

# Today

## CSC324H Principle of Programming Languages (Week 4)

*Yilan Gu*

yilan@cdf.toronto.edu

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

- Assignment 2 is due Oct. 8, 2009, 6:00pm sharp
- Term test 1: Oct. 8, from 6:10-7:00pm, BA1210
  - Closed book
  - 20% policy
  - Content covers up till today's material
  - Check course website for more details
- Scheme
  - Different types of recursion
  - Efficiency and tail recursion
  - Let, let\*
  - Unbounded lambda and a taste of high-order functions

1

2

## Basic Functions

- Predicate functions
- (if <predicate> <consequence>  
<alternative>) ; optional
- (cond (<p1> <e1>  
<p2> <e2>  
...  
(else <e>)) ; optional
- (lambda (<formal parameters>) <body>)
- (define (f x1 ... xm) <body>)
- (define f (lambda (x1 ... xm) <body>))

## Review

3

4

## Basic Recursion

- Recursion general strategies
- Several example: computing  $x^n$ ,  $n!$

Now, Let's continue today ...

5

## Different Types of Recursion

6

### Length of a list

- Given a list, compute its length. There is already a built-in length function that computes this. We can also define our own version of length:  

```
(define (length x)
  (cond ((null? x) 0)
        (else (+ 1 (length (cdr x)))))
  )
)
```

```
> (length '(1 2 3))
3
```
- The recursion used in length is called "**cdr-recursion**".
  - At each step, a shorter list is passed to the next function call.
  - Also a **linear** recursion.

7

### Tracing Length

- Tracing (by hand) a call to length:  
Call: (length '(a b c))  
Trace:  

```
(length '(a b c))
(+ 1 (length '(b c)))
(+ 1 (+ 1 (length '(c))))
(+ 1 (+ 1 (+ 1 (length ())))))
(+ 1 (+ 1 (+ 1 0)))
(+ 1 (+ 1 1))
(+ 1 2)
3
```

8

## Tail Recursion

Tail-recursion:

- There is at most one recursive call made in any execution of function body.
- The recursive call is in the last function application in function body.
- Can drastically decrease the amount of stack space used and improve efficiency in implementation.
- Tail-recursion is implemented very efficiently and should be used whenever possible.

9

## Atomcount

- Parameter: a (possibly nested) list.
- Result: the number of atoms in the list.

```
(define (atomcount x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (atomcount (car x)) (atomcount (cdr x))))))
```

```
> (atomcount '(1 2))
```

```
2
```

```
> (atomcount '(1 (2 (3)) (5)))
```

```
4
```

- This is called "[car-cdr recursion](#)".
  - We go off in two directions at once.

11

## Tail recursion: length

- Program:

```
(define (length lst) (length-tail lst 0))

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

```
(length-tail '(a b c) 0)
=> (length-tail '(b c) 1)
=> (length-tail '(c) 2)
=> (length-tail '() 3)
<= 3
```

10

## Append

- `append` is a built-in function that, given two lists L1 and L2, returns a list formed by appending L2 to L1.

```
> (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 '() '())
```

```
()
```

12

## Define Append

```
(define (append list1 list2)
  (if (null? list1) list2
      (cons (car list1)
            (append (cdr list1) list2))))
```

- It is a **flat** recursion
  - recursion only applies to the “top” elements of a list

13

## Odd and Even

- (evenlen? lst) returns #t if length of lst is even, #f otherwise;  
(oddden? lst) returns #t if length of lst is odd, #f otherwise

```
(define evenlen?
  (lambda (lst)
    (if (null? lst) #t
        (oddden? (cdr lst)))))
```

```
(define oddlen?
  (lambda (lst)
    (if (null? lst) #f
        (evenlen? (cdr lst)))))
```

- It is a **mutual** recursion
  - Two functions that call each other rather than themselves

15

## Flatten a List

- (flatten lst) returns a flat list that contains all elements in, lst, on any level, in order

```
(define flatten
  (lambda (lst)
    (cond ((null? lst) '())
          ((list? (car lst))
           (append (flatten (car lst)) (flatten (cdr lst))))
          (else (cons (car lst) (flatten (cdr lst)))))))
```

- This is **deep** recursion
  - (aka tree recursion) recursion applied over all items
  - e.g., car-cdr recursion

14

## Recursion Type Summary

- cdr recursion
- car-cdr recursion
- Linear recursion
- Flat recursion
- Deep recursion
- Mutual recursion
- A special recursion -- **tail** recursion (Efficiency)

**Different classifications from different aspects**

16

## Efficiency

17

### Efficiency: Helper Function

- One solution: Bind values to parameters in a helper function:

```
(define (maximum x y) ; or use the built-in max function.
  (cond ((> x y) x)
        (else y)))
(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        (else (maximum (length x) (length y)))
  ))
```

```
> (longest-nonzero '(a b c) '(a b))
3
```

- Observe that length is now called on each argument just once.
  - The results can be used more than once within the helper function, since they are bound to the helper function's parameters.

19

## Efficiency

- A function that, given two lists, returns -1 if both lists are empty, and otherwise returns the length of the longest list:

```
(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        ((> (length x) (length y)) (length x))
        (else (length y))
  )
)
```

- Problem: Evaluating the same expression twice.
  - length is called on the same argument more than once.
  - We'd like to be able to reuse the result instead.
- Without an assignment statement, what can we do?

18

### Efficiency: Let, Let\* and Letrec

- What if we don't want to define a helper function? How can we still reuse the results of a function call?
- Solution: Use a let, let\* or letrec construct that binds variables to expression results.
- General form:  
**(let ((var1 expr1) ... (varn exprn))  
 <expression>)**  
**(let\* ((var1 expr1) ... (varn exprn))  
 <expression>)**
- This is not the same as variable assignment, since it doesn't let us modify the value of a variable.
  - This is just a convenient way of doing what helper functions already let us do.

20

## Efficiency: Let and Let\*

- Both establish the variables to have values in the expression.
  - What's the difference between let and let\*?
- let does the binding in parallel (which means the order of binding has no effect).
- let\* does the binding in sequence.

Earlier definitions can be used in later ones.

For example,

```
> (let ((x 2) (y (+ x 1))) (+ x y))
```

Error: reference to an identifier before its definition: x

```
> (let* ((x 2) (y (+ x 1))) (+ x y))
```

5

21

## Longest-nonzero with let

```
(define (longest-nonzero x y)
```

```
  (let ((lenx (length x))
```

```
        (leny (length y)) )
```

```
    (cond ((and (= 0 lenx) (= 0 leny)) -1 )
```

```
          (> lenx leny) lenx )
```

```
          (else leny )
```

```
    )))
```

- Observe that length is called on each argument just once.
- Another possible improvement:
  - Note that length gets called (twice) even when x and y are both empty.
  - It might be faster to perform a null? test first, and postpone the let definitions until after this test.

23

## Let and Let\* Examples

```
> (let ((x 2)) (* x x))
```

4

```
> (let ((x 4)) (let ((y (+ x 2))) (* x y)))
```

24

```
> (let ((x 4) (y (+ x 2))) (* x y))
```

reference to undefined identifier: x

```
> (let* ((x 4) (y (+ x 2))) (* x y))
```

24

```
> (let ((x 4)) (let ((x 6) (y (+ x 2))) (* x y)))
```

36

```
> (let ((x 4)) (let* ((x 6) (y (+ x 2))) (* x y)))
```

48

22

## Longest-nonzero yet again

```
(define (longest-nonzero x y)
```

```
  (if (and (null? x) (null? y))
```

```
      -1
```

```
      (let ((lenx (length x))
```

```
            (leny (length y)) )
```

```
        (if (> lenx leny) lenx leny)
```

```
      )))
```

24

## Another Inefficient Example

- Let's write a function `rev`, to return its parameter with the elements in reverse order.
- Note that there is already a built-in reverse function that does this.

```
(define (rev lst)
  (cond ((null? lst) ())
        (else (append (rev (cdr lst))
                        (list (car lst))))))
```

```
> (rev '(1 2 3))
(3 2 1)
```

- It works, but there are a lot of list operations going on.

25

## Tracing rev

Call: (rev '(a b c d))

Trace:

```
(rev '(a b c d))
(rev-acc '(a b c d) ())
(rev-acc '(b c d) '(a))
(rev-acc '(c d) '(b a))
(rev-acc '(d) '(c b a))
(rev-acc () '(d c b a))
'(d c b a)
```

- Note that whenever `rev-rec` makes a recursive call, it returns whatever the recursive call returns (there is no further computation).

27

## A More Efficient Way: Tail Recursion

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

```
> (rev '(a b c d))
(d c b a)
```

- Now each element of the original list only needs to be added to another list once, and it goes on the front, where the work is cheap.
- Observe that `rev-acc`'s second parameter "accumulates" the result.

26

## 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.

28

## Unbounded Lambda

- Write a function (sum ...), which has unlimited number of arguments and it returns the sum of all arguments. E.g.,  
> (sum) ; if there is no argument  
“expecting at least one argument”  
> (sum 0)  
0  
> (sum 3 4 6)  
13

## Unbounded Lambda

29

30

## 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))))))
```

## A Taste of Higher-Order Functions

31

32

## Apply

- `(apply proc obj1 ... objn 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`

33

## Exercises (1)

- write a function called `swapFirstTwo` that takes a list `L`, and swaps the first two elements of `L`. e.g:  
`> (swapFirstTwo '(1 2 3 4))`  
`(2 1 3 4)`
- Write a function called `swapTwoInLists` that takes a list `L` whose elements are themselves lists, and returns a list of all the elements in all the lists in `L`, but with the first two elements in each list swapped. e.g.  
`> (swapTwoInLists '((1 2 3) (4 5 6) (7 8)))`  
`(2 1 3 5 4 6 8 7)`

35

## Map

- `(map proc list1... listn)`
- All lists must be the same length
- returns: A list of applying `proc` to all  $i^{\text{th}}$  ( $i=1..n$ ) 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)`

34

## Exercises (2)

- Write a function called `cdrLists` that takes a list `L` whose elements are themselves lists, and returns a list giving all the elements in the `cdrs` of these lists. e.g:  
`> (cdrLists '((1 2) (3 4 5) (6)))`  
`(2 4 5)`
- Write a function called `addSums` that takes a list `L` of numbers, and returns the total of all sums from 0 to each number. e.g.  
`> (addSums '(1 3 5))`; this is  $1 + 6 + 15$   
`22`
- Re-write `addSums` so that your solution uses tail recursion. You'll need to write an appropriate helper function.

36