Core ML

- Expressions

$$
\begin{array}{rcl}
M ::= & & x \\
 & | & M_1\, M_2 \\
 & | & \lambda x.M \\
 & | & C(M_1, \cdots, M_n) \\
 & | & \text{match } M \text{ with } | P_1 \text{ when } c_1 \to M_1 \cdots | P_n \text{ when } c_n \to M_n
\end{array}
$$

where the patterns P are built from variables and constructors only.

Core ML

- Programs

$$
\begin{array}{rcl}
\mathit{pgm} ::= & & \epsilon \\
& | & \text{letrec } x = M \ ;; \\
& & \mathit{pgm} \\
& | & M;; \\
& & \mathit{pgm} \\
& | & \text{type } \tau = c_1[\text{ of } \tau_1] \mid \cdots \mid c_n[\text{ of } \tau_n];; \\
& & \mathit{pgm}
\end{array}
$$

Expression and Programs in OCaml

```
type expr
 = Var of string
 | Appl of expr*expr
 | Cons of string * expr list
 | Fun of string * expr
 | Match of expr * (expr*expr*expr) list
;;
type program
 = Empty
 | Type of string * string list *
          (string * type_expr list) list * program
 | LetRec of string * expr * program
 | Expr of expr * program
;;
```

See `expr.ml`

Syntactic Sugar

- $\text{match} \cdots \mid P \to M \cdots \overset{def}{=} \text{match} \cdots \mid P \text{ when true} \to M \cdots$
- $\text{if } b \text{ then } x \text{ else } y \overset{def}{=} \text{match } b \text{ with} \mid \text{true} \to x \mid \text{false} \to y$
- $\text{let } x = M \text{ in } N \overset{def}{=} \text{match } M \text{ with} \mid x \to N$
- $\text{let rec } f_1 = M_1 \text{ and } \cdots \text{ and } f_n = M_n$ can be translated to

$$\begin{array}{l}\text{let rec } h = \text{fun } i \to \text{match } i \text{ with} \mid 1 \to M_1\sigma \cdots \mid n \to M_n\sigma \\ \sigma = [f_1 := h\,1; \cdots ; f_n := h\,n] \\ \text{let } f_1 = h\,1 \\ \vdots \\ \text{let } f_n = h\,n\end{array}$$

Values

- A value is the result of an evaluation.
- Any expression built from constructors is a value.
- A function is also a value.
  But a function has a body which may use defined sybols.
  These defined symbols need to be packaged with the function.

Values as OCaml type

```
type value
 = VCons of string * value list
 | VFun of string * expr * (string,value) map
 | VRec of string*string*expr*(string,value)map
;;
```

where

- VFun is used for simple functions:
  VFun(name of argument, body, environment)
- VRec is used for recursive functions:
  VFun(function name, argument name, body, environment)

Evaluation of Expressions

```
let rec eval_expr env e = match e with
 | Var(x) -> (match get env x with
    | None -> failwith "free␣variable"
    | Some(v) -> v
  )
 | Cons(c,args) ->
    VCons(c,map (eval_expr env) args)
 | Fun(x,e) -> VFun(x,e,env)
```

Evaluation of Expressions

```
let rec eval_expr env e = match e with
 | Appl(e1,e2) ->
   ( let v = eval_expr env e2 in
     match eval_expr env e1 with
       | VCons(_) -> failwith "not␣a␣function"
       | VFun(x,e,env2) ->
          eval_expr (set env2 x v) e
       | VRec(f,x,e,env2) ->
          eval_expr (set (set env2 x v)
                         f (VRec(f,x,e,env2))) e
   )
```

Evaluation of Expressions

```
let rec eval_expr env e = match e with
 | Match(e,cases) ->
     eval_match env (eval_expr env e) cases
and eval_match env v cases = match cases with
 | [] -> failwith "missing␣case"
 | (p,c,e)::cs -> match match_with p v with
     | None -> eval_match env v cs
     | Some(env2) ->
         let env3 = merge env env2 in
         if (eval_expr env3 c)=VCons("true",[])
         then (eval_expr env3 e)
         else (eval_match env v cs)
;;
```

Pattern Matching

```
let rec match_with p v = match p,v with
  | Var(x),_ -> Some(set empty x v)
  | Cons(c,ps),VCons(d,vs) ->
      if c = d then match_list empty ps vs
               else None
  | _ -> None
and match_list m pl vl = match pl,vl with
  | p::ps,v::vs -> (match match_with p v with
      | None -> None
      | Some(m2) -> match_list (merge m m2) ps vs )
  | [],[] -> Some(m)
  | _ -> None
;;
```

Evaluation of Programs

```
let rec eval_pgm env p = match p with
  | Empty -> []
  | Type(_,_,_,pgm) ->
      eval_pgm env pgm
  | LetRec(f,Fun(x,e),pgm) ->
      eval_pgm (set env f (VRec(f,x,e,env))) pgm
  | LetRec(x,e,pgm) ->
      eval_pgm (set env x (eval_expr env e)) pgm
  | Expr(e,pgm) ->
      (eval_expr env e)::(eval_pgm env pgm)
;;
let eval pgm = eval_pgm empty pgm;;
```