uscheme and the Five Questions

Abstract syntax: imperative core, \texttt{let}, \texttt{lambda}

Values: S-expressions
    (especially \texttt{cons} cells, function closures)

Environments:
A name stands for a mutable location holding value

Evaluation rules: \texttt{lambda} captures environment

Initial basis: yummy higher-order functions
A Scheme program is *just another S-expression*

- Function `define-syntax` manipulates syntax at compile time
- Macros are hygienic—name clashes impossible
- `let`, `and`, many others implemented as macros
Real Scheme: Conditionals

(cond (c1 e1) ; if c1 then e1
     (c2 e2) ; else if c2 then e2
     ...) ; else if c2 then e2
     ... ... ...
     (cn en)) ; else if cn then en

; Syntactic sugar---’if’ is a macro:
(if e1 e2 e3) == (cond (e1 e2)
                      (#t e3))
Real Scheme: Mutation

Not only variables can be mutated.

Mutate heap-allocated cons cell:

\[(\text{set-car!} \; '(a \; b \; c) \; 'd) \Rightarrow (d \; b \; c)\]

Circular lists, sharing, avoids allocation

• still for specialists only
fun length [] = 0
  | length (x::xs) = 1 + length xs

val res = length [1,2,3]
fun map f [] = []
  
  | map f (x::xs) = (f x) :: (map f xs)

val res1 =
  map length [[], [1], [1,2], [1,2,3]]
Map, without redundant parentheses

fun map f [] = []
    | map f (x::xs) = f x :: map f xs

val res1 = map length [[], [1], [1,2], [1,2,3]]
fun filter pred [] = []
    | filter pred (x::xs) = (* pred? not legal *)
        let val rest = filter pred xs
        in if pred x then
            (x::rest)
        else rest
        end

val res2 =
    filter (fn x => (x mod 2) = 0) [1,2,3,4]

(* Note fn x => e is syntax for lambda in SML *)
fun filter pred [] = []
  | filter pred (x::xs) = (* no 'pred?' *)
      let val rest = filter pred xs
      in  if pred x then
           x :: rest
      else
           rest
      end

val res2 =
  filter (fn x => (x mod 2) = 0) [1,2,3,4]
fun exists pred [] = false
| exists pred (x::xs) =
  (pred x) orelse (exists pred xs)

val res3 =
  exists (fn x => (x mod 2) = 1) [1,2,3,4]
fun exists pred [] = false
    | exists pred (x::xs) =
        pred x orelse exists pred xs

val res3 =
    exists (fn x => (x mod 2) = 1) [1,2,3,4]
fun all pred [] = true
| all pred (x::xs) =
  (pred x) andalso (all pred xs)

val res4 = all (fn x => (x >= 0)) [1,2,3,4]
fun all pred [] = true
| all pred (x::xs) =
    pred x andalso all pred xs

val res4 = all (fn x => (x >= 0)) [1,2,3,4]
exception ListTooShort
fun take 0      1  = []
    | take n  []    = raise ListTooShort
    | take n (x::xs) = x::(take (n-1) xs)

val res5 = take 2 [1,2,3,4]
val res6 = take 3 [1]
    handle ListTooShort =>
        (print "List too short!"; [])

(* Note use of exceptions. *)
Take, without redundant parentheses

exception TooShort
fun take 0 _ = [] (* wildcard! *) |
   take n [] = raise TooShort |
   take n (x::xs) = x :: take (n-1) xs

val res5 = take 2 [1,2,3,4]
val res6 = take 3 [1]
          handle TooShort =>
                  (print "List too short!"; [])

(* Note use of exceptions. *)
fun drop 0 1 = 1
| drop n [] = raise ListTooShort
| drop n (x::xs) = (drop (n-1) xs)

val res7 = drop 2 [1,2,3,4]
val res8 = drop 3 [1]
    handle ListTooShort =>
    (print "List too short!"; [])

Drop
fun takewhile p [] = []
  | takewhile p (x::xs) =
      if p x then (x::(takewhile p xs))
      else []

fun even x = (x mod 2 = 0)
val res8 = takewhile even [2,4,5,7]
val res9 = takewhile even [3,4,6,8]
Takewhile, without redundant parentheses

fun takewhile p [] = []
  | takewhile p (x::xs) =
      if p x then x :: takewhile p xs
    else []

fun even x = (x mod 2 = 0)
val res8 = takewhile even [2,4,5,7]
val res9 = takewhile even [3,4,6,8]
fun dropwhile p [] = []
    | dropwhile p (zs as (x::xs)) = 
      if p x then (dropwhile p xs) else zs
val res10 = dropwhile even [2,4,5,7]
val res11 = dropwhile even [3,4,6,8]

(* fancy pattern form: zs as (x::xs) *)
fun dropwhile p [] = [] 
  | dropwhile p (zs as (x::xs)) = 
    if p x then dropwhile p xs else zs
val res10 = dropwhile even [2,4,5,7]
val res11 = dropwhile even [3,4,6,8]

(* fancy pattern form: zs as (x::xs) *)
Folds

fun foldr p zero [] = zero
    | foldr p zero (x::xs) = p (x, (foldr p zero xs))

fun foldl p zero [] = zero
    | foldl p zero (x::xs) = foldl p (p (x, zero)) xs

val res12 = foldr (op +) 0 [1,2,3,4]
val res13 = foldl (op *) 1 [1,2,3,4]

(* Note 'op' to use an infix operator as a value. *)
Folds, without redundant parentheses

fun foldr p zero [] = zero
    | foldr p zero (x::xs) = p (x, foldr p zero xs )

fun foldl p zero [] = zero
    | foldl p zero (x::xs) = foldl p (p (x, zero)) xs

val res12 = foldr (op +) 0 [1,2,3,4]
val res13 = foldl (op * ) 1 [1,2,3,4]

(* Note 'op' to use infix operator as a value *)
ML—The Five Questions

Syntax: definitions, expressions, patterns, types

Values: num/string/bool, record/tuple, algebraic data

Environments: names stand for values (and types)

Evaluation: uScheme + case and pattern matching

Initial Basis: medium size; emphasizes lists

(Question Six: type system—a coming attraction)