

CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout



Plan

- **Announcements**
 - HW6 is due **Friday**
- **Last time**
 - Midterm
- **Today**
 - ML intro

uscheme and the Five Questions

Abstract syntax: imperative core, `let`, `lambda`

Values: S-expressions

(especially `cons` cells, function closures)

Environments:

A name stands for a mutable location holding value

Evaluation rules: `lambda` captures environment

Initial basis: yummy higher-order functions

Common Lisp, Scheme

- **Advantages**

- High-level data structures
- Automatic memory management (garbage collection)
- Programs as data!
- Hygienic macros for extending the language
- Used in AI applications

- **Disadvantages**

- Hard to talk about data
- Hard to detect errors at compile time

- **All about the lambda**

- Major win
- Real implementation cost (heap allocation)

ML Overview

- **Designed for programs, logic, symbolic data**
- **Theme: Precise ways to describe data**
- **ML = uScheme + pattern matching + exceptions + static types**
- **Three new ideas**
 - (1) Pattern matching is big and important. You might really like it.
 - (2) Exceptions are easy
 - (3) Static types get two to three weeks in their own right

uScheme -> ML Rosetta Stone

uScheme	SML
(cons x xs)	x :: xs
' ()	[]
' ()	nil
(lambda (x) e)	fn x => e
(lambda (x y z) e)	fn (x, y, z) => e
&&	andalso orelse
(let* ([x e1]) e2)	let val x = e1 in e2 end
(let* ([x1 e1] [x2 e2] [x3 e3]) e)	let val x1 = e1 val x2 = e2 val x3 = e3 in e end

Example: The length function

• Algebraic laws

```
length []          = 0
length (x::xs)     = 1 + length xs
```

• Code

```
fun length []      = 0
| length (x::xs)   = 1 + length xs
```

• Notice

- No parentheses! (Yay!)
- Function application by juxtaposition
- Infix operators
- Function application has higher precedence than any infix operator
- Compiler checks all the cases

Length

- Algebraic laws

```
length []          = 0
length (x::xs)     = 1 + length xs
```

- Code

```
fun length []      = 0
|   length (x::xs) = 1 + length xs

val res = length [1,2,3]
```

- uScheme code?

Map

- Algebraic laws?

```
map f [] = ??  
map f (x::xs) = ??
```

- Code

```
fun map f [] = []  
  | map f (x::xs) = (f x) :: (map f xs)  
  
val res1 = map length [[], [1], [1,2], [1,2,3]]
```

- uScheme code?

Map, without redundant parentheses

- **Code from previous slide**

```
fun map f [] = []  
  | map f (x::xs) = (f x) :: (map f xs)  
  
val res1 = map length [[], [1], [1,2], [1,2,3]]
```

- **Code 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]]
```

Filter

• Code

```
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]
```

• Questions

- All operators in uScheme were prefix operators. Which operators in ML in this code are infix operators?
- What is the syntax for a lambda expression in ML?

Exists

• Code

```
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]
```

• Questions

- What must the semantics of “orelse” be? Why?
- What parentheses can we take away? Do we really want to?

All

• Code

```
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]
```

• Questions

- What must the semantics of “andalso” be? Why?
- What other style changes were made from the last slide?

```
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]
```

Take

• Code

```
exception ListTooShort
fun take 0      _      = []
  | 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!"; [])
```

• Questions

- Can we use constants in pattern matching? Where do we see that?
- What in the above code is a wildcard?
- Where else could we put a wildcard in the above?

Drop

• Code

```
fun drop 0 l = l
  | drop n [] = raise ListTooShort
  | drop n (x::xs) = drop (n-1) xs

val res7 = drop 2 [1,2,3,4]
val res8 = drop 3 [10, 20, 30]
           handle ListTooShort =>
             (print "List too short!"; [])
```

• Questions

- What is res7 going to be?
- What is res8 going to be?

Folds

• Code

```
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]
```

• Questions

- What are the results for res12 and res13?
- What is the difference between foldr and foldl? Show with parenthesization of res12 and res13 and using (op -).
- Why do you think we need to say (op +)? How different than uScheme?

ML -- The Five Questions

- **Syntax: definitions, expression, patterns, types**
- **Values: num/string/bool, record/tuple, algebraic data types**
- **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)**