already have a value, then we return its value. If we have some expression, we recursively evaluate the sub-expressions to values until we have one return value.

Our inference rules have the following shape:

$$\frac{premise \quad \ldots \quad premise}{conclusion}$$

To achieve the conclusion, we must satisfy each premise.

$$\frac{}{n \Downarrow n}\ B-NUM$$

$$\frac{}{true \Downarrow true}\ B-TRUE$$

$$\frac{}{false \Downarrow false}\ B-FALSE$$

$$\frac{e1 \Downarrow v1 \quad e2 \Downarrow v2}{e1\ op\ e2 \Downarrow v1\ \bar{op}\ v2}\ B-OP$$

The $B-OP$ rule is another way of wording what we said before - we can find the value of an expression by recursively finding the values of the sub-expressions. We also have $B-IFTRUE$ and $B-IFFALSE$, which are self-explanatory.

Note that we have not imposed an order of evaluation of the premises – this is a feature of "Big=step" evaluation, which abstracts over the order of evaluation of sub-expressions.

Evaluation will fail when expressions do not yield a value – e.g. the derivation for `if 0 then 3 else 4` fails as $0 \Downarrow 0$ and the rules for derivation require that the "guard" evaluations to a boolean.

We have a couple properties, whose proofs are beyond the scope of COMP 302:

Value soundness If $e \Downarrow v_1$, then v_1 is a value.

Determinacy If $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$.

12 Towards Mini-ML

Right now, Nono-ML only has basic expressions and values. We'll extend Nano-ML to Mini-ML by adding variables and let-expressions, and we'll deal with issues regarding their uses.

12.1 Syntax for variables and let-expressions

`Expressions e ::= ... | x | let x = e1 in e2 end`

`let z = if true then 2 else 43 in z + 123 end` and `let x = x+3 in y + 123 end` are examples of well-formed expressions using our new expression definition. `let x = 3 x + 2 end` and `let z = if true then 2 else 43 in - 123 end` are ill-formed.

12.2 Free variables and substitutions

We can soon define the operational semantics for evaluating let-expressions, but first we must discuss *variable binding*. When is a variable bound and when is it free?

First we'll need to define the set of free variables in an expression e inductively. The function FV takes as input an expression e and returns a set of free variables occurring in e.

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(e_1\ op\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{if e then e1 else e2}) &= FV(e) \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{let x = e1 in e2 end}) &= FV(e_1) \cup (FV(e_2)/\{x\})
\end{aligned}$$

It's easiest to clarify the difference between free and bound variables in an example:

```
let x=5 in (let y=x+3 in y+x end) end
```

Going "backwards", the expression $y + x$ contains free occurrences of x and y. y is then bound in the expression `let y=x+3 in y+x end`, so only x is free. y's scope is only within the body of the let-expression.

To substitute an expression for a free variable in another expression, we'll write $[e'/x]e$. This means that we'll replace all free occurrences of x in e with the expression e'.

$$\begin{aligned}
[e'/x]x &= e' \\
[e'/x](e_1\ op\ e_2) &= [e'/x]e_1\ op\ [e'/x]e_2 \\
[e'/x](\texttt{if e then e1 else e2}) &= \texttt{if [e'/x] e then [e'/x] e1 else [e'/x] e2}
\end{aligned}$$

For let-expressions, substitution is a bit tricky. In the expression `let y=e1 in e2 end`, we wish to apply the substitution $[e'/x]$. Just applying the substitution to the sub-expressions can lead to *variable capture*, where previously free variables become bound if the names are changed. It's easiest to illustrate this with an example:

$$[5/x]\texttt{(let y=x+3 in y+x end}$$

`let y = [5/x](x+3) in [5/x](y+x) end` = `let y = 5+3 in y+5 end` (this seems to work fine)

If we were to replace x with $y+1$, however, we'll run into a variable capture issue:

$$[y+1/x](\texttt{let y = x+3 in y+x end})$$

`let y = [y+1/x](x+3) in [y+1/x](y+x) end` = `let y=(y+1)+3 in y+(y+1) end`

Notice here that y was free in $y+1$, but is bound in the result. The name of bound variables does not matter.

To ensure substitution works for let-expressions, we'll need to impose the following condition:

`[e'/x](let y=e1 in e2 end) = let y=[e'/x]e1 in [e'/x]e2 end` provided $x \neq y$ and $y \notin FV(e')$.

12.3 Evaluation of let-expressions

For the let-expression `let x = e1 in e2`, we first evaluate e_1 to v_1, then replace all free occurrences of x in e_2 by v_1 and then evaluate this to some value v_2. Formally:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\texttt{let x=e1 in e2 end} \Downarrow v}\ B-LET$$

12.4 Functions

```
Expressions e ::= ... | fn x => e | e1 e2 | rec f => e
```

$$\begin{aligned} FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\ FV(\texttt{fn x => e}) &= FV(e)/\{x\} \\ FV(\texttt{rec f => e}) &= FV(e)/\{f\} \end{aligned}$$

For `fn x => e`, the input x is bound, while for `rec f => e`, the function name f is bound.

For substitution, remember to be careful to avoid variable capture:

$$\begin{aligned} [e'/x](e_1 e_2) &= [e'/x]e_1\ [e'/x]e_2 \\ [e'/x](\texttt{fn y => e}) &= \texttt{fn y => [e'/x]e}\text{provided } x \neq y \text{ and } y \notin FV(e'). \\ [e'/x](\texttt{rec f => e}) &= \texttt{rec f => [e'/x] e}\text{provided } x \neq f \text{ and } f \notin FV(e'). \end{aligned}$$

Functions are considered first-class values, and will evaluate to themselves:

$$\frac{}{\texttt{fn x => e} \Downarrow \texttt{fn x => e}}\ B-FN$$

$$\frac{e_1 \Downarrow \texttt{fn x => e} \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}\ B-APP$$

For recursive functions, we'll use substitution to replace any occurrence of the function name with the actual function definition.

$$\frac{[\texttt{rec f => e}/f]e \Downarrow v}{\texttt{rec f => e} \Downarrow v}\ B-REC$$

13 Types

So far, with out Nano/Mini ML language, we've only been concerned with syntax and operational semantics. Typing allows us to prevent expressions which cannot be evaluated from ever being brought to run-time. Types approximate run-time behaviour and allow us to better reason about the behaviour of the programs we write.

13.1 Basic types

Recall our original Nano-ML language:

```
Operations op ::= + | - | * | < | =
Expressions e ::= n | e1 op e2 | true | false | if e then e1 else e2
```

When is an expression well-typed?

Types classify expressions according to their value. We can add another item to our form list:

```
Types T ::= int | bool
```

We'll also say that the judgement $e : T$ means "expression e has type T".

From what we have, $n : \mathit{int}$, $\mathit{true} : \mathit{bool}$, $\mathit{false} : \mathit{bool}$. For if-expressions:

$$\frac{\texttt{e : bool e1 : T e2 : T}}{\texttt{if e then e1 else e2 : T}}\ T-IF$$

The other primitive operations are trivial to show.

13.2 Typing for tuples and projections

```
Expressions e ::= ... | (e1, e2) | fst e | snd e
```

We can create tuples via (e_1, e_2) and take them apart using `fst e` and `snd e`. Tuples are another value, just like numbers and booleans. Tuples are classified under the product type, written $T_1 \times T_2$.

$$\frac{e_1 : T_1 \quad e_2 : T_2}{(e_1, e_2) : T_1 \times T_2}\ T-PAIR$$

$$\frac{e : T_1 \times T_2}{fst\ e : T_1}\ T-FST$$

$$\frac{e : T_1 \times T_2}{snd\ e : T_2}\ T-SND$$

13.3 Typing for variables and let-expressions