

6.001 Tutorial 3 – Solutions

David Ziegler

September 27, 2004

1 Reminders

- Problem set 3 is due tomorrow at midnight.
- Quiz 1 is a week from Thursday!
Please email me at david@ziegler.ws if you have things you'd like to cover next week.

2 Orders of Growth

The *order of growth* of a procedure gives us some notion of the amount of a resource consumed by it – not a precise value, but a useful way of approximating the behavior of the function for different inputs. The order of growth of a function is given in terms of the “size of the problem,” typically a function of the input arguments.

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 $\Theta(2^n)$.

In writing the order of growth, we discard constants and lower order terms. As n gets to be very large, the highest order term dominates, so lower order terms don't matter. Since we care about the behavior of the function as the n gets large, lower order terms are insignificant. The same argument holds for the constants.

We care about growth for both time and space. We measure growth in time in terms of the number of primitive operations performed and space in terms of the number of deferred operations.

3 let

```
(let ((name value-exp)
      (name value-exp)
      ...))
  expr
  expr
  ...)
```

The `let` special form temporarily binds the names to the values of the `value-exps`, and evaluates the `exprs`. The value of the entire `let` expression is the value of the last `expr`.

Like many special forms, `let` can be desugared into a `lambda`:

```
((lambda (name name ...)
  expr
  expr ...)
  value-exp
  value-exp ...)
```

4 cons 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.

```
;; 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 = 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)
      (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 = length of lst1.

```
;; 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 = length of lst.

```
;; This procedure returns the first
;; pair of lst whose car is obj;
;; if obj is not in lst, #f is returned
(define (member obj lst)
```

```
(cond ((null? lst) #f)
      ((equal? (car lst) obj) #t)
      (else (member obj (cdr lst)))))
```

Time = $\Theta(n)$, space = $\Theta(1)$, n = length of lst.

```
;; This procedure returns a new list
;; that has exactly one instance of
;; each value in the original
(define (remove-duplicates lst)
  (cond ((null? lst) nil)
        ((member (car lst) (cdr lst))
         (remove-duplicates (cdr lst)))
        (else
         (cons (car lst)
               (remove-duplicates (cdr lst))))))
```

Time = $\Theta(n^2)$, space = $\Theta(n)$, n = length of lst.