

Contents

CSC324H Principle of Programming Languages (Week 5)

Yilan Gu

yilan@cdf.toronto.edu

<http://www.cs.toronto.edu/~yilan/324f09>

- Scheme
 - ✓ Basic types
 - ✓ Basic functions
 - ✓ Different recursions
 - ✓ Efficiency and tail recursion
 - Let, let*
 - **letrec**
 - **More examples of unbounded lambda (with parameter list)**
 - **higher-order functions**
- **Theory – lambda calculus**

1

2

letrec

- `(letrec ((var1 expr1) ... (varn exprn)) body)`
- Scope: Each binding of a variable has the entire letrec expression as its region, variables are visible within expressions as well
 - recursion
- Evaluation: `expr1, ..., exprn` are evaluated in an undefined order, saved, and then assigned to `var1, ..., varn`, with the appearance of being evaluated in parallel.
- Let and let* can't be used for binding with recursion procedure

3

Recall: Tail Recursion for rev

```
(define (rev lst) (rev-acc lst ()))  
(define rev-acc  
  (lambda (lst acc)  
    (if (null? lst) acc  
        (rev-acc (cdr lst) (cons (car lst) acc))))))
```

4

Letrec Examples: rev

```
(define reverse
  (lambda (lst)
    (letrec ((rev-acc
              (lambda (lst acc)
                (if (null? lst) acc
                    (rev-acc (cdr lst)
                              (cons (car lst) acc))))))
      (rev-acc lst '()))))
```

5

Letrec Examples: length

```
(define (length lst)
  (letrec ((length-tail
            (lambda (lst len)
              (if (null? lst) len
                  (length-tail (cdr lst) (+ len 1))))))
    (length-tail lst 0)))
```

6

Unbounded Lambda with Parameter Lists

Review: Unbounded Lambda

```
(define sum
  (lambda arg
    (cond ((null? arg)
           (display "expecting at least one argument"))
          ((null? (cdr arg)) (car arg))
          (else (+ (car arg) (apply sum (cdr arg)))))))
```

OR

```
(define (sum . arg)
  (cond ((null? arg)
         (display "expecting at least one argument"))
        ((null? (cdr arg)) (car arg))
        (else (+ (car arg) (apply sum (cdr arg))))))
```

7

8

More Examples of Parameter Lists

- ```
(define list-args
 (lambda (arg
 arg))
```

```
> (list-args)
()
>(list-args 'a)
(a)
>(list-args 'a 'b 'c '(d e))
(a b c (d e))
```

9

## Higher-Order Functions

11

## More Examples of Parameter Lists

- ```
(define sum-non1-args  
  (lambda (first . rest)  
    (apply + rest)))
```

```
> (sum-non1-args 1 2 4)  
6  
>(sum-non1-args 1)  
0  
>(sum-non1-args )  
error
```

10

Functional Programming

- One of the most important common aspects:
 - Functions as values:
- Arguments or return values of "higher-order functions".
- A **higher-order function** is a function that takes a function as a parameter, returns a function or does both.
 - An example in math – **function composition**
- for any two unary functions $f(x)$, $g(x)$, we define a higher-ordered function $h(f, g, x) = f(g(x))$

12

Procedures as Input Values

```
(define (all-num? list)
  (or (null? list)
      (and (number? (car list))
            (all-num? (cdr list)))))

(define (abs-list list)
  (cond ((null? list) '())
        (else (cons (abs (car list)) (abs-list (cdr list))))))

(define (all-num-f fun list)
  (if (all-num? list) (fun list)
      'error))
```

13

Procedures as Returned Values

```
(define add-mult
  (lambda (x)
    (cond
      ((and (number? x) (> x 0))
       (lambda (y) (+ x y)))
      ((and (number? x) (< x 0))
       (lambda (y) (* x y)))
      (else (lambda (x) x)))))
```

14

Define a Limited map

- A limited version (mymap op list)
- Example
> (mymap abs '(-1 0 9))
 (1 0 9)
> (mymap (lambda (x) (+ 1 x)) '(1 2 3))
 (2 3 4)

```
(define (mymap f l)
  (cond ((null? l) '())
        (else (cons (f (car l))
                      (mymap f (cdr l))))))
```

- The build-in map is more general and more powerful

15

Map

- (map proc list1... listn)
- All lists must be the same length
- returns: A list of applying proc to all i^{th} ($i=1..n$) elements of each list
- Examples:
(map abs '(-4 5 0 -1)) => (4 5 0 1)
(map * '(1 3 5) '(6 8 3)) => (6 24 15)
(map * '(1 3 5) '(6 8 3) '(0 1 2)) => (0 24 30)

16

What's Wrong Here?

```
(define (atomcount s)
  (cond ((null? s) 0)
        ((not (pair? s)) 1)
        (else (+ (map atomcount s))))
))
```

```
> (atomcount '(a b))
error
```

- Why? What's wrong?
 - Apply
 - Eval

17

Fix atomcount using eval

```
(define (atomcount s)
  (cond ((null? s) 0)
        ((not (pair? s)) 1)
        (else (eval
                 (cons '+ (map atomcount s)))))
))
```

```
> (atomcount '(a b))
2
```

18

Eval

- The eval function takes a “quoted” expression or definition and evaluates it:

```
> (eval '(+ 1 2))
3
```

- The power of eval is that an expression can be constructed dynamically:

```
(define (eval-formula formula)
  (eval `(let ([x 2] [y 3]) ,formula)))
> (eval-formula '(+ x y))
5
> (eval-formula '(+ (* x y) y))
9
```

19

Eval (examples)

- Be careful when using eval

```
> (eval '(+ 1 2))
3
```

```
> (eval '(append (a) (b)))
Error
```

```
> (eval '(append '(a) '(b)))
(a b)
```

- Too complicated, another solution: apply

20

Apply (revised)

- `(apply proc obj1 ... objm list1... listn)`
- returns: the result of applying `proc` to `obj ...` and the elements of `list`
- `apply` is useful when some or all of the arguments to be passed to a procedure are in a list, since it frees the programmer from explicitly destructuring the list.
- Examples:
`(apply + '(4 5)) => 9`
`(apply min 5 1 3 '(6 8 3 2 5)) => 1`

21

Fold Right

- `(foldr op id lst1 lst2 ... lstn)`
 - `op` : an binary procedure
 - `lst` : list of arguments (can be more than one lists if `op` can be applied to multiple arguments, all lists are same length)
 - apply `op` right-associatively to elements of `lst` (or all the lists), and return result of evaluation
 - the identity element `id` is always used
- That is: (one list as an example)
`(fold-right op id '()) => id`
`(fold-right op id '(e)) => (op e id)`
`(fold-right op id '(e1 e2 ... en)) => (op e1 (op e2 (op ... (op en id))))`
- Other dialect:
 - In MIT Scheme: `(fold-right op id lst1 ... lstn)`

23

Fix atomcount using apply

```
(define (atomcount s)
  (cond ((null? s) 0)
        ((not (pair? s)) 1)
        (else (apply + (map atomcount s))))
))

> (atomcount '(a b))
2
```

22

Fold Right (examples)

```
> (foldr cons 'a '(1 2 3))
(1 2 3 . a)

(foldr cons 'a '(1 2 3))
(cons '1 (foldr cons 'a '(2 3)))
(cons '1 (cons '2 (foldr cons 'a '(3))))
(cons '1 (cons '2 (cons '3 (foldr cons 'a '()))))
(cons '1 (cons '2 (cons '3 'a)))
....
(1 2 3 . a)
```

24

Fold Right (more examples)

```
> (foldr list 'a '(1 2 3))
(1 (2 (3 a)))
> (foldr cons 'a '(1 2 3) '(4 5 6))
.. foldr: arity mismatch, does not accept 2 arguments
   #<procedure:cons>
> (foldr list 'a '(1 2 3) '(4 5 (k)))
(1 4 (2 5 (3 (k) a)))
> (foldr list 'a '())
a
> (foldr cons 'a '())
a
```

25

Define a Limited Fold Right

```
(define (myfoldr op id list)
  (if (null? list) id
      (op (car list)
          (myfoldr op id (cdr list))))))
```

26

Fold Left (two different versions)

- PLT scheme version:
(foldl **op** **id** **lst1** ...**lstn**)
 - op : an binary procedure
 - lst : list of arguments (accepts multiple lists)
 - apply op left-associatively to elements of lst, and return result of evaluation
 - the identity element **id** is always used
- That is: (one list as an example)
(foldl op id '()) => **id**
(foldl op id '(e)) => (op **e** **id**)
(foldl op id '(e1 e2 ... en)) => (op **en** ... (op **e2** (op **e1** **id**)))

27

foldl (PLT version)

```
> (foldl list 'a '(1 2 3))
(3 2 1 a)
>(foldl cons 'a '(1 2 3))
(3 2 1 . a)
> (foldl list 'a '(1 2 3) '(4 5 6))
(3 6 (2 5 (1 4 a)))
> (foldl list 'a '())
a
> (foldl cons 'a '())
a
```

28

Fold Left (two different versions)

- MIT scheme version:
(fold-left **op** **id** **lst**)
 - op : an binary procedure
 - lst : list of arguments (only one list allowed)
 - apply op left-associatively to elements of lst, and return result of evaluation
 - the identity element **id** is always used
- That is: (one list as an example)
(fold-left op id '()) => **id**
(fold-left op id '(e)) => (op **id** e)
(fold-left op id '(e1 e2 ... en)) => (op... (op (op **id** e1) e2) ...en)

29

fold-left (MIT version)

```
> (fold-left list 'a '(1 2 3))
(((a 1) 2) 3)

>(fold-left cons 'a '(1 2 3))
(((a . 1) . 2) . 3)

> (fold-left list 'a '())
a
> (fold-left cons 'a '())
a
```

30

Define fold-left using foldl and vice versa?

How to define a limited fold-left using foldl in PLT scheme, and in MIT scheme, how to define foldl using fold-left?

Bonus question for assignment 2.

31

Reduce-right (MIT Scheme)

- (reduce-right op id lst)
 - op : an binary procedure
 - lst : list of arguments
 - apply op right-associatively to elements of lst return result of evaluation:if lst is empty, return id; if lst has one element, return that element.

That is:

```
(reduce-right op id '()) => id
(reduce-right op id '(e)) => e
(reduce-right op id '(e1 e2 ... en)) =>
  (op e1 (op e2 (op ... (op e_{n-1} en))))
```

32

Reduce-right (example)

Higher-order Procedures: reduce-right

```
(reduce-right list '() '(1 2 3 4))  
=> (1 (2 (3 4)))
```

```
(reduce-right list '() '(1 2 3 4))  
(list 1 (reduce-right list '() '(2 3 4)))  
(list 1 (list 2 (reduce-right list '() '(3 4))))  
(list 1 (list 2 (list 3 (reduce-right list '() '(4))))))  
(list 1 (list 2 (list 3 4)))  
(list 1 (list 2 '(3 4)))  
(list 1 '(2 (3 4)))  
(1 (2 (3 4)))
```

33

Define reduce-right

- Exercise: define our own reduce-right (myreducer)

```
(define myreducer  
(lambda (op id lst)  
  (cond ((null? lst) id)  
        ((null? (cdr lst)) (car lst))  
        (else (op (car lst)  
                  (myreducer op id (cdr lst)))))))
```

34

More Practice

- Suppose we want a procedure that will test every element of a list and return a list containing only those that pass the test
- We want to be very general:
 - Use any test we might give it

```
(define (prune test lst)  
  (cond ((null? lst) '())  
        ((test (car lst))  
         (cons (car lst) (prune test (cdr lst))))  
        (else (prune test (cdr lst)))))
```

35

Other Useful Scheme: set!, Vectors, Strings, sequencing, assoc

36

set!

- Global Assignment (Generally EVIL) – avoid if possible
- Memory location are:
 - Maintained after the procedure call is complete
 - Are used for their values in this or other procedure
- set! is useful for counters

```
(define cons-count 0)
```

```
(define (cons-co x y)
```

```
  (set! cons-count (+ cons-count 1))
```

```
  (cons x y))
```

```
>(cons-co 'a '(b c)) ;(a b c)
```

```
>cons-count ;1
```

```
>(cons-co 'a (cons-co 'b 'c)) ;(a b . c)
```

```
>cons-count ;3
```

Strings

- Sequence of characters
- Written within double quotes, e.g., “hi mom”
- Useful string predicate procedures:


```
(string=? ...)
```

```
(string<? ...)
```

```
(string<=? ...) ;etc.
```
- Case-insensitive versions


```
(string-ci=? ...)
```

```
(string-ci<? ...)
```

```
(string-ci<=? ...) ;etc.
```
- Other procedures:


```
(string-length <string>)
```

```
(string->symbol <string>)
```

```
(symbol->string <symbol>)
```

```
(string->list <string>)
```

```
(list->string <list>)
```

Scheme: vectors

- **Another compound structure** (*lists, paris*)
 - Similar to lists, it can hold heterogeneous elements



- **What’s the problem with lists?**

– Access time... Vectors are accessed in constant time.

- **Example:**

```
]=>(make-vector 5 0)
```

; creates a vector of length 5 initialized to 0

```
]=> #(0 0 0 0 0)
```

; prefix notation # to do the same thing

```
]=>(vector-length (make-vector 150))
```

; returns 150

```
]=>(vector-ref #(a b c) 2)
```

; returns the third element c

```
]=>(let ((v (vector 'a 'b 'c 'd 'e)))
```

```
  (vector-set! v 2 'x))
```

; #(a b x d e)

```
]=>(vector-fill! v y)
```

; replace each element of v with y

```
]=>(vector->list #(a b c))
```

; return list (a b c)

```
]=>(list->vector '(a b c))
```

; return vector #(a b c)

```
]=> (let ((v (make-vector 5)))
```

```
  (for-each (lambda (i)
```

```
    (vector-set! v i (* i i))) '(0 1 2 3 4)) v)
```

sequencing

- **Another control structure** (*we have already seen cond and if*)

- **Syntax:**

(begin exp₀ exp₁ ... exp_n)

– expressions are evaluated from left to right

– The value of the rightmost expression is the value of the entire expression. Other expressions are for side-effects only

- **Example:**

```
]=> (define x 3)
```

```
]=> (begin
```

```
  (set! x (+ x 1))
```

```
  (+ x x))
```

; returns 8

assoc function

- **assoc** does lookup in a list

- **Eg:**

```
] => (define NAMES '((Smith Pat Q)
                    (Jones Chris J)
                    (Walker Kelly T)
                    (Thompson Shelly P)))
```

```
] => (assoc 'Smith NAMES)
(Smith Pat Q)
```

```
] => (assoc 'Walker NAMES)
(Walker Kelly T)
```

- **assoc** returns the first sublist if more than one sublist with the same key exist.