CS 314 Principles of Programming Languages

Lecture 15: Functional Programming

Prof. Zheng Zhang

Rutgers University

October 24, 2018

Class Information

- HW5 posted in Sakai.
- HW6 will be posted by the end of today.
- Next week's recitation is a review for midterm. Attendance is encouraged!
- Reminder: **Midterm exam** 11/7, in class, closed book, closed notes.
  Midterm covers lecture 1-12, hw 1-5, and corresponding book chapters.
- There will be extended office hours next week.
  Pay attention to Sakai announcements!

Scheme - Functions as First Class Values (Higher-Order)

Functions as arguments:

(define f (**lambda** (g x) (g x)))

- (f number? 0) ⇒ (number? 0) ⇒ #t
- (f length '(1 2)) ⇒ (length '(1 2)) ⇒ 2
- (f (lambda (x) (* 2 3)) 3) ⇒ ((lambda (x) (* 2 3)) 3) ⇒ (* 2 3) ⇒ 6

Scheme - Functions as First Class Values (Higher-Order)

Computation, i.e., **function application** is performed by reducing the initial S-expression (program) to an S-expression that represents a value.

Reduction is performed by **substitution**, i.e., replacing formal by actual parameters in the function body.

Examples for S-expressions that directly represent values, i.e., cannot be further reduced:

- function values (e.g.: (**lambda** (x) e))
- constants (e.g.: 3, #t)

Computation completes when S-expression cannot be further reduced

Review: Higher-Order Functions (Cont.)

Functions as returned value:

```
(define plusn
        ( lambda (n)
                (lambda (x) (+ n x))
        )
)
```

Return a function

- **(plusn 5)** evaluates to a function that adds 5 to its argument:

  Question: How would you write down the value of (plusn 5)?

  (lambda (x) (+ 5 x))

- **((plusn 5) 6) = ?**

Review: Higher-Order Functions (Cont.)

In general, any n-ary function

(lambda (x_1 x_2 … x_n) e)

can be rewritten as a nest of *n* unary functions:

(lambda (x_1)

  (lambda (x_2)

    (… **(lambda**(x_n) e) …)))

Review: Higher-Order Functions (Cont.)

In general, any n-ary function

(**lambda** (x_1 x_2 … x_n) e)

can be rewritten as a nest of *n* unary functions:

```
(lambda (x_1)
    (lambda (x_2)
        (… (lambda(x_n) e) … )))
```

This translation process is called currying. It means that having functions with multiple parameters do not add anything to the expressiveness of the language:

((**lambda** (x_1 x_2 ... x_n) e) v_1 v_2 ... v_n)

↓

```
( …  (((lambda (x_1)
          (lambda (x_2)
            ...
             (lambda (x_n) e )...)) v_1) v_2) ... v_n)
```

Review: Higher-order Functions: map

function

```
(define map
  (lambda (f  l)
    (if (null? l)
        '( )
        ( cons (f (car l))  (map f (cdr l)) )
    )
  )
)
```

list

Apply *f* to the first element of *l*

Apply *map and f* to the rest of *l*

- **map** takes two arguments: a function **f** and a list **l**
- **map** builds a new list by applying the function to every element of the (old) list

Review: Higher-Order Functions: map

- Example:
  (map **abs** '(-1 2 -3 4)) ⇒ (1 2 3 4)
  (map (**lambda** (x) (+ 1 x)) '(-1 2 3) ⇒ (0 3 4)

- Actually, the built-in **map** can have more than two arguments:
  (map + '(1 2 3) '(4 5 6)) ⇒ (5 7 9)

Review: More on Higher-Order Functions

reduce Higher order function that takes a binary, associative operation and uses it to “roll up” a list

```
(define reduce
  (lambda ( op l id)
          ( if (null? l)
             id
             (op (car l) (reduce op (cdr l) id)) ) ) )
```

Example: (reduce + '(10 20 30) 0) ⇒

(+ 10 (reduce + '(20 30) 0)) ⇒

(+ 10 (+ 20 (reduce + '(30) 0))) ⇒

(+ 10 (+ 20 (+ 30 (reduce + '() 0)))) ⇒

(+ 10 (+ 20 (+ 30 0))) ⇒

60

Review: Higher-Order Functions

Compose higher order functions to form compact powerful functions

(define sum
        (**lambda** (*f* *l*)
                (reduce + (map *f* *l*) 0)))

(sum (lambda (x) (* 2 x)) '(1 2 3)) ⇒

(reduce (lambda (x y) (+ 1 y)) '(a b c) 0) ⇒

Lexical Scoping and let, let*, and letrec

All are variable binding operations:

LET = let, let*, letrec

```
(LET (   (v1 e1)
         (v2 e2)
         …
         (vn en)  )
         e
)
```

Lexical Scoping and let**,** let***, and** letrec

- **let**:
  - binds variables to values (no specific order), and evaluate body **e** using bindings
  - new bindings are not effective during evaluation of any $\boldsymbol{e_i}$
- **let***:
  - binds variables to values in textual order of write-up
  - new binding $\boldsymbol{e_{i\text{-}1}}$ is effective for next $\boldsymbol{e_i}$.
- **letrec**:
  - bindings of variables to values in no specific order
  - independent **evaluations of all** $\boldsymbol{e_i}$ to **values** have to be possible
  - new bindings effective for all $\boldsymbol{e_i}$
  - mainly used for recursive function definitions

let **and** let* **example**

(**let** ((a 5) (b 6))
        (+ a b)   )

let **and** let* **example**

```
(  let (  (a 5)  (b 6)  )
        (  + a b  )    )          ;; ⇒ 11
```

let **and** let* **example**

```
( let ( (a 5) (b 6) )
      ( + a b )  )          ;; ⇒ 11
```

```
(let ((a 5) (b (+ a 6)) )
   ( + a b )        )
```

let **and** let* **example**

```
( let ( (a 5) (b 6) )
      ( + a b )   )        ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
    ( + a b )       )
```

let **and** let* **example**

```
( let (  (a 5)  (b 6)  )
        (  + a b  )    )          ;; ⇒ 11
```

```
(let ( (a 5) (b (+ a 6)) )
    (  + a b  )          )          ;; ⇒ ERROR: a: undefined
```

let **and** let* **example**

```
( let ( (a 5) (b 6) )
      ( + a b )   )         ;; ⇒ 11


(let ( (a 5) (b (+ a 6)) )
    ( + a b )       )       ;; ⇒ ERROR: a: undefined


(let* ( (a 5) (b (+ a 6)) )
      ( + a b )     )
```

let **and** let* **example**

(**let** ((a 5) (b 6))
(+ a b)) ;; ⇒ 11

(**let** ((a 5) (b (+ a 6)))
(+ a b)) ;; ⇒ ERROR: a: undefined

(**let*** ((a 5) (b (+ a 6)))
(+ a b)) ;; ⇒ 16

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec (  (a 5)
           (b ( lambda() (+ a 6) ) )  )
           (  + a (b)  )      )         ;; ⇒ 16
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec (  (a 5)
           (b ( lambda () (+ a 6) ) )  )
           (  + a (b)  )       )          ;; ⇒ 16

(letrec (  (b (lambda () (+ a 6) ) )
           (a 5)                         )
           (  + a (b) )        )          ;; ⇒ 16
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec ( (even? (lambda (x)
                    ( or (= x 0) (odd? (- x 1)) ) ) )
          (odd?  (lambda (x)
                    ( and (not (= x 0)) (even? (- x 1)) ))) )

          ( list (even? 3) (even? 20) (odd? 21) ) )     ;; ⇒ (#f #t #t)
```

letrec **examples**

Typically used for local definitions for **recursive** functions

```
(letrec (  (even? (lambda (x)
                       ( or (= x 0) (odd? (- x 1)) ) ) )
           (odd?  (lambda (x)
                       ( and (not (= x 0)) (even? (- x 1)) ))) )
           (  list (even? 3) (even? 20) (odd? 21) ) )      ;;  ⇒ (#f #t #t)
```

Lexical (Static) Scoping

- The occurrence of a variable matches the lexically closed **binding** occurrence.
- An occurrence of a variable without a matching binding occurrence is called *free*.
- A variable can occur free and bound in an expression.

We only substitute free occurrences of the formal arguments in the function body when making a function application!

Free and Bound

Consider the following function definition:

```
(define plusn
        ( lambda (n)
                ((lambda (x) (+ n x)))
        )
)
```

n is free

Free and Bound

Consider the following function definition:

```
(define plusn
        ( lambda (n)
                ( lambda (x) (+ n x) )
        )
)
```

n is bound

Environment and Closure

Environment:

- Record the bindings for the variables. If we need the value that a variable denotes, we just look it up in the environment.

An environment is a finite map from variables to values

$$\rho \in \text{Env} = \text{Variables} \rightarrow \text{Values}$$

Closures

Pair the environment with a function (lambda abstraction). The environment must contain values for all free variables of the function. The function can only be evaluated in its attached environment.

Such a pairing is called a **Closure**.

A **closure** is a pair consisting of an environment and a function definition.

$$cl \in \text{Closure} = \{\langle \lambda, \rho \rangle \mid \text{FreeVar}(\lambda) \subseteq \text{DOM}(\rho)\}$$

Closures can be used to implement lexical scoping.
They represent lexically scoped **function values**.

How to Apply a Closure?

How to apply a closure value to actual argument values?

- Let c_v be the closure value < (lambda (x) e), ρ >
- Apply c_v to a value a_v as follows:

Evaluate the body *e* of the function in the environment ρ of the closure extended by the mapping of the **formal parameter x** to the **actual parameter** $\mathbf{a_v}$ ($\rho[x \longrightarrow a_v]$).

How to Apply a Closure: Examples

((lambda(**x**)
    ((lambda(z) ((lambda(x)(z x)) 3)) (lambda(y)(+ x y)))) **1**)

closure interpreter	
	{ }
((lambda(**z**) ((lambda(x)(z x)) 3)) **(lambda(y)(+ x y))**)	{ x $\rightarrow$ 1 }
((lambda(**x**) (z x)) **3**)	{ x $\rightarrow$ 1, z$\rightarrow\langle$ (lambda(y)(+ x y)),{x$\rightarrow$1}$\rangle$ }
(z x)	{ x$\rightarrow$3, z$\rightarrow\langle$(lambda(y)(+ x y)),{x$\rightarrow$1}$\rangle$ }
(+ x y)	{ x $\rightarrow$ 1, y $\rightarrow$ 3}
4	

Next Lecture

Reading:

- Scott, Chapter 11.1 - 11.3
- Scott, Chapter 11.7