MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Wednesday, March 1

1. Some of the Important Things to Know

- Rules for Evaluation (substitution model) – when in doubt **stop thinking!**
- Rules for Special Forms
- Iteration vs. Recursion
- Pairs, Lists, and Trees (defining, manipulating, contracts, abstractions)
- Higher Order Procedures (procedures returning procedures)
- Abstraction of List Operations (`map`, `accumulate`, and `filter`.

2. Tricky Stuff

What are the values of the following expressions:

```
(if 0 1 2) ⇒

(define (my-if pred conseq alt)
  (if (zero? pred) alt conseq))

(my-if 0 1 2) ⇒

(define (factorial n)
  (my-if n
         (* n (factorial (- n 1)))
         1))

(fact 5) ⇒
```

```
(define x 5)
(define y 6)
(let ((x 7)
      (y x))
  (+ x y))  ⇒

((lambda (x y) ((y 6) x)) 4
  (lambda (w) (lambda (z) (* 2 z))))  ⇒

(list 1 (list 2 list 3) 4)
  ⇒

((if + - *) 4 3)  ⇒
```

3. Writing Some Procedures

Write the following procedures:

The procedure `count-pairs` that counts the number of `cons` pairs in a tree structure.

```
(count-pairs (list 2 (list 3 4) (list 5))) ⇒6

(define (count-pairs tree)



)
```

The procedure `copy-some` that copies the first `n` elements of a list

```
(copy-some 3 (list 1 2 3 4 5)) ⇒(1 2 3)

(define (copy-some n lst)



)
```

4. Order of Growth

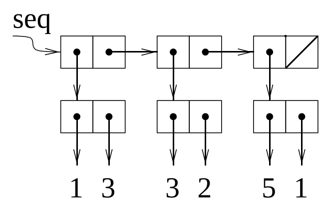
What's the order of growth of the procedure `copy-some` above? $\Theta(\quad)$. Consider the following procedure to copy the last n elements of a list. What is the order of growth of `last-n`? $\Theta(\quad)$.

```
(define (last-k k lst)
  (if (= (length lst) k)
      lst
      (last-k k (cdr lst))))
```

5. Defining a New List Abstraction

Notice that it can take a long time to find the length of one of our list structures. Say we want to define a new sequence abstraction, similar to lists, but that can return the length in constant time. Here's the contract:

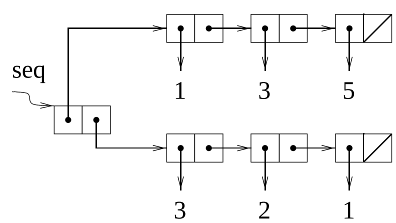
```
(head (attach x seq)) == x
(tail (attach x seq)) == seq
(seq-empty? empty-seq) == #t
(seq-empty? (attach x seq)) == #f
(seq-length seq) == the length in Θ(1) time.
```



How can we do this? Let's define a sequence as show to the right. The list (1 3 5) would be represented as a list of pairs. The `car`s are the elements of the list, and the `cdr`s are the lengths of the lists. Fill in the blanks below to complete the abstraction.

```
(define empty-seq nil)
(define seq-empty? null?)

(define (head seq)                          )
(define (tail seq)                          )

(define (seq-length seq)
  (if (seq-empty? seq)
      0
                               ))

(define (attach x seq)

)
```

```
(define (list->seq lst)





)

(define (seq->list seq)

)
```



How about another way? Here a sequence is a pair of two things. The `car` is the original list, and the `cdr` stores the lengths of the list. Fill in the blanks below to complete the abstraction.

```
(define empty-seq (cons nil nil))
(define (seq-empty? seq) (null? (car seq)))

(define (head seq)                                 )
(define (tail seq)                                 )

(define (seq-length seq)
  (if (seq-empty? seq)
      0
                                                 ))

(define (attach x seq)


)
```

```
(define (list->seq lst)





)

(define (seq->list seq)

)
```

Notice that with either of these abstractions, lists behave in the same way as they did before, except that the length of a list can be computed in constant time. We traded time for space.