Lists

A simple but powerful general-purpose datatype, (How many databases have we seen so far?)

1. #t
   ()
2. (1 2 3 () )

Building block: the cons cell,

1 2 3 ()

Note: Subrata uses Nil, That is LISP notation in Scheme, we use ().

Things you should know about cons, pairs and lists

The pair or cons cell is the most fundamental of Scheme's structured object types

A pair is a sequence of pairs; each pair's car is the next pair in the sequence.

The car of the last pair in a property list is the empty list. Otherwise the sequence of pairs forms an improper list: e.g., an empty list is a proper list, and any pair whose car is a proper list is a proper list.

An improper list is printed in dotted notation following a period (.) preceding the final element of the list. A pair whose car is not a list is often called a dotted pair.

Creating lists

- Quote: '( (2 3) ()) => ( (2 3) ())
- or (quote (1 (2 3) ()) ) => ( (1 (2 3) ()) )
- list: (list ' (2 3) ()) => ( (2 3) () )
- Build it, piece by piece:
  (cons 1 (cons (cons 2 (cons 3 () )) (cons () ())))
- Appending lists:
  (append list ' (4 5) ) => ( ((1 (2 3) ) (4 5) ) )

cons vs. list: The procedure cons actually builds pairs, and there is no reason that the car of a pair must be a list, as illustrated on the previous page.

The procedure list is similar to cons, except that it takes an arbitrary number of arguments and always builds a proper list.

E.g., (list 'a 'b 'c) => (a b c)

Useful predicates

Testing for equality

- eq? a b: Returns #t if a and b are the same Scheme object, (Don't use eq? with numbers!)
- (= a b): Returns #t if a and b are numerically equal, Prog a and b must evaluate to numbers.
- eqv? a b: Similar to eq?, but works for both numbers and characters. More expensive than eq?, however.
- (equal? a b): Returns #t if a and b have the same structure and contents. This, equal? recursively tests for equality. The most expensive equality predicate

Recommended Reading:
- Dygvi 6.1, 2nd ed, (available online), or Dygvi 6.2, 3rd ed.

Equality Checking

The eq? predicate doesn't work for lists,

Why eq?:

1. (cons 'a '()) makes a new list
2. (cons 'a '()) makes a (other) new list
3. eq? checks if its two args are the same
4. (eq? (cons 'a '()) (cons 'a '())) evaluates to t (or #t)

Lists are stored as pointers to the first element (car) and the rest of the list (cdr).

Symbols are stored uniquely, so eq? works on them.

Equality Checking for Lists

For lists, need a comparison procedure to check for the same structure in two lists. How might you write such a procedure?

(define (equal? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x)) (not (atom? y))
           (equal? (car x) (car y))
           (equal? (cdr x) (cdr y))))

- (equal? 'a 'a ) evaluates to #t
- (equal? 'a 'b ) evaluates to ()
- (equal? '(a) '(a) ) evaluates to 10 #t
- (equal? '((a) (a) ) evaluates to ()

Note there is a built-in predicate procedure equal?, Play around with it!

More predefined predicates

- (null? x): Returns #t if x is the empty list (or #t, depending on the implementation).
- (pair? x): Returns #t if x is a pair, i.e., a cons cell.
- (number? x): Returns #t if x is a number.

Lots more in Dygvi §6,

Code as Data—Eval

Scheme code is simply data that is treated as code, If you build an expression, using any data processing technique, and you want to evaluate it as code, use eval:

(define x (+ 4 6))

a => 10

(define b (+ 4 6))

b => (+ 4 6)

(10 b b) => 10

More on this later...

Recursive Procedures: Counting

(define (atomin? x)
  (cond ((null? x) #t)
        ((atom? x) #t)
        (else (atomin? (car x)))))

- (atomin? '(1 2) ) => 2
- (atomin? '(1 (2 (3) ) ) ) => 4

This is called 'car-cdr-recursion;
Problem: Evaluating the same expression twice.

Example:

(define (longest-common x y)
  (cond ((and (null? z) (null? y)) -1)
        ((< (length x) (length y))
         (length x))
        (else (length y))
    ))

What can you do if there is no assignment statement?

Solution 1: Bind values to parameters in a helper procedure.

(define (maximum x y)
  (cond ((> x y) x)
         (else y))
)

(define (longest-common x y)
  (cond ((and (null? z) (null? y)) -1)
        ((> (maximum (length x) (length y)))
         (maximum (length x) (length y)))
        (else (length y))
    ))

Note: There is a built-in max function.

Solution 2: Use a let or let* construct, that binds variables to expression results.

(let ((var exp1)
       ...
       (var expr))
  (let* ((var! expr1)
         ...
         (var! expr)))

Note 2: Helper procedures are an important and useful tool.

Higher-Order Procedures

Procedures as input values:

(define (all num list)
  (for (null? list)
       (and (number? (car list))
            (all num (cdr list)))))

(define (all-count f list)
  (cond ((null? list) (f list))
        (else (f list)
              (all-count f (cdr list)))))

1] ⇒ (all num (abs list '((1 2 3)))
     ;Value: 1: (1 2 3)

1] ⇒ (all null (abs list '((a)))
     ;Value: error

Procedures as returned values:

(define (plus list x)
  (cond ((number? x)
          (lambda (y) (* (sum-list x) y)))
        (list? x)
          (lambda (y) (* (sum-list (car x)) y))
        (else (lambda (x) x))
    ))

1] ⇒ ((plus list 3) 4)
     ;Value: 10

1] ⇒ ((plus list '((1 3 5)) 5)
     ;Value: 14

Built-In Higher-Order Procedures: \(\text{map}\)

(define (map f l)
  (cond ((null? l) ()
          (else (cons (f (car l))
                      (map f (cdr l)))))))

• \(\text{map}\) takes two arguments: a function and a list

• \(\text{map}\) builds a new list whose elements are the result of applying the function to each element of the old list

Higher-Order Procedures: \(\text{map}\)

• Example:

(map abs '((-1 2 3 4)) ⇒
  (1 2 3 4)
(map (lambda (x) (+ 1 x)) '(1 2 3)) ⇒
  (2 3 4))

• Actually, the built-in \(\text{map}\) can take more than two arguments:

(map \(\text{abs}\) '(a b c) \(\text{((1)} (2) (3))\) ⇒
  '(a 1) (b 2) (c 3))

Polymorphic and Monomorphic Functions

• Polymorphic functions can be applied to arguments of many forms

• The function length is polymorphic: it works on lists of numbers, lists of symbols, lists of lists, lists of anything

• The function square is monomorphic: it only works on numbers
What's Wrong Here??

Using eval to Correct the Problem

```lisp
(define (atomcount a)
  (cond ((null? a) 0)
        ((atom? a) 1)
        (else (+ (map atomcount a)))))

;Value: atomcount
1 ]> (atomcount '(a b))
;Value: 2
1 ]> (atomcount '((1 2 3) 4))
;Value: 4
1 ]> (atomcount '(a b (c d e)))
;Value: 6
1 ]> (atomcount '(()))
;Value: 0
```

Limitations of Using eval

BUT: eval only works in the current definition of atomcount because numbers evaluate to themselves.

```lisp
1 ]> (a b)
;Value: 6
1 ]> (cons 'a (b))
;Value: 6
1 ]> (eval (cons 'a (b)))
;Value: 6
```

Too complicated!!

Applying Procedures with apply

```lisp
1 ]> (apply + '(1 2 3))
;Value: 6
1 ]> (apply append '(((a) (b)) (c)))
;Value: 6
1 ]> (apply + (a b))
;Value: 6
```

Higher-order Procedures: reduce

```lisp
(define reduce op 1 sd)
  (if (null? sd)
      0
      (op (car sd)
           (reduce op (cdr sd))))

;Value: reduce
1 ]> (reduce + (1 2 3) 0)
;Value: 6
```

A binary n-ary procedure.

The reduce procedure takes a binary operation and applies it right-associatively to a list of an arbitrary number of arguments.

**NOTE:** `reduce` is not equivalent to `apply`.

Higher-order Procedures: reduce

```lisp
(reduce + '(1 2 3) 0)  => 6
(reduce * '(1 2 3) 0)  => 0
(reduce / '(1 2 3) 0)  => 0
(reduce + (reduce + (1)) (reduce + (2)))  => 3
(reduce / (reduce / (1)) (reduce / (2)))  => 0
(reduce + (reduce + (1)) (reduce + (2)))  => 3
```

**Note:** `reduce` is not equivalent to `apply`.

Higher-order Procedures: reduce

```lisp
Given union, which takes two lists representing sets and returns their union:
1 ]> (apply union '(1 3) (2 4))
;Value: 21: (1 2 3 4)
1 ]> (apply union '(1 3 (2 4 5)))
;The procedure #apply has been called with 3 arguments; it requires exactly 3 arguments.
1 ]> (reduce union '(((1 3) (2 4 5) (6)) (7)))
;Value: 22: (1 2 3 4 5 6 7)
```

**Question:** How would you have to change `reduce` to be able to take intersection as its function argument?