ML

ML is a statically typed functional language

- Originally, "ML" stood for "meta language"
- Like Scheme, but with types and type inference
- The type system is named ***Hindley-Milner***, it's like the type system we saw with let-based polymorphism

ML Dialects

Two main dialects: Standard ML and OCaml

- Standard ML is the original
- We'll look at the OCaml dialect

Syntax to Implement in ML

$$
\begin{array}{lcl}
\mathbf{M} & = & \lceil \mathbf{n} \rceil \\
 & | & \mathbf{M} - \mathbf{M} \\
 & | & \mathbf{M} \bullet \mathbf{M} \\
 & | & \mathtt{if0}\ \mathbf{M}\ \mathtt{then}\ \mathbf{M}\ \mathtt{else}\ \mathbf{M} \\
 & | & \lambda \mathbf{x} . \mathbf{M} \\
 & | & \mathbf{M}\ \mathbf{M} \\
 & | & \mathbf{x} \\
\mathbf{n} & = & \text{an integer} \\
\mathbf{x} & = & \text{a variable}
\end{array}
$$

Abstract Syntax

```
type xpr = Value of xval
         | Minus of xpr * xpr
         | Times of xpr * xpr
         | Lam of xvar * xpr
         | Var of xvar
         | App of xpr * xpr
         | IfZero of xpr * xpr * xpr
type xval = Num of int
          | Fun of (xval → xval)
```

$\lambda \mathbf{x} . (\mathbf{x} - \lceil 5 \rceil) \overset{\text{parse}}{\Longrightarrow}$ **Lam("x",Minus(Var("x"),Value(Num(5))))**

Step 1

- Plain interpreter with substitution for variables

Step 1

```
let rec eval = function
   Value(v) → v
 | Minus(m1,m2) → let Num(n1) = eval(m1)
                  and Num(n2) = eval(m2)
                   in Num(n1 - n2)
 | Times(m1,m2) → let Num(n1) = eval(m1)
                  and Num(n2) = eval(m2)
                   in Num(n1 * n2)
 | Lam(var,m) → Fun(fun v → eval(replace (var, v) m))
 | App(m1,m2) → let Fun(f) = eval(m1)
                 in f(eval(m2))
 | IfZero(m1,m2,m3) → let Num(n) = eval(m1)
                       in eval(if (n=0)
                                then m2
                                else m3)
```

Step 2

- Use an environment for function bodies instead of replacement

Step 2

```
let rec eval = function
   (Const(v), e) → Num(v)
 | (Minus(m1,m2), e) → let Num(n1) = eval(m1, e)
                       and Num(n2) = eval(m2, e)
                        in Num(n1 - n2)
 | (Times(m1,m2), e) → let Num(n1) = eval(m1, e)
                       and Num(n2) = eval(m2, e)
                        in Num(n1 * n2)
 | (Lam(var,m), e) → Fun(fun v →
                        eval(m, Extend(var,v,e)))
 | (App(m1,m2), e) → let Fun(f) = eval(m1, e)
                      in f(eval(m2, e))
 | (IfZero(m1,m2,m3), e) → let Num(n) = eval(m1, e)
                            in eval((if (n==0)
                                      then m2
                                      else m3),
                                    e)
 | (Var(var), e) → lookup(var, e)
```

Step 3

- Pre-compute variable locations in the environment
- Introduce a "bytecode" compiler for pre-computing

$$\lambda \mathbf{x}.(\lambda \mathbf{y}.(\mathbf{x} \bullet \mathbf{y}))$$

$\overset{\text{compile}}{\Longrightarrow}$

$$\lambda\,.(\lambda\,.(@2 \bullet @1))$$

Step 3

```
let rec comp = function
    (Const(v), e) → CConst(v)
  | (Minus(m1,m2), e) → CMinus(comp(m1, e),comp(m2, e))
  | (Times(m1,m2), e) → CTimes(comp(m1, e),comp(m2, e))
  | (Lam(var,m), e) → CLam(comp(m, CExtend(var,e)))
  | (App(m1,m2), e) → CApp(comp(m1, e),comp(m2, e))
  | (IfZero(m1,m2,m3), e) → CIfZero(comp(m1, e),
                                    comp(m2, e),
                                    comp(m3, e))
  | (Var(var), e) → CVar(offset(var, e))
```

Step 3

```
let rec eval = function
    (CConst(v), e) → Num(v)
  | (CMinus(m1,m2), e) → let Num(n1) = eval(m1, e)
                         and Num(n2) = eval(m2, e)
                          in Num(n1 - n2)
  | (CTimes(m1,m2), e) → let Num(n1) = eval(m1, e)
                         and Num(n2) = eval(m2, e)
                          in Num(n1 * n2)
  | (CLam(m), e) → Fun(fun v → eval(m, Extend(v,e)))
  | (CApp(m1,m2), e) → let Fun(f) = eval(m1, e)
                       in f(eval(m2, e))
  | (CIfZero(m1,m2,m3), e) → let Num(n) = eval(m1, e)
                             in eval((if (n=0)
                                      then m2
                                      else m3),
                                     e)
  | (CVar(n), e) → lookup(n, e)
```

Step 4

- Stop relying on ML functions to implement our functions
- Instead, define a function as an expression-envrionment pair:

```
type xval = Num of int
          | Fun of cxpr * xenv
```

Step 4

```
let rec eval = function
   (CConst(v), e) → Num(v)
 | (CMinus(m1,m2), e) → let Num(n1) = eval(m1, e)
                        and Num(n2) = eval(m2, e)
                         in Num(n1 - n2)
 | (CTimes(m1,m2), e) → let Num(n1) = eval(m1, e)
                        and Num(n2) = eval(m2, e)
                         in Num(n1 * n2)
 | (CLam(m), e) → Fun(m, e)
 | (CApp(m1,m2), e)→ let Fun(fm, fe) = eval(m1, e)
                     in eval(fm, Extend(eval(m2, e), fe))
 | (CIfZero(m1,m2,m3), e) → let Num(n) = eval(m1, e)
                            in eval((if (n=0)
                                        then m2
                                        else m3),
                                    e)
 | (CVar(n), e) → lookup(n, e)
```

Step 5

- Stop relying on ML recursion
- Instead, package work-to-do in a ***continuation***

eval $\lceil 3 \rceil - \lceil 2 \rceil$ **then** *kont*

$\rightarrow$

eval $\lceil 3 \rceil$ **then** $? - \lceil 2 \rceil$ **then** *kont*

$\rightarrow$

eval $\lceil 2 \rceil$ **then** $3 - ?$ **then** *kont*

$\rightarrow$

kont **with** 1

Step 5

```
type kont = Done
          | KSubArg of cxpr * xenv * kont
          | KMultArg of cxpr * xenv * kont
          | KSub of xval * kont
          | KMult of xval * kont
          | KAppArg of cxpr * xenv * kont
          | KApp of xval * kont
          | KIfZero of cxpr * cxpr * xenv * kont
```

Step 5

```
let rec eval = function
   (CConst(v), e, k) → kontinue(Num(v), k)
 | (CMinus(m1,m2), e, k) → eval(m1, e, KSubArg(m2,e,k))
 | (CTimes(m1,m2), e, k) → eval(m1, e, KMultArg(m2,e,k))
 | (CLam(m), e, k) → kontinue(Fun(m,e), k)
 | (CApp(m1,m2), e, k) → eval(m1, e, KAppArg(m2,e,k))
 | (CIfZero(m1,m2,m3), e, k) →
                           eval(m1, e, KIfZero(m2,m3,e,k))
 | (CVar(n), e, k) → kontinue(lookup(n, e), k)
```

Step 5

```
let rec kontinue = function
   (v, KSubArg(m,e,k)) → eval(m, e, KSub(v,k))
 | (v, KMultArg(m,e,k)) → eval(m, e, KMult(v,k))
 | (Num(n2), KSub(Num(n1),k)) → kontinue(Num(n1-n2), k)
 | (Num(n2), KMult(Num(n1),k)) → kontinue(Num(n1*n2), k)
 | (v, KAppArg(m,e,k)) → eval(m, e, KApp(v,k))
 | (v, KApp(Fun(m,e),k)) → eval(m, Extend(v,e), k)
 | (Num(n), KIfZero(m2,m3,e,k)) → eval((if (n=0)
                                          then m2
                                          else m3),
                                       e, k)
 | (v, Done) → v
```

Step 6

- Stop relying on ML's argument passing
- Instead, use a fixed set of registers for arguments

Step 6

```
let rec eval = function unit →
 match (!mReg, !eReg, !kReg) with
   (CConst(v), e, k) → vReg := Num(v); kontinue()
 | (CMinus(m1,m2), e, k) → mReg := m1;
        kReg := KSubArg(m2,e,k); eval()
 | (CTimes(m1,m2), e, k) → mReg := m1;
        kReg := KMultArg(m2,e,k); eval()
 | (CLam(m), e, k) → vReg := Fun(m,e); kontinue()
 | (CApp(m1,m2), e, k) → mReg := m1;
        kReg := KAppArg(m2,e,k); eval()
 | (CIfZero(m1,m2,m3), e, k) → mReg := m1;
        kReg := KIfZero(m2,m3,e,k); eval()
 | (CVar(n), e, k) → vReg := lookup(n, e); kontinue()
```

Step 6

```
let rec kontinue = function unit →
 match (!vReg, !kReg) with
   (v, KSubArg(m,e,k)) → mReg := m; eReg := e;
        kReg := KSub(v, k); eval()
 | (v, KMultArg(m,e,k)) → mReg := m;
        eReg := e; kReg := KMult(v,k); eval()
 | (Num(n2), KSub(Num(n1),k)) → vReg := Num(n1 - n2);
        kReg := k; kontinue()
 | (Num(n2), KMult(Num(n1),k)) → vReg := Num(n1 * n2);
       kReg := k; kontinue()
 | (v, KAppArg(m,e,k)) → mReg := m; eReg := e;
       kReg := KApp(v,k); eval()
 | (v, KApp(Fun(m,e),k)) → mReg := m;
       eReg := Extend(v,e); kReg := k; eval()
 | (Num(n), KIfZero(m2,m3,e,k)) →
       mReg := (if (n=0) then m2 else m3);
       eReg := e; kReg := k; eval()
 | (v, Done) → v
```

Step 7

- Stop using ML's fancy datatypes
- Instead, assume only number and cons cells

Step 7

```
let rec comp = function
   (Const(v), e) → Cons(Int(1), Int(v))
 | (Minus(m1,m2), e) → Cons(Int(2),
                          Cons(comp(m1, e), comp(m2, e)))
 | (Times(m1,m2), e) → Cons(Int(3),
                          Cons(comp(m1, e), comp(m2, e)))
 | (Lam(var,m), e) → Cons(Int(4),
                       comp(m, CExtend(var, e)))
 | (App(m1,m2), e) → Cons(Int(5),
                       Cons(comp(m1, e), comp(m2, e)))
 | (IfZero(m1,m2,m3), e) →
     Cons(Int(6), Cons(comp(m1, e), Cons(comp(m2, e),
                                         comp(m3, e))))
 | (Var(var), e) → Cons(Int(7), Int(offset(var, e)))
```

Step 7

```
let rec eval = function unit →
 let e = !eReg and k = !kReg
 in match (!mReg) with
   Cons(Int(1), v) → vReg := v;
      kontinue()
 | Cons(Int(2), Cons(m1, m2)) → mReg := m1;
      kReg := Cons(Int(1), Cons(m2, Cons(e, k)));
      eval()
 | Cons(Int(3), Cons(m1, m2)) → mReg := m1;
      kReg := Cons(Int(2), Cons(m2, Cons(e, k)));
      eval()
 | ...
```

Step 7

```
let rec kontinue = function unit →
 match (!vReg, !kReg) with
   (v, Cons(Int(1), Cons(m, Cons(e, k)))) →
     mReg := m;
     eReg := e;
     kReg := Cons(Int(3), Cons(v, k));
     eval()
 | (v, Cons(Int(2), Cons(m, Cons(e, k)))) →
     mReg := m;
     eReg := e;
     kReg := Cons(Int(4), Cons(v, k));
     eval()
 | ...
```

Step 8

- Stop using cons cells
- Instead, we have a flat, numerically addressed memory containing only numbers

Step 8

```
let rec
comp = function
    (Const(v), e) → malloc(1, v)
  | (Minus(m1,m2), e) →
      malloc(2, malloc(comp(m1, e), comp(m2, e)))
  | (Times(m1,m2), e) →
      malloc(3, malloc(comp(m1, e), comp(m2, e)))
  | (Lam(var,m), e) →
      malloc(4, comp(m, CExtend(var, e)))
  | ...
```

Step 8

```
let rec eval = function unit →
 let e = !eReg and k = !kReg and p = !mReg
 in match (read p) with
      1 → vReg := read(p+1);
          kontinue()
    | 2 → mReg := read(read(p+1));
          kReg := malloc(1,
                    malloc(read(read(p+1)+1),
                    malloc(e, k)));
          eval()
    | 3 → ...
    | 4 → vReg := malloc(read(p+1), e);
          kontinue()
    | ...
```

Step 8

```
let rec kontinue = function unit →
 let p = !kReg and v = !vReg
  in match (read p) with
     1 → mReg := read(read(p+1));
         eReg := read(read(read(p+1)+1));
         kReg := malloc(3, malloc(v,
                   read(read(read(p+1)+1)+1)));
         eval()
   | 2 → mReg := read(read(p+1));
         eReg := read(read(read(p+1)+1));
         kReg := malloc(4, malloc(v,
                   read(read(read(p+1)+1)+1)));
         eval()
   | ...
```

Step 9

- Implement a garbage collector

(see the code)

Step 10

- Convert *eval* and *kontinue* to assembly

(not provided)