New Procedures and Predefined Values

1. \((\text{cons a b})\) - makes a \text{cons}-cell (pair) from \text{a} and \text{b}
2. \((\text{car c})\) - extracts the value of the first part of the pair.
3. \((\text{cdr c})\) - extracts the value of the second part of the pair.
4. \((\text{caddr c})\) - shortcuts (\text{e.g.} \((\text{caddr c}) \equiv (\text{car (cdr (cdr c))})\)).
5. \((\text{list a b c ...})\) - builds a list of the arguments to the procedure.
6. \((\text{adjoin a lst})?\) - doesn’t exist (use \text{cons})
7. \((\text{list-ref lst n})\) - returns the \text{n}th element of \text{lst}.
8. \((\text{append l1 l2})\) - make a new list containing the elements of both lists.
9. The empty list...
   • '() - the safe way of specifying the empty list
   • \text{null} - a DrScheme-specific definition
   • () - another DrScheme-specific definition (what DrScheme prints)
   • \text{nil} - an MIT Scheme-specific definition (what the book uses)
   • #f - another MIT Scheme-specific definition (what the tutor prints)
10. Testing for the empty list...
    • \text{null?} - The only safe way of testing for the empty list.

car/cdr history...

Which of \text{cons/car/cdr/list} are special forms?

\((\text{car (cons (+ 3 4) (/ 4 0)))}\)

Box and Pointer Diagrams

Diagramming Rules

1. Any time you see \text{cons}, draw a double box with 2 pointers
2. Evaluate the stuff inside & point to it
3. Denote \text{null}'()/empty list with a slash through a box.
4. Any time you see \text{list} with \text{n} items, draw a chain of \text{n} \text{cons} cells
5. \text{list} with no items is the empty list

Printing a \text{cons} structure

1. Each \text{cons} cell is printed ( \text{car-part . cdr-part} )
2. Cross out any “. \text{null}”
3. Cross out any “. (” and its matching “)”
4. The empty list is printed as () in DrScheme
Practice

(define a (cons 1 2))
(define b (list (cons 1 2) (cons 1 2)))
(define c (list a a))
(define d (cons 2 '()))
(define e (cons '() 2))
(define f (list 2))
(define g (list))
(define h (list '()))
(define i (list 1 2 3 4))
(define j (list 5 (cdr (cdr i)) (cons 6 7)))

Draw the box-and-pointer diagrams here:

What’s the minimum number of cons cells needed to store n items?  

Write the printed form of the a – j examples above
List Problems

“cdr-ing down a list”

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

(length (list 1 3 4 7)) →
(length j) →

;; Write an iterative version...
(define (length lst)

“cdr-ing down the input and cons-ing up a result”

;; Write cube-neighbor-diff
;; (cube-neighbor-diff (list 1 3 4 7)) => (8 1 27)
;; Takes the difference between neighboring values then cubes the difference.
(define (cube-neighbor-diff l)
In our fictitious consulting firm, we began developing a fast-food order processing system. We built abstractions for combos and orders, but nearly every procedure had to know that our low-level representation used digits 1–4 for non-biggie-sized combos and 5–8 for the corresponding biggie-sized combos. What if we wanted to add salads, baked potatoes, drinks, apple pies, and so forth? We’d have to rewrite nearly all of our code! Let’s create some better abstractions.

We’ll break up our consulting team into several subteams. For orders-of-growth questions, use \( I \) as how many types of items there are and \( N \) as the number of items in an order.

1. Implement the item abstraction. Do intelligent things for cases like \( \text{biggie-size (make-salad)} \). Come up with some additional high-level operations/constructors/accessors that might be useful but don’t break the abstraction barrier.
2. Implement the order abstraction by keeping a list of items, sorted by the order they were added. Example: \( \text{(item1 item2 item3 ... itemn)} \). What are the orders of growth for each procedure?
3. Implement the order abstraction by keeping a list of items and their counts, e.g. \( \text{(salad 1) (single 4) (biggie-quad 7)} \). What are the orders of growth for each procedure? Why would you ever want this representation?
4. Implement the order abstraction by keeping a list of each item ordered, sorted by price, e.g. \( \text{(salad single single single single biggie-quad)} \). What are the orders of growth for each procedure? Why would you ever want this representation?
Announcements / Notes

• How many of you are taking 8.02 or otherwise have a conflict with the quiz (8 Mar, 7:30-9:30pm)?
• How many would have a conflict with the exam on Wednesday 7 Mar at 7:30pm?
• Anyone with hard conflicts for both times?

New Procedures and Predefined Values

1. (cons a b) - makes a cons-cell (pair) from a and b
2. (car c) - extracts the value of the first part of the pair.
3. (cdr c) - extracts the value of the second part of the pair.
4. (caddr c) - shortcuts (e.g. (caddr c) ≡ (car (cdr (cdr c)))).
5. (list a b c ...) - builds a list of the arguments to the procedure.
6. (adjoin a lst) - doesn’t exist (use cons)
7. (list-ref lst n) - returns the n\textsuperscript{th} element of lst.
8. (append l1 l2) - make a new list containing the elements of both lists.
9. The empty list...
   • '() - the safe way of specifying the empty list
   • null - a DrScheme-specific definition
   • () - another DrScheme-specific definition (what DrScheme prints)
   • nil - an MIT Scheme-specific definition (what the book uses)
   • #f - another MIT Scheme-specific definition (what the tutor prints)
10. Testing for the empty list...
    • null? - The only safe way of testing for the empty list.

car/cdr history...

Which of cons/car/cdr/list are special forms?

none – they work like regular combinations
(car (cons (+ 3 4) (/ 4 0)))

Box and Pointer Diagrams

Diagramming Rules

1. Any time you see cons, draw a double box with 2 pointers
2. Evaluate the stuff inside & point to it
3. Denote null/'()/empty list with a slash through a box.
4. Any time you see list with n items, draw a chain of n cons cells
5. list with no items is the empty list
Printing a cons structure

1. Each cons cell is printed (car-part . cdr-part)
2. Cross out any “. null”
3. Cross out any “. (“ and its matching “)”)”
4. The empty list is printed as () in DrScheme

Practice

(define a (cons 1 2))
(define b (list (cons 1 2) (cons 1 2)))
(define c (list a a))
(define d (cons 2 '()))
(define e (cons '() 2))
(define f (list 2))
(define g (list))
(define h (list '()))
(define i (list 1 2 3 4))
(define j (list 5 (cdr (cdr i)) (cons 6 7)))

Draw the box-and-pointer diagrams here:

- Don’t forget to label your diagrams with any defined variables
- Arrows that point to cons cells can point to any part of the outer box,
- You can write numbers inside the boxes, or point to them outside the box
- Don’t forget to put something in each box (don’t forget to put a slash in the last box of a list)

What’s the minimum number of cons cells needed to store \( n \) items? \( n - 1 \)
Write the printed form of the a – j examples above

\[a \rightarrow (1 . 2)\]
\[b \rightarrow ((1 . 2) (1 . 2))\]
\[c \rightarrow ((1 . 2) (1 . 2))\]
\[d \rightarrow (2)\]
\[e \rightarrow ((1 . 2))\]
\[f \rightarrow (2)\]
\[g \rightarrow ()\]
\[h \rightarrow ()\]
\[i \rightarrow (1 2 3 4)\]
\[j \rightarrow (5 (3 4) (6 . 7))\]

**List Problems**

“cdr-ing down a list”

\[
\text{(define (length lst)}
\begin{align*}
&\quad \text{(if (null? lst)}
\quad 0
\quad (+ 1 (length (cdr lst))))
\end{align*}
\]

(length (list 1 3 4 7)) \(\rightarrow\) 4
(length j) \(\rightarrow\) 3

;; Write an iterative version...

(define (length lst)
  (define (helper len remainder)
    (if (null? remainder)
      len
      (helper (+ 1 len) (cdr remainder))))
  (helper 0 lst))

; quick verification (should print 4)
(length (list 1 3 4 7))

“cdr-ing down the input and cons-ing up a result”

;; Write cube-neighbor-diff

;; (cube-neighbor-diff (list 1 3 4 7)) \(\rightarrow\) (8 1 27)
;; Takes the difference between neighboring values then cubes the difference.

(define (cube-neighbor-diff l)
  (cond ((null? l) '())
        ((null? (cdr l)) '())
        (else
          (let ((diff (- (cadr l) (car l))))
            (cons (* diff diff diff)
                  (cube-neighbor-diff (cdr l)))))))

(cube-neighbor-diff (list 1 3 4 7))

; alternative implementation:

(define (cube-neighbor-diff l)
  (cond ((null? l) '())
        ((null? (cdr l)) '())
        (else
(cons (cube (- (cdr l) (car l)))
   (cube-neighbor-diff (cdr l)))))

(define (cube x) (* x x x))
In our fictitious consulting firm, we began developing a fast-food order processing system. We built abstractions for combos and orders, but nearly every procedure had to know that our low-level representation used digits 1–4 for non-biggie-sized combos and 5–8 for the corresponding biggie-sized combos. What if we wanted to add salads, baked potatoes, drinks, apple pies, and so forth? We’d have to rewrite nearly all of our code! Let’s create some better abstractions.

```
;;----------------------------------------------------------------
;; item abstraction
;; For pricing, assume patties cost $1.17, biggie-sizing a burger combo costs $0.50, and salads cost $0.99.
;; Constructors
(make-burger-combo num-patties) ; integer -> item
(make-salad) ; void -> item
;; Accessors, etc.
(get-num-patties item) ; item -> integer
(item-price item) ; item -> number
(biggie-size? item) ; item -> boolean
(items-equal? a b) ; item, item -> boolean
;; Operators
(biggie-size item) ; item -> item
(unbiggie-size item) ; item -> item
```

We’ll break up our consulting team into several subteams. For orders-of-growth questions, use $I$ as how many types of items there are and $N$ as the number of items in an order.

1. Implement the item abstraction. Do intelligent things for cases like (biggie-size (make-salad)). Come up with some additional high-level operations/constructors/accessors that might be useful but don’t break the abstraction barrier.
2. Implement the order abstraction by keeping a list of items, sorted by the order they were added. Example: (item1 item2 item3 ... itemn). What are the orders of growth for each procedure?
3. Implement the order abstraction by keeping a list of items and their counts, e.g. ((salad 1) (single 4) (biggie-quad 7)). What are the orders of growth for each procedure? Why would you ever want this representation?
4. Implement the order abstraction by keeping a list of each item ordered, sorted by price, e.g. (salad single single single single biggie-quad). What are the orders of growth for each procedure? Why would you ever want this representation?
The solutions are given in item-1.scm, order-2.scm, order-3.scm, and order-4.scm. A script for automatically validating this code is given in testdriver.scm. These files are embedded in the source .zip file downloadable from http://people.csail.mit.edu/dalleyg/6.001/SP2007/.