

# More ML

Handout #29

CS251 Lecture 19

March 13, 2002

# User-Defined Datatypes I

```
datatype Figure =  
  Circle of real (* radius *)  
| Square of real (* side length *)  
| Rect of real * real (* width x height *)  
  
val pi = 3.14159  
  
fun perim (Circle radius) = 2.0*pi*radius  
  | perim (Square side) = 4.0*side  
  | perim (Rect (width,height)) = 2.0*(width+height)  
  
fun double (Circle r) = (Circle (2.0*r))  
  | double (Square s) = (Square (2.0*s))  
  | double (Rect (w,h)) = (Rect (2.0*w,2.0*h))
```

# User-Defined Datatypes II

Here are the types of the datatypes and functions on the previous slide:

```
datatype Figure = Circle of real | Rect of real * real |  
    Square of real  
val pi = 3.14159 : real  
val perim = fn : Figure -> real  
val double = fn : Figure -> Figure
```

# Standard Library Datatypes

Standard ML of New Jersey (SMLNJ) comes equipped with the following datatypes:

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

```
datatype order = LESS | EQUAL | GREATER
```

Many datatypes provide a `compare` function that returns an order.

E.g.: `Int.compare : (int * int) -> order`

Although lists are “built into” ML, it is possible to define your own list datatype from scratch:

```
datatype 'a lyst = Nil | Cons of 'a * ('a lyst)
```

# Trees

```
datatype `a tree = Leaf
                | Node of (`a tree) * `a * (`a tree)

(* List of elements in in-order traversal *)
(* val inorder: `a tree -> `a list *)
fun inorder Leaf = []
    | inorder (Node(l,v,r)) = (inorder l) @ [v] @ (inorder r)

(* binary search tree insertion *)
fun insert compare x Leaf = Node(Leaf,x,Leaf)
    | insert compare x (nd as (Node(l,v,r))) =
      (case (compare(x,v)) of
         LESS => Node(insert compare x l,v,r)
        | EQUAL => nd (* assume duplicates ignored *)
        | GREATER => Node(l,v,insert compare x r)
         )
```

# Signatures

```
structure Env :
  sig
    val bind : 'a * 'b * ('a -> 'b option)
              -> 'a -> 'b option
    val empty : 'a -> 'b option
    val extend : 'a list * 'b list * ('a -> 'b option)
               -> 'a -> 'b option
    val lookup : 'a * ('a -> 'b) -> 'b
  end
```

# Structures

```
structure Env =  
  struct  
  
    fun empty var = NONE  
  
    fun bind (name, valu, env) =  
      fn var => if var = name then  
                  SOME(valu)  
                  else (env var))  
  
    fun lookup (var, env) = env var  
  
    fun extend (names, vals, env) =  
      Listops.foldr2 bind env names vals  
  
  end
```

Can use functions as `Env.empty`, `Env.bind`, etc.

# Sum-of-Product Datatypes in Scheme

Tagged data = lists with explicit tags = “data-directed programming” (SICP 2.4)

```
(define (double fig)
  (cond ((circle? fig)
        (make-circle (* 2 (circle-radius fig))))
        ((square? fig)
         (make-square (* 2 (square-side fig))))
        ((rect? fig)
         (make-rect (* 2 (rect-width fig))
                    (* 2 (rect-height fig))))
        (else (error "Unknown figure") fig)
        ))
```

```
(define (make-rect w h) (list 'rect w h))
(define (rect-width r) (second r))
(define (rect-height r) (third r))
(define (rect? x) (eq? (first x) 'rect))
```



# Sum-of-Product Datatypes in Other Languages

Java: Make `Circle`, `Square`, `Rect` subclasses of an abstract class `Figure`

C: Use unions (sums) and structs (products)

Haskell: Very similar to ML.

# IBEX Datatypes

```
datatype Primop =
  Add | Sub | Mul | Div | Mod (* Arithmetic ops *)
  | LT | LEQ | EQ | NEQ | GT | GEQ (* Relational ops *)
  | Band | Bor | Not (* Logical ops *)

datatype Value =
  IntVal of int
  | BoolVal of bool
  | StringVal of string

type Var = string

datatype Exp =
  Lit of Value
  | VarRef of Var
  | PrimApp of Primop * Exp list (* rator, rands *)
  | Bind of string * Exp * Exp (* name, defn, body *)
  | If of Exp * Exp * Exp (* test, then, else *)

datatype Program = Prog of Var list * Exp
```

# Sample IBEX Program

```
val avg =  
  Prog(["a", "b"],  
    Bind("c",  
      PrimApp(Add, [VarRef("a"), VarRef("b")]),  
      PrimApp(Div, [VarRef("c"), Lit(IntVal(2))]))))
```

# Evaluation I

```
fun envEval (Lit lit) env = lit
  | envEval (VarRef var) env =
    (case (Env.lookup(var,env)) of
      NONE => raise UnboundVariable(var)
      | SOME(valu) => valu
    )
  | envEval (PrimApp(rator,rands)) env =
    primapply rator
      (map (fn r => envEval r env) rands)
  | envEval (Bind(name,defn,body)) env =
    envEval body
      (Env.bind(name, envEval defn env, env))
  | envEval (If(test,thenExp,elseExp)) env =
    let val testVal = envEval test env
    in case testVal of
      BoolVal(b) =>
        if b then envEval thenExp env
        else envEval elseExp env
      | _ => raise NonBooleanTestError testVal
    end
end
```

# Evaluation II

```
fun envRun (Prog(params,body)) ints =  
  envEval body  
    (Env.extend(params,  
                 map IntVal ints,  
                 Env.empty))
```

# Primitive Application

```
fun primapply Add [IntVal(x), IntVal(y)] = IntVal(x + y)
| primapply Sub [IntVal(x), IntVal(y)] = IntVal(x - y)
| primapply Mul [IntVal(x), IntVal(y)] = IntVal(x * y)
| primapply Div [IntVal(x), IntVal(y)] = IntVal(x div y)
| primapply Mod [IntVal(x), IntVal(y)] = IntVal(x mod y)

| primapply LT [IntVal(x), IntVal(y)] = BoolVal(x < y)
| primapply LEQ [IntVal(x), IntVal(y)] = BoolVal(x <= y)
| primapply EQ [IntVal(x), IntVal(y)] = BoolVal(x = y)
| primapply NEQ [IntVal(x), IntVal(y)] = BoolVal(not(x = y))
| primapply GT [IntVal(x), IntVal(y)] = BoolVal(x > y)
| primapply GEQ [IntVal(x), IntVal(y)] = BoolVal(x >= y)

| primapply Band [BoolVal(x), BoolVal(y)] =
  BoolVal(x andalso y)
| primapply Bor [BoolVal(x), BoolVal(y)] =
  BoolVal(x orelse y)
| primapply Not [BoolVal(x)] = BoolVal(not(x))
| primapply primop rands =
  raise PrimitiveApplicationError(primop, rands)
```