CS 461

Programming Language Concepts

Gary Tan
Computer Science and Engineering
Penn State University

* Some slides are adapted from those by Dr. Danfeng Zhang

1

Supplementary Slides
Chap 11 Functional Languages

2

2

Why Study Functional Programming (FP)?

- Expose you to a new programming model
  - 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

3

3

A Brief History of Functional Programming

- Theoretical foundation: Lambda calculus
  - Alonzo Church (1930s)
  - Computability: Lambda calculus = Turing Machine
  - Church-Turing Thesis
- Lisp (McCarthy, 1950s)
  - Directly based on lambda calculus
  - Mostly used for symbolic computation (e.g., symbolic differentiation)
- Scheme (Steele and Sussman, 1970s)
  - A relatively small language that provides constructs at the core of Lisp
  - Racket is a variant of Scheme
- OCaml; Haskell; F#;…

4

4

Racket

5

5

Learning Functional Programming in Racket

- Follow the lectures
- Chap 11 in the textbook
- Online tutorials (links on the course website)
  - Racket guide: https://docs.racket-lang.org/guide/
    - Especially chapter 2

6

6

DrRacket

- An interactive, integrated, graphical programming environment for Racket
- Installation
  - You could install it on your own machines
    - http://racket-lang.org/
- Interactive environment
  - read-eval-print loop
    - try 3.14159, (* 2 3.14159)
  - Compare to typical Java/C development cycle

7

7

DrRacket: Configuration

- #lang racket
- Select View->Hide Definitions to focus on interpreter today

8

8

Functional Programming in Racket

9

9

Racket Identifiers

- Identifiers
  - (define pi 3.14)
  - No need to declare types
- Identifier names are case sensitive
  - In contrast, Scheme identifiers are case insensitive

10

10

Racket Expressions

- Prefix notation (Polish notation):
  - *3+4* is written in Racket 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
- Exercise: write the following in Racket
  - *(3 + 8) + 2*
  - *3 + 8/2*

11

11

Racket Expressions

- General syntax: ($\underset{\text{Function to invoke}}{E_1}$ $\underbrace{E_2 \;\ldots\; E_k}_{\text{Function arguments}}$)
  - Applying the function E1 to arguments E2, …, Ek
  - Examples: (+ 3 4), (+ 4 (* 5 7))
  - Uniform syntax, easy to parse

12

12

Built-in Functions

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

13

13

User-Defined Functions

- Mathematical functions
  - Take some arguments; return some value
- E.g., $f(x) = x^2$
  - $f(3) = 9; f(10) = 100$
- Racket 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)

14

14

Anonymous Functions

- Syntax based on Lambda Calculus: $\lambda x.\ x^2$
- Anonymous functions
  - (lambda (x) (* x x))
  - 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))

15

15

Racket Parenthesis

- Racket 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))

16

Defining Recursive Functions

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

17

Booleans

- Boolean values
  - #t, #f for true and false
- Predicates: funs that evaluate to true or false
  - convention: names of Racket predicates end in "?"
  - number?: test whether argument is a number
  - equal?
    - ex: (equal? 2 2), (equal? x (* 2 y)), (equal? #t #t)
  - =, >, <, <=, >=
    - = is only for numbers
    - (= #t #t) won't work
  - and, or, not
    - (and (> 7 5) (< 10 20))

18

If expressions

- If expressions
  - (if P E1 E2)
    - eval P to a boolean, if it's true then eval E1, else eval E2
  - examples: max
    - (define (max x y) (if (> x y) x y))
  - It does not evaluate both branches
    - (define (f x) (if (> x 0) 0 (diverge x))
    - what is (f 1)? what is (f -1)

19

Mutual Rec. Functions

- even = true, if n =0
  odd(n-1), otherwise
- odd = false, if n =0
  even(n-1), otherwise

```
(define myeven?
    (lambda (n)
      (if (= n 0) #t (myodd? (- n 1)))))
(define myodd?
    (lambda (n)
      (if (= n 0) #f (myeven? (- n 1)))))
```

20

Multi-Case Conditionals

- (cond [P_1 E_1]
  ...
  [P_n E_n]
  [else E_{n+1}])
  - "If P E_1 E_2" is a syntactic sugar
- examples
  - Problem: Write a function to assign a grade based on the value of a test score. an A for a score of 90 or above, a B for a score of 80-89, a C for a score of 70-79, a D for 60-69, a F otherwise.

```
(define (testscore x)
  (cond [(>= x 90) 'A]
        [(>= x 80) 'B]
        [(>= x 70) 'C]
        [(>= x 60) 'D]
        [else 'F]))
```

Discuss DrRacket "definition" panel

- Debugging support; set up break points
- Strings in Racket: "Hello" is case sensitive

21

Higher-Order Functions

- Functions that
  - take functions as arguments
  - return functions as results
- Example:
  - $g(f,x) = f(f(x))$
  - if $f_1(x) = x + 1$,
    then $g(f_1,x) = f_1(f_1(x)) = f_1(x+1) = (x+1) + 1 = x + 2$
  - if $f_2(x) = x^2$,
    then $g(f_2,x) = f_2(f_2(x)) = f_2(x^2) = (x^2)^2 = x^4$

22

22

Higher-Order Functions in Racket

- The ability to write higher-order functions
- Functions are first-class citizens in Racket
- Examples:

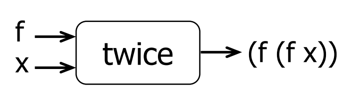
```
(define (twice f x) (f  (f  x)))
(define (plusOne x) (+ 1 x))
(twice plusOne 2)
(twice square 2)
(twice (lambda (x) (+ x 2)) 3)
```

23

23

A Graphical Representation of Twice

- (define (twice f x) (f (f x)))
  - It takes a function f and an argument x, and returns the result of applying f to x twice



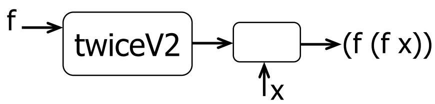
Q: Would Racket accept (twice plusOne)?

24

24

Writing Twice in a Different Way

- (define (twiceV2 f)
  (lambda (x) (f (f x))))



- twiceV2 takes a function f as its argument, and returns a function, which takes x as its argument and returns (f (f x))
- Q: Would Racket accept (twiceV2 plusOne)?

25

25

Let constructs

- (let ([x_1 E_1] [x_2 E_2] … [x_k E_k]) E)
  - Semantics
    - E_1, …, E_k are all evaled; then E is evaled, with x_i representing the value of E_i. The result is the value of E
    - The scope of x_1, …, x_k is E
  - Simultaneous assignment
  - examples
    - (* (+ 3 2) (+ 3 2)) is OK, but repetitive
    - writing (let ([x (+ 3 2)]) (* x x))) is better
    - (+ (square 3) (square 4)) to
      - (let ([three-sq (square 3)] [four-sq (square 4)]) (+ three-sq four-sq))
    - (define x 0)
      (let [(x 2) (y x)] y) to 0

26

Let* constructs

- (let* ([x1 E1] [x2 E2] … [xk Ek]) E)
  - binds x_i to the val of E_i before E_{i+1} is evaled
  - The scope of x_1 is E_2, E_3,… and E_k and E
  - example:
    (define x 0)
    (let ([x 2] [y x]) y) to 0
    (let* ([x 2] [y x]) y) to 2
  - let* is a syntactic sugar
    - (let* ([x 2] [y x]) y)
      = (let ([x 2]) (let ([y x]) y)

      (define x 0)
      (define y 1)
      (let ([x y] [y x]) y) to 0
      (let* ([x y] [y x]) y) to 1

27

Letrec constructs

- (letrec ([x1 E1] [x2 E2] … [xk Ek]) E)
  - The scope of x_1 is E_1, E_2,… and E_k and E
- (letrec
  ([fact (lambda (n)
  (if (= n 0) 1 (* n (fact (- n 1)))))])
  (fact 3))

the let won't work

28