Programming Language Concepts

CMPSC 461

Functional Programming--Scheme

(Variable, Expression, and Function)

Professor: Suman Saha

Why Study Functional Programming ?

- Expose you to a new programming model
  - Functional Programming (FP) is drastically different
    - Scheme: no loops; recursion everywhere
- FP has had a long tradition
  - Lisp, Scheme, ML, Haskell, …
  - The debate between FP and imperative programming
- FP continues to influence modern languages
  - Most modern languages are multi-paradigm languages
  - Delegates in C#: higher-order functions
  - Python: FP; OOP; imperative programming
  - Scala: mixes FP and OOP
  - C++11: added lambda functions
  - Java 8: added lambda functions in 2014
  - Erlang: behind WhatsApp

Who’s using them?

PennState

twitter

map-reduce in their data centers

Google

Scala for
correctness, maintainability, flexibility

facebook

Microsoft Be what’s next.

Erlang for
concurrency,
Haskell for
managing PHP

F# in Visual Studio

JANE STREET

Haskell to
synthesize hardware

bluespec

O’Caml
for reliability

BARCLAYS

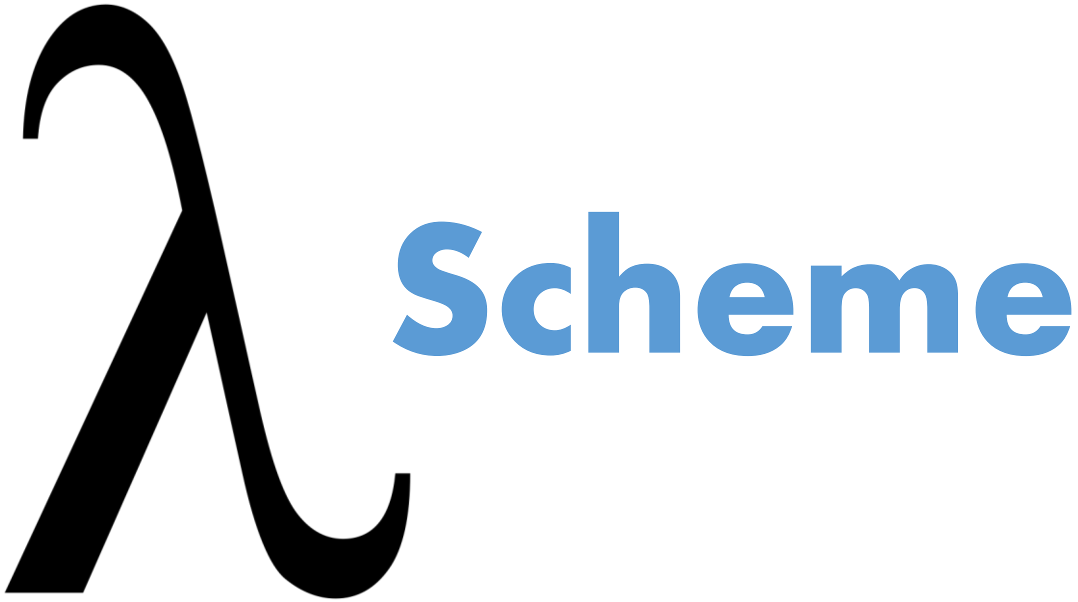
Haskell
for specifying
equity derivatives

mathematicians

Coq proof of
4-color theorem

www.artima.com/scalazine/articles/twitter_on_scala.html
http://gregosuri.com/how-facebook-uses-erlang-for-real-time-chat
http://www.janestcapital.com/technology/ocaml.php
http://msdn.microsoft.com/en-us/fsharp/cc742182
http://labs.google.com/papers/mapreduce.html
http://www.haskell.org/haskellwiki/Haskell_in_industry

A Functional Programming Language



DrRacket

- An interactive, integrated, graphical programming environment for Scheme
- Installation
  - You could install it on your own machines
    - http://racket-lang.org/
- Be sure that the language “Standard (R5RS)” is selected
  - Click Run

Scheme is Simple

- Design for teaching” “A language for describing processes”
- Almost minimally simple syntax
- Only one thing you can do
- Only one data structure

Scheme is Simple:
The one thing you can do

(operator operand1 operand2 …)

Simple - Scheme Expressions

- Prefix notation (Polish notation):
  - *3+4* is written in Scheme as (+ 3 4)
  - Parentheses are necessary
  - Compare to the infix notation: (3 + 4)
- *4+(5 * 7)* is written as
  - (+ 4 (* 5 7))
  - Parentheses are necessary

Simple – Arithmetic

- (+ 3 4)

7

- (* 3 4)

12

- (+ 5 (* 2 2))

?

Your Turn

- In Scheme, “(3+8)+2” is written as

A. (+ (3 + 8) 2)

B. (+ 2 (+ 3 8))

C. (+ (+ 3 8) 2)

D. + (+ 3 8) 2

E. (+ + 3 8 2)

Your Turn

- In Scheme, “3+8/2” is written as

A. (+ (8 / 2) 3)

B. (+ 3 (/ 8 2))

C. (+ (/ 8 2) 3)

D. (+ 3 (/ 2 8))

E. 3 + (/ 8 2)

Scheme Variables

- Variables
  - (define pi 3.14)
  - No need to declare types
- Variables are case insensitive
  - pi is the same as Pi

Simple – Defining values

- (define foo 3)

- foo

3

- (* foo 4)

? 3 4

12

Scheme Expressions

- General syntax: $(\underset{f_1}{E_1}\ \ \underbrace{E_2\ \dots\ E_k})$

Function to invoke

Function arguments

  - Applying the function E1 to arguments E2, …, Ek
  - Examples: $(\underbrace{+\ 3\ 4}_{7})$, $(+\ 4\ (\underbrace{*\ 5\ 7}))$ $4 + 35 = 39$
  - Uniform syntax, easy to parse

$(f_1\ a_1\ a_2\ a_3)$

User-Defined Functions

- Mathematical functions
  - Take some arguments; return some value
- E.g., $f(x) = x^2$
  - $f(3) = 9;\ f(10) = 100$
- Scheme syntax
  - (define (square x) (* x x))
- A two-argument function: $f(x,y) = x + y^2$
  - (define (f x y) (+ x (* y y)))
  - calling the function: (f 3 4)

(+ x (* y y))

a_1 a_2

Simple – Defining function

- (define (square x) (* x x))

- (square 4) x

16

- (+ (square 2) (square 3))

?

(+ 4 9)

= 13

Built-in Functions

0 / 1 / more

- +, *
  - take 0 or more parameters
  - applies operation to all parameters together
  - (+ 2 4 5) = 11
  - (* 3 2 4) = 24
  - zero or one parameter?
    - (+) → 0
    - (*) → 1
    - (+ 5) → 5
    - (* 8) → 1*8 = 8

4+5   3*4

```
def prod()
    f.val = 1
    list = [ ... ]
    for ( )
        f.v = f_v * List[i]
    return f.val
```

```
def sum()
    total = 0
    list = [ . . . . . ]
    for ( )
        total = total + list[i]
    return total
```

5

5

Simple – Flow control

```
➢(define (abs x)
         (if (< x 0)
             (- x)   => x
             x))
```

```
➢(abs -3)
3
```

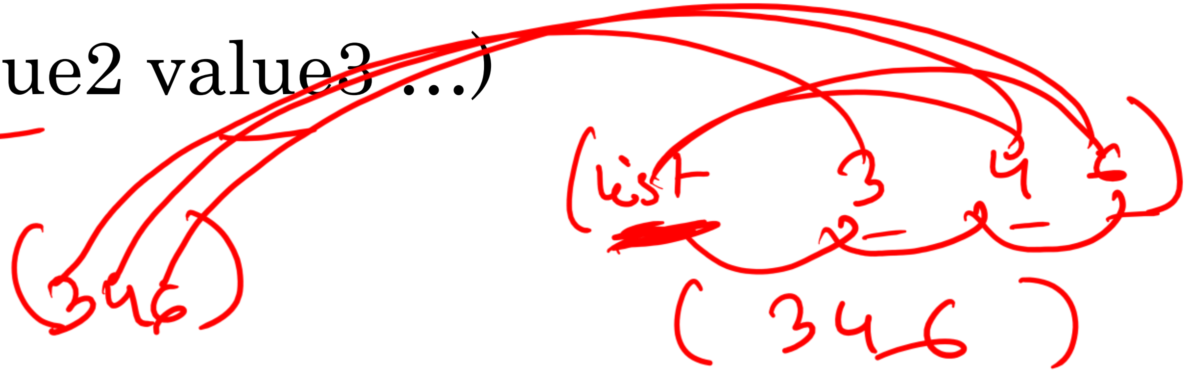
```
➢(abs 3)
3
```

Scheme is Simple:
The one data structure

(value1 value2 value3 …)

; To make one, we write:

(list value1 value2 value3 …)



Simple – Using data

- (sort (list 4 6 5))

(4 5 6)

- (length (list 1 2))

2

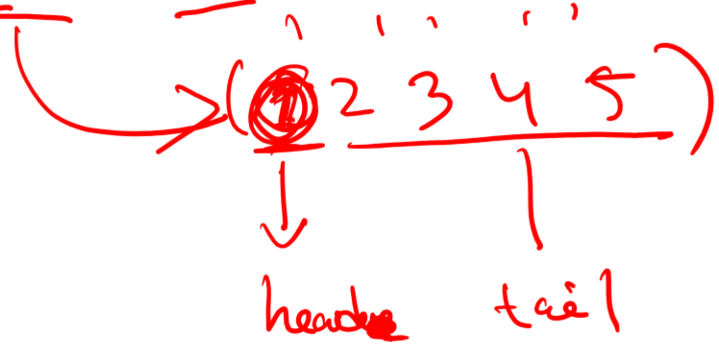
(1 2)

Scheme is Weird

- Functional
- Dynamic typing
- Functions are values

Weird – Functional – list manipulation

- (define my-list (list 1 2 3 4 5))



- my-list

(1 2 3 4 5)

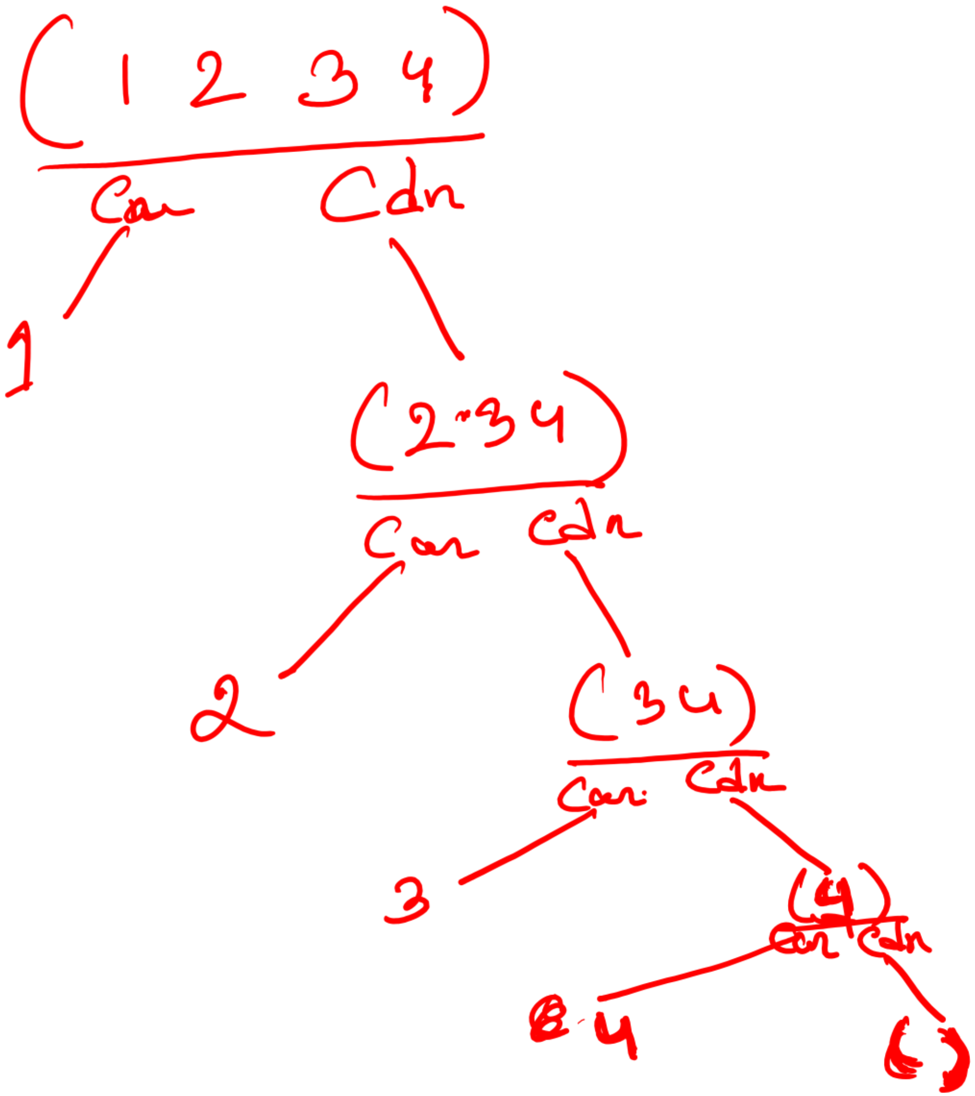
- (car my-list)

1

Car -> ( )

- (cdr my-list)

(2 3 4 5)



Weird – Dynamic typing

- (define (improved-code q) (* q 2)) ⇒ 8

poon 4

- (define code-quality 4)

- (improved-code code-quality)

4

8

- (define code-quality “poor”)

- (improved-code code-quality)

**: expects type as 1st argument, give.....*

(* "poor" 2)

Weird – Functions are values

- (define (double value) (* 2 value))

*2 → fn

- (define (apply-twice fn value) (fn (fn value)))

a_1 a_2

- (apply-twice double 2)

8

(fn (fn value))

(double (double 2))

(double ((* 2 2)))

a_1

(apply-twice (lambda(x) (*x x)) 2)

→ (* 2 (* 2 2)) ⇒ (* 2 4)

⇒ 8

Scheme is Cool

- Generic without all that syntax

➢(sort (list 5 4 3 2 1) <)

(1 2 3 4 5)

➢(sort (list “abc” “a” “ab”) string<?)

(“a” “ab” “abc”)

Anonymous Functions

- Syntax based on Lambda Calculus: $\lambda x.\ x^2$
- Anonymous functions
  - (lambda (x) (* x x))
  - are small function can take any number of arguments, but can only have one expression
  - are often arguments being passed to higher-order function
  - are not bound to an identifier
  - can be used only once: ((lambda (x) (* x x)) 3)
  - Introduce names
    - (define square (lambda (x) (* x x)))
    - Same as (define (square x) (* x x))

Top Hat

Scheme Parenthesis

- Scheme is very strict on parentheses
  - which is reserved for function call (function invocation)
  - (+ 3 4) vs. (+ (3) 4)
  - (lambda (x) x) vs. (lambda (x) (x))
    - the second treats (x) as a function call
  - (lambda (x) (* x x)) vs. (lambda (x) (* (x) x))

(_)

x vs (x)

Defining Recursive Functions

- (define diverge (lambda (x) (diverge (+ x 1))))
  - Call this a diverge function

3

4

+ 3 1 = 4

+ 4 1 = 5

diverge 3

diverge 4

diverge 5