

# 6.001 Tutorial 3 Notes

Gerald Dalley  
22 Feb 2005

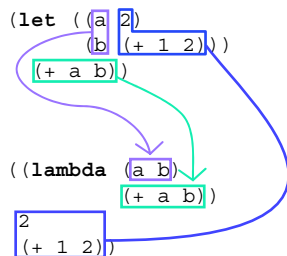
## Orders of Growth Review

The simplest way of determining order of growth is to figure out how much extra work is necessary when you change the size of the problem. A few rules you can *generally* use to solve by inspection ( $n$  is the size of the old problem,  $n'$  is the size of the new problem):

- If  $n' = n - C$ , the order is  $\Theta(n)$ .
- If  $n' = \frac{n}{C}$ , the order is  $\Theta(\log n)$ .
- If  $n' = (n - C) + (n - C)$ , the order is  $\Theta(2^n)$ .

## The *let* Special Form

*let* is a piece of syntactic sugar to allow us to create temporary local variables. We do transformations of the form:



Rewrite

```
(define (greatest-value x y)
  (max (+ x y) (- x y)))
```

by using the *let* special form instead of the *max* procedure:

```
(define (greatest-value x y)
  (let ((a (+ x y))
        (b (- x y)))
    (if (a > b)
        a
        b)))
```

## Pairs and Lists

The *cons* procedure creates a *cons cell* (also known as a pair), which has two elements, the *car* and *cdr*. The *car* and *cdr* can both have any type of value. We draw pairs as two boxes stuck together, where the first box is the *car*, the second is the *cdr*. We draw arrows out of the boxes to show what values they have – always drawing a down arrow out of the *car*, a right arrow out of the *cdr*.

We can chain *cons* cells together to create lists. A list is a set of *cons* cells where the *cdr* of one points to the next, and the *cdr* of the last *cons* cell points to *nil*, the special value that means the empty list. For various (sometimes) useful reasons, our implementation of Scheme defines *nil* to be the same as *false* – *nil*, *()*, and *#f* all mean exactly the same thing.

The basic procedures that you need to understand and be familiar with are *cons*, *car*, *cdr*, and *list*. For the most part, we don't care about dotted pairs, so *cons* takes something and a list (possibly empty), and sticks that something on the front of the list. *car* takes a non-empty list and returns the first thing in it; *cdr* returns everything but the first thing. *list* takes any number of things and returns a list of them all.

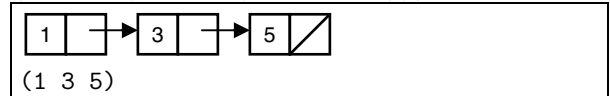
## Boxes and Pointers

For the following problems, give the box-and-pointer representation and the printed representation.

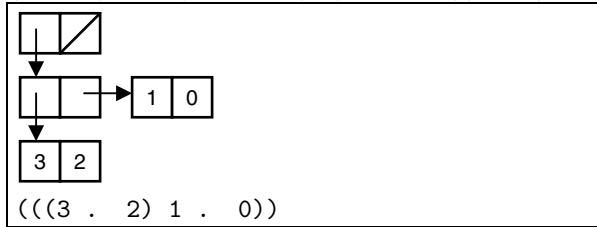
(cons 1 2)



(cons 1 (cons 3 (cons 5 nil)))



```
(cons (cons (cons 3 2) (cons 1 0)) nil)
```



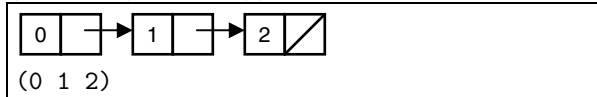
```
(car lst)
(list-ref (cdr lst) (- n 1)))
```

Time =  $\Theta(n)$ , Space =  $\Theta(1)$ ,  $n = n$

```
;; This procedure joins two lists together
;; e.g. (append (list 1 2) (list 3 4))
;; gives (1 2 3 4)
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))
```

Time =  $\Theta(n)$ , Space =  $\Theta(n)$ ,  $n = \text{length of lst1}$

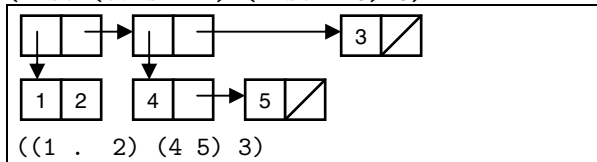
```
(cons 0 (list 1 2))
```



```
;; This procedure reverses the order of
;; elements in a list append may be useful
(define (reverse lst)
  (if (null? lst)
      nil
      (append (reverse (cdr lst))
              (list (car lst)))))
```

Time =  $\Theta(n^2)$ , Space =  $\Theta(n)$ ,  $n = \text{length of lst}$

```
(list (cons 1 2) (list 4 5) 3)
```



Write Scheme code that would produce the following printed representations.

```
(1 2 3)
```

```
(list 1 2 3)
```

```
(1 2 . 3)
```

```
(cons 1 (cons 2 3))
```

```
;; Swaps the first and second items in
;; a list that has at least 2 items
;; (swap-first-and-second (list 1 2 3))
;; ; Value: (2 1 3)
(define (swap-first-and-second lst)
  (cons (cadr lst) (cons (car lst) (cddr lst))))
```

Time =  $\Theta(1)$ , Space =  $\Theta(1)$ ,  $n = \text{length of lst}$

```
((1 2) (3 4) (5 6))
```

```
(list (list 1 2) (list 3 4) (list 5 6))
```

## HOPs

Higher-order procedures are procedures that either accept procedures as arguments or return procedures as their values.

## Practice Problems

```
;; This procedure returns the length
;; (i.e. number of elements) in a list
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

Time =  $\Theta(n)$ , Space =  $\Theta(n)$ ,  $n = \text{length of lst}$

```
;; This procedure applies f to each
;; element of the list, and returns a
;; new list made from those values
(define (map f lst)
  (if (null? lst)
      nil
      (cons (f (car lst))
            (map f (cdr lst)))))
```

Time =  $\Theta(n\Theta f(\cdot))$ , Space =  $\Theta(n\Theta f(\cdot))$ ,  
 $n = \text{length of lst.}$

```
;; This procedure returns the nth
;; element of a list, where the first
;; element has index 0
(define (list-ref lst n)
  (if (= n 0)
```