Pair Abstraction

1. Constructor
   \[(\text{cons } \langle x\text{-exp} \rangle \langle y\text{-exp} \rangle) \Rightarrow \langle P \rangle\]
   - \(\langle x\text{-exp} \rangle\) and \(\langle y\text{-exp} \rangle\) evaluate to values \(\langle x\text{-val} \rangle\) and \(\langle y\text{-val} \rangle\) of any Scheme type;
   - returns a pair \(\langle P \rangle\) whose car-part is \(\langle x\text{-val} \rangle\) and whose cdr-part is \(\langle y\text{-val} \rangle\)

2. Accessors
   \[(\text{car } \langle P \rangle) \Rightarrow \langle x\text{-val} \rangle; \text{returns the car-part of the pair } \langle P \rangle\]
   \[(\text{cdr } \langle P \rangle) \Rightarrow \langle y\text{-val} \rangle; \text{returns the cdr-part of the pair } \langle P \rangle\]

3. Predicates
   \[(\text{null? } \langle P \rangle) \Rightarrow \#t \text{ if } \langle P \rangle \text{ is empty list, else } \#f\]
   \[(\text{pair? } \langle P \rangle) \Rightarrow \#t \text{ if } \langle P \rangle \text{ is a pair, else } \#f\]

Box and pointer diagrams help visualize the structure of arbitrarily complex pair structures.
Pairs have the property of closure: the value resulting from \texttt{cons} can itself be supplied as an argument to another application of \texttt{cons}.

List Convention

\[(\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 3 \ (\text{cons } 4 \ \text{nil}))))\]
is equivalent to
\[(\text{list } 1 \ 2 \ 3 \ 4)\]
consisting of a “backbone” of cons cells, from which hang the items of the list.

Common Patterns – List Procedures

Common Pattern \#1: \texttt{cdr’ing} down a list

\[
\begin{align*}
\text{(define (list-ref lst n)} \nonumber \\
\text{ (if (= n 0)} \nonumber \\
\text{ (car lst)} \nonumber \\
\text{ (list-ref (cdr lst) (- n 1)))))}
\end{align*}
\]
\[
\begin{align*}
\text{(define (length lst)} \nonumber \\
\text{ (if (null? lst)} \nonumber \\
\text{ 0)} \nonumber \\
\text{ (+ 1 (length (cdr lst)))))}
\end{align*}
\]
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)
  (accumulate * 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))))