MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes, Sept. 17 – Compound Data and List Processing

Pair Abstraction

1. Constructor

   (cons <x-exp> <y-exp>) ==> <P>
   - <x-exp> and <y-exp> evaluate to values <x-val> and <y-val> of any Scheme type;
   - returns a pair <P> whose car-part is <x-val> and whose cdr-part is <y-val>

2. Accessors

   (car <P>) ==> <x-val> ; returns the car-part of the pair <P>
   (cdr <P>) ==> <y-val> ; returns the cdr-part of the pair <P>

3. Predicates

   (null? <P>) ==> #t if <P> is empty list, else #f
   (pair? <P>) ==> #t if <P> is a pair, else #f

Box and pointer diagrams help visualize the structure of arbitrarily complex pair structures.

Pairs have the property of *closure*: the value resulting from `cons` can itself be supplied as an argument to another application of `cons`.

List Convention

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

is equivalent to

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

consisting of a “backbone” of cons cells, from which hang the items of the list.

Common Patterns – List Procedures

Common Pattern #1: cdr’ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst) (- n 1))))

(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

Common Pattern #2: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (cons from (enumerate-interval (+ 1 from) to))))
```

Some examples of procedures that both cdr down the list, and cons up a result:

```
(define (copy lst)
  (if (null? lst)
      nil                          ; base case
      (cons (car lst)              ; recursion
            (copy (cdr lst)))))

(define (append list1 list2)
  (if (null? list1)
      list2                  ; base case
      (cons (car list1)      ; recursion
            (append (cdr list1) list2))))  ; BUG CORRECTED
```

Common Pattern #3: transforming a list

```
(define (square-list lst)
  (if (null? lst)
      nil
      (cons (square (car lst))
            (square-list (cdr lst)))))

(define (map proc lst)
  (if (null? lst)
      nil
      (cons (proc (car lst))
            (map proc (cdr lst)))))

(define (square-list lst) (map square lst))
```

Common Pattern #4: filtering

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

Common Pattern #5: accumulation

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (car lst)
         (add-up (cdr lst)))))

(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))

(define (add-up lst) (accumulate + 0 lst))
```

Write `length` as an accumulation:

Conventional Interfaces

```
(define (easy lo hi)
  (accum * 1
         (map fib
              (filter even?
                      (integers-between lo hi)))))
```

`Easy` as a series of black boxes connected by lists:

```
(define (hard lo hi)
  (cond ((> lo hi) 1)
        ((even? lo) (* (fib lo)
                       (hard (+ lo 1) hi)))
        (else (hard (+ lo 1) hi))))
```