CS 314 Lecture 9

Functional programming

February 21, 2019

Special (Primitive) Functions

car and cdr can break up any list:

- (car (cdr (cdr '((a) b (c d))))) =
- (caddr '((a) b (c d)))

cons can construct any list:

- (cons 'a '()) =
- (cons 'd '(e)) =
- (cons '(a b) '(c d)) =
- (cons '(a b c) '((a) b)) =

Defining functions

```
(define <name> (lambda (<params>) <expression>))
```

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

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

Conditional Execution: if

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

- Evaluate condition
- If the result is a "true value" (i.e., anything but `#f`), then evaluate and return result1
- 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) '()))))
```

Conditional Execution: cond

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

- Evaluate conditions in order until obtaining one that returns a true value
- Evaluate and return the corresponding result
- If none of the conditions returns a true value, evaluate and return else-result

Conditional Execution: cond

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

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

Recursive Scheme Functions: Length

- (length '()) $\Rightarrow$ 0
- (length '(a b c)) $\Rightarrow$ 3
- (length '(a (d e f) c)) $\Rightarrow$ 3

```
(define length
  (lambda (l) ...
)
```

Recursive Functions: Abs-List

- (abs-list '(1 -2 -3 4 0)) $\Rightarrow$ (1 2 3 4 0)
- (abs-list '()) $\Rightarrow$ ()

```
(define abs-list
  (lambda (l) ...
)
```

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) ...
)
```

Equality Checking

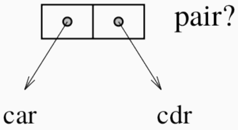
The eq? predicate doesn't work for lists.

Why not?

- (cons 'a '()) produces a new list
- (cons 'a '()) produces another new list
- eq? checks if its 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 lists, 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

```scheme
(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 Values (Higher-order)

Functions as arguments: (define f (lambda (g x) (g x)))

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

Scheme: Functions as Values (Higher-order)

REMINDER: 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 arguments 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`)

Higher-order Functions (Cont.)

Functions as returned values:

```
(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) $\Rightarrow$ 11

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 (x1)
  (lambda (x2)
    ( ... (lambda (xn)
             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.

Higher-order Functions (Cont.)

Question: How to write an application of the original vs. the curried version?

```
((lambda (x1 x2 ... xn) e) v1 v2 ... vn)

(lambda (x1)
  (lambda (x2)
    ( ...
      (lambda (xn) e )...))) v1 v2 ... vn
```

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)) $\Rightarrow$ (1 2 3 4)
- (map (lambda (x) (+ 1 x)) '(-1 2 -3)) $\Rightarrow$ (0 3 -2)

Actually, the built-in map can take more than two arguments:

- (map + '(1 2 3) '(4 5 6)) $\Rightarrow$ (5 7 9)

More on Higher Order Functions

reduce: a 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)) )))
```

More on Higher Order Functions

Example:

- (reduce + '(10 20 30) 0)
- $\Rightarrow$ (+ 10 (reduce + '(20 30) 0))
- $\Rightarrow$ (+ 10 (+ 20 (reduce + '(30) 0)))
- $\Rightarrow$ (+ 10 (+ 20 (+ 30 (reduce + '() 0))))
- $\Rightarrow$ (+ 10 (+ 20 (+ 30 0)))
- $\Rightarrow$ 60

More on Higher Order Functions

Now we can compose higher order functions to form compact powerful functions.

Examples:

```
(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)