CS 314 Principles of Programming Languages

Lecture 17: Functional Programming

Zheng (Eddy) Zhang
Rutgers University
April 4, 2018

Class Information

- Homework 6 will be posted later today.
- All test cases for project 1 have been released.
- If a *small* issue in project 1 results in your code not working for most test cases, you can submit a patch (using the linux utility tool patch to generate the difference of two versions of a file) **by the end of this week.** We will review the change before we determine if you will get partial credit back.

Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: select first element of the list
  (contents of address part of register)
- **cdr**: select rest of the list
  (contents of decrement part of register)
- **(cons element list)**: constructs lists by adding **element** to the front of **list**
- **quote or ':** produces constants

Do not evaluate the ' the content after '. Treat them as list of literals.

Quotes Inhibit Evaluation

> **> (cons** 'a (**cons** 'b '(c d)))
> (a b c d)

;; Now if we quote the second argument

> (cons 'a '(cons 'b '(c d)))

(a cons 'b '(c d))

;; If we unquote the first argument

> (cons a (**cons** 'b '(c d)))

a: undefined;

 cannot reference undefined identifier

   context …

Review: Defining Scheme Functions

(define <fcn-name> **(lambda** (<*fcn-params*>) <expression> **))**

Example: Given function **pair?** (true for non-empty lists, false o/w) and function **not** (boolean negation):

Evaluating (**atom? '(a)**):

1. Obtain function value for **atom?**
2. Evaluate **'(a)** obtaining **(a)**
3. Evaluate **(not (pair? object))**
   a) Obtain function value for **not**
   b) Evaluate **(pair? object)**
      i. Obtain function value for **pair?**
      ii. Evaluate object obtaining **(a)**
      iii. Evaluates to #t
   c) Evaluates to #f
4. Evaluates to #f

```
(define   atom?
          ( lambda (object)
            (not (pair? object) )
          )
 )
```

Review: Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>
2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

```
(define abs-val
        ( lambda (x)
            ( if ( >= x 0) x (- x) )
        )
)

(define rest-if-first
        ( lambda (e l)
                ( if ( eq? e (car l) ) ( cdr l ) '() )
        )
)
```

Review: Conditional Execution *cond*

(cond (<condition1> <result1>)
    (<condition2> <result2>)
    ...
    (<conditionN> <resultN>)
    (else <else-result>)**)** *; optional else clause*

1. Evaluate conditions in order until obtaining one that returns a #t value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>

Conditional Execution: cond

```
(define rest-if-first
  (lambda (e l)
          (cond ( (null? l)          '( )  )
                ( (eq? e (car l) ) ( cdr l) )
                ( else               '( ) )
          )
  )
)
```

Example 1

Example 2

```
(define abs-val
  (lambda ( x )
          ( cond ( (>= x 0) x )
                 (else (- x)  )
          )
  )
)
```

Recursive Scheme Functions: abs-List

```
(define abs-list
  (lambda (l)
          (if (null? l)  '()
                  (cons (abs (car l)) (abs-list (cdr l)))
          )
  )
)
```

- (abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
- (abs-list '()) ⇒ ()

Recursive Scheme Functions: Append

- (append '(1 2) '(3 4 5)) ⇒ (1 2 3 4 5)
- (append '(1 2) '(3 (4) 5)) ⇒ (1 2 3 (4) 5)
- (append '() '(1 4 5)) ⇒ (1 4 5)
- (append '(1 4 5) '()) ⇒ (1 4 5)
- (append '() '()) ⇒ ()

```
(define append
 (lambda (x y)
        (cond ( (null? x) y )
              ( (null? y) x )
              ( else (cons (car x) (append (cdr x) y) ))
        )
 )
)
```

Equality Checking

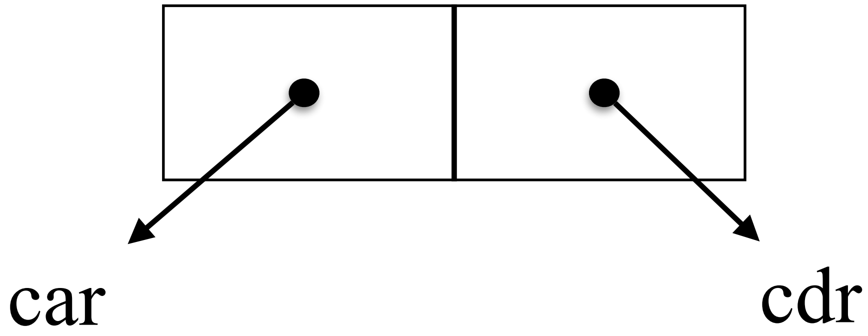
The eq? predicate does not work for lists.

Why not?

- (**cons** 'a '()) produces a new list
- (**cons** 'a '()) produces another new list
- **eq?** checks whether two arguments are the *same*
- (**eq?** (**cons** 'a '()) (**cons** 'a '())) evaluates to #f

Equality Checking

Lists are stored as pointers to the first element (car) and the rest of the list (cdr). This "elementary" data structure, the building block of a list, is called a *pair*



Symbols are stored uniquely, so **eq?** works on them.

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
            (or ( and (atom? x) (atom? y) (eq? x y) )
                ( and (not (atom? x)) (not (atom? y))
                      (equal? (car x) (car y))
                      (equal? (cdr x) (cdr y))
                )
            )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
        (or ( and (atom? x) (atom? y) (eq? x y))
            ( and (not (atom? x)) (not (atom? y))
                  (equal? (car x) (car y))
                  (equal? (cdr x) (cdr y))
            )
        )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
         (or ( and (atom? x) (atom? y) (eq? x y) )
             ( and (not (atom? x)) (not (atom? y))
                 (equal? (car x) (car y))
                 (equal? (cdr x) (cdr y))
             )
         )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
            (or ( and (atom? x) (atom? y) (eq? x y) )
                ( and (not (atom? x)) (not (atom? y))
                      (equal? (car x) (car y))
                      (equal? (cdr x) (cdr y))
                )
            )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
        (or ( and (atom? x) (atom? y) (eq? x y) )
            ( and (not (atom? x)) (not (atom? y))
                  (equal? (car x) (car y))
                  (equal? (cdr x) (cdr y))
            )
        )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
        (or ( and (atom? x) (atom? y) (eq? x y) )
            ( and (not (atom? x)) (not (atom? y))
                (equal? (car x) (car y))
                (equal? (cdr x) (cdr y))
            )
        )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
          (or ( and (atom? x) (atom? y) (eq? x y) )
              ( and (not (atom? x)) (not (atom? y))
                    (equal? (car x) (car y))
                    (equal? (cdr x) (cdr y))
              )
          )
  )
)
```

Equality Checking for Lists

For lists, need a comparison function to check for the **same structure** in two lists.

```
(define equal?
  (lambda (x y)
               (or ( and (atom? x) (atom? y) (eq? x y) )
                   ( and (not (atom? x)) (not (atom? y))
                          (equal? (car x) (car y))
                          (equal? (cdr x) (cdr y))
                   )
               )
  )
)
```

- (equal? 'a 'a) evaluates to #t
- (equal? 'a 'b) evaluates to #f
- (equal? '(a) '(a)) evaluates to #t
- (equal? '((a)) '(a)) evaluates to #f

Scheme: Functions as First Class Values

Functions as arguments:

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

- (f number? 0)
- (f length '(1 2))
- (f (lambda (x) (* 2 3)) 3)

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.

Higher-Order Functions (Cont.)

Functions as returned value:

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

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

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

- **((plusn 5) 6)**

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) …)))

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 <u>currying</u>. 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)
```

Higher-order Functions: map

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

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

Higher-order Functions: map

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

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

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)

Next Lecture

Things to do:

- Read Scott, Chapter 9.1 - 9.3 (4th Edition) or Chapter 8.1 - 8.3 (3rd Edition), Chapter 11.1 - 11.3 (4th Edition) or Chapter 10.1 - 10.3 (3rd Edition)