



CSCI-344

Programming Language Concepts (Section 3)

Lecture 11

ML Overview

Instructor: Hossein Hojjat

September 26, 2016

Done:

- Scheme
 - Lambda expressions, Closures, Currying, Algebraic Laws, Higher-order functions, Continuation-Passing Style

This session:

- ML: Overview

Lisp (Scheme)

- Lisp is the earliest representative of functional programming languages
- Lisp (and its dialects) pioneered many interesting ideas in programming languages
 - Higher-order functions
 - Recursion
 - Computation with symbols
- First is not always best
- Lisp (Scheme) isn't perfect:

there are things that could be done better!

Compile time vs. run time errors

- Consider the append function

```
(define append (xs1 xs2)
  (if (null? xs1) xs2
    (cons (car xs1) (append (cdr xs1) xs2))))
```

Compile time vs. run time errors

- Consider the append function

```
(define append (xs1 xs2)
  (if (null? xs1) xs2
    (cons (car xs1) (append (cdr xs1) xs2))))
```

- Assume we'd mistaken **car** with **cdr** and wrote append as following

```
(define append (xs1 xs2)
  (if (null? xs1) xs2
    (cons (cdr xs1) (append (car xs1) xs2))))
```

- Scheme detects such errors dynamically (at run-time)
- When Scheme interpreter evaluates a function call to `append` it returns
“error: cdr applied to non-pair”
- This error can be caught statically at an earlier stage

Type

- Type is a classification for a set of values that determines operations that can be done on values of that type

2	integer?
#f	boolean?
(lambda (x) (< x 5))	integer? → boolean?
car	pair? → any

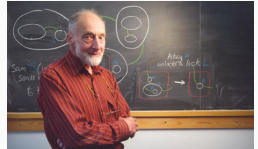
- Type error happens when program is not faithful to the intended semantics of type
 - e.g. applying **car** to a non-pair
- Scheme: run-time (dynamic) type checking
- Dynamic type checks slow down the program

LISP = (**L**ots of (**I**rritating **S**purious) (**P**arentheses)))

– old hacker joke

- Not elegant syntax: it is sometimes unreadable due to lack of syntactic variety
 - Everything is parenthesized prefix notation

“Types are the leaven of computer programming:
they make it digestible.”



Sir Robin Milner
(1934 – 2010)

One of the world's leading
computer scientists

Meta Language

- **ML**: Functional programming language designed by Robin Milner in the 1970s
- It was originally designed as a Meta-Language: language to manipulate Theorems/Proofs

Several dialects:

- Standard ML (SML)
 - Has a formal specification and operational semantics ("The Definition of Standard ML")
- OCaml
 - French dialect with support for objects
 - State-of-the-art: applicable for large-scale software engineering
 - Extensive library, tool, user support
 - F# is a Microsoft .NET language based on OCaml

Companies using OCaml

<http://ocaml.org/learn/companies.html>



Bloomberg



From μ Scheme to ML

μ Scheme

`; this is a comment`

ML

`(* this is a comment *)`

From μ Scheme to ML

μ Scheme

```
; this is a comment  
' ()
```

ML

```
(* this is a comment *)  
nil or []
```

From μ Scheme to ML

μ Scheme

`; this is a comment`

`'()`

`'a`

ML

`(* this is a comment *)`

`nil` or `[]`

`"a"` (ML doesn't support symbols)

From μ Scheme to ML

μ Scheme

```
; this is a comment  
' ()  
' a  
#t / #f
```

ML

```
(* this is a comment *)  
nil or []  
"a" (ML doesn't support symbols)  
true / false
```

From μ Scheme to ML

μ Scheme

```
; this is a comment  
' ()  
' a  
#t / #f  
and / or
```

ML

```
(* this is a comment *)  
nil or []  
"a" (ML doesn't support symbols)  
true / false  
andalso / orelse
```

From μ Scheme to ML

μ Scheme

```
; this is a comment  
' ()  
' a  
#t / #f  
and / or  
(+ 1 2)
```

ML

```
(* this is a comment *)  
nil or []  
"a" (ML doesn't support symbols)  
true / false  
andalso / orelse  
1 + 2
```


From μ Scheme to ML

μ Scheme

```
; this is a comment  
' ()  
' a  
#t / #f  
and / or  
(+ 1 2)  
' (1 2)
```

ML

```
(* this is a comment *)  
nil or []  
"a"      (ML doesn't support symbols)  
true / false  
andalso / orelse  
1 + 2  
[1,2]
```

From μ Scheme to ML

μ Scheme

```
; this is a comment  
' ()  
' a  
#t / #f  
and / or  
(+ 1 2)  
' (1 2)  
(cons 1 '(2 3))
```

ML

```
(* this is a comment *)  
nil or []  
"a" (ML doesn't support symbols)  
true / false  
andalso / orelse  
1 + 2  
[1,2]  
1 :: [2,3]
```

μ Scheme

```
; this is a comment  
' ()  
' a  
#t / #f  
and / or  
(+ 1 2)  
' (1 2)  
(cons 1 '(2 3))  
(val x 1)
```

ML

```
(* this is a comment *)  
nil or []  
"a" (ML doesn't support symbols)  
true / false  
andalso / orelse  
1 + 2  
[1,2]  
1 :: [2,3]  
val x = 1;
```

From μ Scheme to ML

μ Scheme

```
; this is a comment
' ()
'a
#t / #f
and / or
(+ 1 2)
'(1 2)
(cons 1 '(2 3))
(val x 1)
(func 2)
```

ML

```
(* this is a comment *)
nil or []
"a"      (ML doesn't support symbols)
true / false
andalso / orelse
1 + 2
[1,2]
1 :: [2,3]
val x = 1;
func(2)
```

From μ Scheme to ML

μ Scheme

```
; this is a comment
' ()
'a
#t / #f
and / or
(+ 1 2)
'(1 2)
(cons 1 '(2 3))
(val x 1)
(func 2)
(define f(x) (+ x 1))
```

ML

```
(* this is a comment *)
nil or []
"a" (ML doesn't support symbols)
true / false
andalso / orelse
1 + 2
[1,2]
1 :: [2,3]
val x = 1;
func(2)
fun f(x) = x + 1;
```

From μ Scheme to ML

μ Scheme

```
; this is a comment
' ()
'a
#t / #f
and / or
(+ 1 2)
'(1 2)
(cons 1 '(2 3))
(val x 1)
(func 2)
(define f(x) (+ x 1))
(let (x e1) e2)
```

ML

```
(* this is a comment *)
nil or []
"a"      (ML doesn't support symbols)
true / false
andalso / orelse
1 + 2
[1,2]
1 :: [2,3]
val x = 1;
func(2)
fun f(x) = x + 1;
let val x = e1 in e2;
```

ML = μ Scheme + pattern matching + exceptions + static types

Pattern Matching

- We have used **pattern matching** before in operational semantics
 - When applying inference rules
 - Example: We match the pattern $\langle \text{Literal}(3), \emptyset, \emptyset \rangle$ with the pattern on the left hand-side of conclusion:

$$\frac{}{\langle \text{Literal}(v), \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle} \text{(Literal)}$$

- A **pattern** is an expression containing variables for which other expressions may be substituted (informal definition)
- Matching a pattern against an expression:
find suitable substitution that makes pattern identical to expression

Example:

- Consider the De Morgan's Law in Boolean algebra:

$$\neg(\alpha \wedge \beta) = \neg(\alpha) \vee \neg(\beta)$$

- We can use pattern matching to apply De Morgan's law
 $\neg(\neg p \wedge \neg \neg q) = \neg \neg p \vee \neg \neg \neg q$ (p and q are Boolean variables)
- Here we substituted meta-variables α and β with $\neg p$ and $\neg \neg q$

Pattern Matching

- Pattern matching offers a powerful alternative to define functions

μ Scheme

```
(define length (xs)
  (if (null? xs) 0 (+ 1 (length (cdr xs)))))
```

ML

```
fun length xs =
  if xs = nil then 0 else 1 + length(tl(xs));
```

ML (Pattern Matching)

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

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

- More readable syntax (fewer parentheses than μ Scheme)
- Function application by juxtaposition
- Function application has higher precedence than any infix operator
- Compiler (interpreter) checks all the cases

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

- `map length [[1], [], [6,7,8]]`
reduces to
- `[1, 0, 3]`

```
fun filter pred [] = []
  | filter pred (x::xs) =
      let val rest = filter pred xs in
        if pred x
          then (x::rest)
          else rest
      end;
```

- `filter (fn x => (x mod 2) <> 0) [1,2,3,4,5,6,7]` reduces to
- `[1, 3, 5, 7]`
- The convention of using a question mark in the names of predicates doesn't work in SML

```
fun exists pred [] = false
  | exists pred (x::xs) =
    (pred x) orelse (exists pred xs);
```

- `exists (fn x => (x mod 2) <> 0) [2,4,6,8,10]`
reduces to
- `false`

takewhile

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

- `takewhile (fn x => (x <= 5)) [1,3,5,7,9]`
 reduces to
- `[1, 3, 5]`

fold

```
fun foldr f base [] = base
  | foldr f base (x::xs) = f (x, (foldr f base xs));
fun foldl f base [] = base
  | foldl f base (x::xs) = foldl f (f(x,base)) xs;
```

- `foldr (op +) 0 [1,2,3,4,5]`
reduces to 15

- `foldl (op -) 0 [1,2,3,4,5]`
reduces to 3

- Note: `op` converts an infix operator into a function

Exceptions

- ML was one of the earliest languages to include exceptions at the language level
- Exceptions factor out error handling as “exceptional” code from the “normal” code

```
exception Empty;  
fun hd (x::xs) = x  
  | hd [] = raise Empty;
```

- Handling Exceptions:

```
fun inc_head (xs) = hd xs + 1  
handle Empty => (print "error";0);
```

- **Syntax:** expressions, definitions, patterns, types
- **Values:** num/string/bool, record/tuple, algebraic data
- **Environments:** names stand for values (and types)
- **Evaluation:** μ Scheme + case and pattern matching
- **Initial Basis:** medium size; emphasizes lists

(Question Six: type system - will talk about it later)

- ML: general-purpose functional programming language
- Static Type Checking
- Pattern Matching

Next Lecture

- Abstract Data Types