Main topics of the week:
- A simple interpreter in ML
- Assignment in ML
- Type Inference

Grammar Basics
Suppose we want to have a grammar describing an arithmetic expression (say addition). The grammar might look like:

```
Expr ::= n | Expr + Expr
```

We will assume that the goal is to have types so that we can define a function to perform the operation, i.e., a function that takes an Expr and returns an int. You might think we would do this in ML with the datatype:

```
datatype Expr = int | Expr * Expr;
```

However, this is incomplete and gives a syntax error. To understand why ML doesn’t like this, consider the expression 5 – this is an int, so presumably it is an Expr. But an Expr is not an int. Rather we want to use constructors to show how to make an Expr from an int as well as from two Expr’s. To do this in ML, we define a data type as follows:

```
datatype Expr = N of int | P of Expr * Expr;
```

(N is short for number, P is short for pair) This uses constructors to keep the types straight and is an instance of patterns. In particular, we cannot just use int since then int’s would be directly used in the function call and so would be evaluated by ML prior to the call. Note that these patterns are then used in the function definition, which is expressed as a disjunction of the disjuncts appearing in the type definition.

```
fun Interp (N x) = x
  | Interp (P(E1, E2)) = Interp(E1) + Interp(E2);
val Interp = fn : Expr -> int

Interp(P(N 4, N 5));
val it = 9 : int

Interp(P(P(N 2, N 3), N 1));
val it = 6 : int
```

Note that by defining things this way, evaluation is deferred until the call, i.e., the function performs the operation. Constructors are just ways of specifying types, they are actually implemented as functions that take the argument and just return it. The last of the expressions is essentially the arithmetic expression (2+3)+1. What we are really doing here is defining Interp to evaluate the addition operation, and using P to construct sum expressions.

Returning to the datatypes for the expression, we could expand it to cover two operations, e.g., addition and subtraction by (we shorten to just E instead of Expr):

```
datatype E = N of int | P of E*E | M of E*E;
```

or

```
datatype binop = P | M;
```

Note these are all patterns of types. It’s a way to break up types into components. In a sense it suppresses evaluation and turns programs (code) into data (types).
Let’s try expanding the evaluation of the arithmetic expression, but have the “minus” operation produce an error.

```ml
datatype E = N of int | P of E * E | M of E * E;
fun Interp(N x) = x
| Interp(P(e1,e2)) = Interp(e1) + Interp(e2)
| Interp _ = "error";
```

Error: right-hand-side of clause doesn’t agree with function result type

Obviously, this doesn’t work. What is wrong is that there is a type mismatch – Interp sometimes returns an int, and sometimes a string. So we try defining a return type to capture this and adjusting the first and third clauses:

```ml
datatype R = N of int | Err;
fun Interp(N x) = N x
| Interp(P(e1,e2)) = let val x = Interp(e1)
| val y = Interp(e2)
| in case (x,y) of
| (N1 p1, N1 p2) => N1(p1 + p2)
| _ => Err
| end
```

Error: operator and operand don’t agree

But we get the error since we don’t know how to “add” two R’s in the second clause. We correct this by introducing a new type for int in R for the return and adjusting the second clause to only perform the addition if we didn’t get an error on evaluating Interp on the components:

```ml
datatype E = N of int | P of E * E | M of E * E;
fun Interp(N x) = x
| Interp(P(e1,e2)) = let val x = Interp(e1)
| val y = Interp(e2)
| in case (x,y) of
| (N1 p1, N1 p2) => N1(p1 + p2)
| _ => Err
end
```

Alternatively, we could simply raise an exception:

```ml
exception Err;
fun Interp(N x) = x
| Interp(P(e1,e2)) = raise Err;
```

Changing without Assignment

Since ML does not support assignment to variables (we’ll see side effects later), what can we do without assignment? Suppose we have a tree structure with integer values and we want to write a function that changes the values in the tree by adding one to each value. We can do this as follows:

```ml
datatype Tree = Leaf of int | Node of int * Tree * Tree;
```
A simple interpreter

- fun change (Leaf x) = Leaf (x+1)
  = |change(Node (x,l,r)) = Node (x+1, change l, change r);
val change = fn : Tree -> Tree

The thing about this function is that it doesn’t really change anything – it actually produces a new copy of the tree with the values adjusted. This may seem inefficient, but the philosophy is that the programmer should try to program as elegantly as possible and not worry about this type of economy of storage or efficiency. The job of the compiler is to make this efficient. After all, the compiler is in a better position to know if doing an update in place is possible without violating the integrity of the program – it knows when things are referenced, for example.

Let’s reconsider the tree type, which is monomorphic, i.e., applies only to trees of integers. We can make it polymorphic as follows:

- datatype 'a Tree = Leaf of 'a | Node of 'a * 'a Tree * 'a Tree;
val change = fn : ('a -> 'b) * 'a Tree -> 'b Tree

Note that the return value of change is a beta tree, that is, ML cannot guarantee that it is of the same type as the tree passed to it. Also note that we have added an argument to change, namely a function from one unspecified type to another. We can use an anonymous function, e.g.,

- change(fn(x)=>true, Leaf 5);
val it = Leaf true : bool Tree

Grammar for the Interpreter

We want to write a simple interpreter to be able to experiment with the mechanisms of different scoping. This interpreter will be a function in ML that evaluates an expression. The expression will be given in abstract syntax. That is, we don’t want to mess with parsing input, so we will provide the input expression in an abstract form. So the first thing to do is to figure out a reasonable grammar for our language fragment and express it as datatypes in ML. We want our language to allow variables, constants, the addition operation, function definitions, local blocks, and function application. So we start with the grammar:

```
datatype E =
  Variable of identifier
  | Literal of int
  | Plus of E * E
  | Lambda of identifier * E
  | Lett of ((identifier * E) list) * E
  | App of E * E ;
```

Note that we have the components we just listed. These are all the forms an expression can take. A Variable is just an identifier (a string). The constants here (Literal) are just integers. We have one operation, Plus, that is a binary operation of two expressions. The function definition consists of a single parameter and an expression. The local block consists of a list of identifiers and expressions, followed by an expression. And finally, the application of a function consists of two expressions – the function name and the parameter.
A simple interpreter

Week 9
Selected Lecture Notes

We need a notion of an activation frame, so we define a type to encapsulate the environment. The environment entry is really just an identifier and a value, and the environment is just a list of these entries. This is defined mutually with a result type which we use for the value since we want to allow for errors and closure when we look up values in the environment.

```
datatype env = Env of (identifier * result) list
  and   result = Value of int
            | Closure of E * env
            | Error of int;
```

We will define several utility functions for accessing and adding to the environment: lookup_env will search for a given id in a given environment, raising a not found exception if nothing is there.

Now we’re ready to look at the implementation of the interpreter. This is defined as a function interp1 (we wrap it with a function interp to handle the environment).

```
fun interp1(exp,env) =
  case  exp of
    Variable(id)     => lookup_env(env,id)
  |  Literal(Int)   => Value(Int)
  |  Lambda(id,exp) => Closure(Lambda(id,exp),env)
  |  Plus(E1, E2)   =>
    let val V1 = interp1(E1,env)
    val V2 = interp1(E2,env)
    in
    case (V1,V2) of (Value(X), Value(Y)) =>  Value(X+Y)
                   | _     => raise err(BadArguments)
    end
  |  App(E1, E2)  =>
    let val V1 = interp1(E1,env)
    val V2 = interp1(E2,env)
    in
    case V1 of Closure(Lambda(id,exp),clos_env) =>
      let val new_env = extend_env(clos_env,id,V2)
      in
      interp1(exp, new_env)
      end
    | _     => raise err(BadOperator)
    end
  |  Lett(id_exp_list, E) =>
    let val id_result_list =
      map(fn (id,exp) => (id,interp1(exp,env))) id_exp_list
    in
    let val new_env = extend_env_all(env, id_result_list)
    in
    interp1(E, new_env)
    end
  end
```

So what can we notice about this interpreter? Given a Literal, it evaluates it. Given a Variable, it looks up the value. Given an addition operation, it invokes the interpreter to produce the values of the operands, then adds them. What about a function definition? Notice that it evaluates to a Closure – this is a way of capturing the environment for the function definition so that we can implement static scope. If we want dynamic scope, we could do away with the closure altogether. The Closure is used in the function application, where we call the interpreter to evaluate the function in the environment we get by evaluating the second expression (the
parameter to the function). Finally, the local block is achieved by using map to evaluate each expression in the block and adding it to the environment.

**Type Inference**

Now we look at the chain of reasoning ML goes through to determine types. Of course, for a given expression, we can let ML tell us the type, but it is useful to see how this is done.

Consider:

```plaintext
fun apply (f, x) = f x;
```

What can we say about types here? We start with a first approximation:

\[ A \times B \rightarrow C \]

This just says that apply takes two arguments and gives back a value. We can observe that the right hand side is syntactically a function, hence

\[ A = E \rightarrow F \]

However, we can now also see from \( x \) that

\[ B = E \]

And from the right hand side that

\[ C = F \]

Putting these together, we reduce to

\[ A \times B \rightarrow C \]

\[ (E \rightarrow F) \times B \rightarrow C \]

\[ (E \rightarrow F) \rightarrow B \rightarrow C \]

\[ (E \rightarrow F) \rightarrow E \rightarrow C \]

\[ (E \rightarrow F) \rightarrow E \rightarrow F \]

That is, the function apply takes a function \( T_1 \rightarrow T_2 \) and a value \( T_1 \), and returns a value \( T_2 \), which is just what ML tells us:

```plaintext
- fun apply (f, x) = f x;
val apply = fn : ('a -> 'b) * 'a -> 'b
```

Here’s another example.

```plaintext
fun foo(nil, l) = l
  | foo(x::xs, l) = x::foo(xs, l);
```

Our first approximation is

\[ A \times B \rightarrow C \]

Further examination gives us:

\[ A = A_1 \text{ list} \] (from \( x::xs \))

\[ C = B_1 \text{ list} \] (from \( x::\text{foo}(...) \))

\[ C = B \] (from first case)

\[ A_1 = B_1 \] (from second case)

Putting these together gives:

\[ A \times B \rightarrow C \]

\[ A_1 \text{ list} \times B \rightarrow B_1 \text{ list} \]

\[ A_1 \text{ list} \times B_1 \text{ list} \rightarrow B_1 \text{ list} \]

\[ A_1 \text{ list} \times A_1 \text{ list} \rightarrow A_1 \text{ list} \]

Which we verify in ML:

```plaintext
- fun foo(nil, l) = l
  = | foo(x::xs,l) = x::foo(xs,l);
val foo = fn : 'a list * 'a list -> 'a list
```
(This is actually how we would code an append function.)

Type inference is an example of a theorem prover, where reasoning is done on what is known to reduce to the least number of unspecified types.

One more example:

```ml
fun foo(f, nil) = nil
    | foo(f, x::y) = (f x) :: foo(f, y);
```

Here we start with

\[ A \times B \rightarrow C \]

And proceed to observe

\[ B = B_1 \text{ list} \] (from args of second clause)
\[ A = D \rightarrow E \] (from body of second clause)
\[ D = B_1 \] (from body of second clause)
\[ C = C_1 \text{ list} \] (from body of second clause)
\[ E = C_1 \] (since \( f \ x \) is first elt of \( C_1 \) list)

And reducing gives

\[ A \times B \rightarrow C \]
\[ (D \rightarrow E) \times B \rightarrow C \]
\[ (D \rightarrow E) \times B_1 \text{ list} \rightarrow C \]
\[ (D \rightarrow E) \times D \text{ list} \rightarrow C \]
\[ (D \rightarrow E) \times D \text{ list} \rightarrow C_1 \text{ list} \]
\[ (D \rightarrow E) \times D \text{ list} \rightarrow E \text{ list} \]

Which we verify with ML:

```ml
- fun foo(f, nil) = nil
  | foo(f, x::y) = (f x) :: foo(f, y);
val foo = fn : ('a -> 'b) * 'a list -> 'b list
```

and this is how we would define a map function on a list that produces a possibly different list type.

Suppose we had the expression

```ml
- foo(fn x:int => true, [true, false]);
Error: operator and operand don't agree
```

Note that this is not well typed since the second argument should be a list of int, not bool.

How about the function

```ml
- fun foo f = (f 1, f true);
Error: operator and operand don't agree
```

Here the problem is we can’t change \( f \) midstream in the body. Once it is seen as one type, it must remain as that type. \( F \) cannot be both a function taking an int and a function taking a bool simultaneously. I.e., the polymorphism becomes monomorphism in the body. However, we can do something along these lines with:

```ml
- let val f = fn x => x
  = in
  = (f 1, f true)
  = end;
val it = (1, true) : int * bool
```
Here we have essentially specified a polymorphic template for \( f \), but \( f \) has not been bound to a specific function, where in the previous example, the call to \( \text{foo} \) would bind \( f \) for the body.

Consider

\[
\begin{align*}
\text{fun } f \text{ g = let } & \text{val } x = \text{g } 1 \\
& \text{val } y = \text{g } \text{true} \\
& \text{in} \\
& \text{x } = \text{y} \\
& \text{end;}
\end{align*}
\]

ML gives us a typecheck error since \( g \) is constrained to type int->'a by \( \text{val } x \) and also to bool->'a by \( \text{val } y \). However, we can try:

\[
\begin{align*}
\text{let } \text{fun } f \text{ x } = \text{x} \\
& \text{in} \\
& \text{f } 1; \text{ f } \text{true} \\
& \text{end;}
\end{align*}
\]

and this will typecheck. The difference is that in the first example, \( g \) gets bound by the first use in \( \text{val } x \) and so is inconsistent with the next expression of \( \text{val } y \). However, in the second example, \( f \) gets bound in each expression separately, so there is no inconsistency.

**Side Effects in ML and calling conventions**

We have seen there is no assignment in ML as we are used to in C. However, in ML we can declare reference types. A reference is like a pointer, actually it has the type of a polymorphic function -  ref: ‘a -> a ref. There is also an operator in ML for de-referencing -  !: ‘a ref -> ‘a, and also a special operator for assignment -  :=  which has type ‘a ref -> ‘a -> int  The following example illustrates these:

\[
\begin{align*}
\text{val } x = \text{ref } 5; \\
\text{val } x = \text{ref } 5 : \text{int ref} \\
\text{! x;} \\
\text{val } it = 5 : \text{int} \\
\text{x } := \text{!x } + \text{ 2;} \\
\text{val } it = () : \text{unit} \\
\text{! x;} \\
\text{val } it = 7 : \text{int} \\
\text{x } := \text{x } + \text{ 1;} \\
\text{Error: operator and operand don't agree}
\end{align*}
\]

Contrast this to imperative languages where we have expressions like \( x = x + 1 \). ML requires that you be explicit about assignment, so the dereferencing on \( x \) is necessary to get a consistent type to add an integer to. Another way of looking at this is that ML distinguishes L values and R values. In C a variable can be an L value or R value, depending on its context in an expression. In ML, only a reference can be an L value, and it cannot be an R value. This means extra work for the programmer, but it also means the compiler can do a better job. The compiler is able to see what is mutable and what is not and use this information to advantage in optimization.

**Side effects and call by name**

Here’s a little example that you probably don’t want to run:
- fun f x = f x;
val f = fn : 'a -> 'b
- fun zero x = 0;
val zero = fn : 'a -> int
- zero (f 5);

Interrupt
This runs forever because (f 5) keeps evaluating to f 5. If this were call by name, the evaluation would be delayed to the body of the function zero, so we wouldn’t loop on evaluation. In call by value, before we get to the body of zero, we must evaluate f 5, but since this keeps resulting in another value to evaluate, it goes on forever.

Consider the code:
val a = ref 10;
fun zero x = (a := !a + x);
fun f x = (x; x; !a);
f ( zero( 5));

Let’s analyze in a call by value language: replace zero(5) by (a := !a +5) and evaluate and pass to f. This evaluation causes a to refer to 15, and even though f evaluates 15 twice, the final value of a returned by the expression is just 15.

If ML were call by name, then the zero(5) would not get evaluated until the evaluation of the expression defining f. That is, it would get evaluated twice, and then the value of a printed out would be 20.

We have already noted that call by name and call by value may differ in efficiency. What we see here is that side effects may cause call by name and call by value to differ in results.

There is also a notion of call by need. What happens here is that the evaluation is only done if we need it, and the value is stored in case we need it again. For pure languages with no side effects, call by need and call by name are equivalent, but in the presence of side effects, they can produce different results.

Finally, we must be careful about the implementation of call by name. If our static scope is implemented by simple substitution, then it is possible for a value to be captured, and in this case it would give the effect of dynamic scope.

Let’s consider free variable capture again. Look at the following example:
let
  fun f x = let
    val y = 10
    in
    x + y
  end
  in
  let
    val y = 20
    in
    f (y + 1)
  end
end

In call by name with substitution, the evaluation is lazy, i.e., delayed. So if we substitute for f in f(y+1), we get essentially: let val y = 20 in let val y = 10 in (y+1) + y end end. And what happens is that y resolves to just the y=10, i.e., the y in ‘y+1’ is captured. So we need to fix this. The fix
is to rename the first \( y \) to \( y_1 \). Then in the expansion we just noted, we would see \((y+1)+y_1\),
which would give us the desired result. (Of course ML is call by value, not call by name with
substitution, so we do get the desired result in real ML.)

Recall in our earlier discussion of L-value versus R-value that ML requires explicit distinction
between the two; nothing is inferred as in C. Let’s consider call by value some more. In a call to
a function where we have call by value, the value of the variable in the caller cannot change – the
function cannot have side effects on the caller’s value (through the formal parameters, anyway).
Local variables containing copies of the caller’s values are used in the function. Contrast to call
by reference, where the function has access to the caller’s variable and so can affect its value. In
this case, no local variables are used, and the function affects the caller’s values directly as
execution proceeds. Although C is call by value, we get the effect of call by reference by passing
addresses of variables and declaring the function interface to expect values which are pointer
types, thus allowing the function access to the caller’s variables.

Now we consider another model: call by value result. In this model, there is a copy-in phase
where the values are copied to local variables for use in the function body, just as in normal call
by value. After execution, there is a copy-out phase, where the values in the local variables are
copied back to the caller’s variables. So it’s somewhere in between call by value and call by
reference. In most cases, it gives the same results as call by reference, but we’ll see where that is
not always true. The idea of call by value result is to encapsulate the function, making its
working opaque to the caller. Side note – we say that call by value is “eager” since it evaluates
arguments even if they are not needed.

Now consider the following C code fragment:

```c
int i = 10;
void foo(int x, int y) { i = y; }
void main() {
    int A[20];
    i = 2; A[i] = 99;
    foo(i, A[i]);
}
```

If we had call by reference, the value of \( i \) would be 99 after execution.
If we had call by value result, the value of \( i \) would be 2 after execution. This happens because the
formal parameter \( x \) passed to \( foo \) is not changed by the body of \( foo \), hence the copy-out phase
will restore \( i \) to the value (2) it had at the beginning of the call. So we see that call by reference
and call by value result can indeed differ. This happens here because of aliasing, that is there are
several names for the same variable – as the global variable \( i \) and as the formal parameter to \( foo \).
The language ADA has call by value result, but it turns out to just be implemented by call by
reference. The claim is that aliasing is incorrect programming, so it doesn’t matter whether call
by value result gives the right behavior! That is, in all “correct” programs, i.e., those with no
aliasing, call by value result and call by reference would produce the same thing, so it’s okay to
implement this way. And you wonder whether the government uses tax dollars effectively….

Here’s an example in Algol:

```algol
var x : int;
   x = 0;
function p(y : int)
begin
    y = 1; x = 0;
```

Atkins – Spring 2007
First note that in Algol, as in most procedural languages, the declaration and initialization are separate. Contrast to functional languages like ML, where the two are inseparable. Consider how the code fragment behaves:

- Call by value: x is 0
- Call by reference: x is 0 (it changes to 1, then back to 0)
- Call by value result: x is 1 (the copy-out sets the formal parameter to its final value – in this case y is the local variable, ends up with 1, and that is copied back to the caller’s parameter, x.)