Lists

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

```
(1 #t 1)
()
(1 2 3 ()
```

Building block: the cons cell.

```
1 2 3
```

Note: Sebesta uses NIL. That is LISP notation! In Scheme, we use ()

Creating lists

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

cons vs. list: The procedure cons actually builds pairs, and there is no reason that the cdr 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 iff a and b are the same Scheme object. (Don't use eq? with numbers!)
- `(# a b): Returns #t iff a and b are numerically equal. Pre: a and b must evaluate to numbers.
- `(eqv? a b): Similar to eq?, but works for numbers and characters. More expensive than eq?, however.
- `(equal? a b): Returns #t iff a and b have the same structure and contents. Thus, equal? recursively tests for equality. The most expensive equality predicate.

Recommended Reading:
Dybvig §6.1, 2nd ed. (available online), or Dybvig §6.2, 3rd ed.
Equality Checking

The eq? predicate doesn't work for lists.
Why not?

1. (cons 'a '()) makes a new list
2. (cons 'a '()) makes a(nother) new list
3. eq? checks if its two args are the same
4. (eq? (cons 'a '()) (cons 'a '())) evaluates to () (ie, #f)

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.

More pre-defined predicates

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

Lots more in Dybvig §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 a (+ 4 6))
a => 10
(define b '(+ 4 6))
b => (+ 4 6)
(eval b ()) => 10

More on this later...

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

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

Recursive Procedures: Counting

(define (atomcount x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (atomcount (car x))
                 (atomcount (cdr x))))))

- (atomcount '(1 2)) => 2
- (atomcount '(1 (2 (3)) (5))) => 4:
  (at '(1 (2 (3)) (5)))
  (+ (at 1) (at '(1 (2 (3)) (5))))
  (+ 1 (+ (at 2) (at '(1 (2 (3)) (5)))))
  (+ 1 (+ 1 (+ (at 3) (at '(1 (2 (3))) (5)))))
  (+ 1 (+ 1 (+ 1 (+ (at 5) (at '(1 (2 (3))) 0))))
  (+ 1 (+ 1 (+ 1 (+ 1 (at 5) (at '(1 (2 (3))) 0)))) (+ 1 0) 0))
  (+ 1 (+ 1 (+ 1 (+ 1 0) 0)) (+ 1 0))
  (+ 1 (+ 1 (+ 1 0) 0))
  (+ 1 (+ 1 0) 0)
  (+ 1 (+ 1 0) 0)
  (+ 1 (+ 2 1))
  (+ 1 3)
  4

This is called "car-cdr-recursion."
**Efficiency Issues**

**Problem:** Evaluating the same expression twice.

Example:

```scheme
(define (longest-nonzero x y)
  (cond ((and (null? x) (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.

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

```scheme
(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        (else
         (maximum (length x) (length y))))
)
```

Note: There is a built-in `max` function.

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

---

**Efficiency Issues**

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

```scheme
(let ((var1 expr1)
       ...
       (varn exprn))
  <vars are defined and can be used here>)
(let* ((var1 expr1)
       ...
       (varn exprn))
  <vars are defined and can be used here>)
```

---

**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
Higher-Order Procedures

Procedures as input values:

(define (all-num lst)
  (or (null? lst)
      (and (number? (car lst))
           (all-num (cdr lst)))))
)

(define (all-num-f f lst)
  (cond ((all-num lst) (f lst))
        (else 'error))
)

1 ]=> (all-num-f abs-list '(1 -2 3))
;Value: 1 (2 3)

1 ]=> (all-num-f abs-list '(1 a))
;Value: error

Built-In Higher-Order Procedures: map

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

• map takes two arguments: a function and a list
• map builds a new list whose elements are the result of applying the function to each element of the (old) list

Higher-order Procedures: map

• Example:

(map abs '(-1 2 -3 4)) ⇒
(1 2 3 4)

(map (lambda (x) (+ 1 x)) '(-1 2 -3)) ⇒
(0 3 -2)

• Actually, the built-in map can take more than two arguments:

(map cons '(a b c) '((1) (2) (3))) ⇒
(((a 1) (b 2) (c 3))
What's Wrong Here??

1 ]=>
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (+ (map atomcount s)))
  ))
;Value: atomcount
1 ]=> (atomcount '(a b))

;The object (1 1), passed as an argument
;to +, is not the correct type.
...
2 error>

Why doesn't this work?

Limitations of Using eval

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

1 ]=> (+ 1 2 3)
;Value: 6

1 ]=> (cons '+ '(1 2 3))
;Value 12: (+ 1 2 3)

1 ]=> (eval (cons '+ '(1 2 3)) '())
;Value: 6

Using eval to Correct the Problem

(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (eval
              (cons '+ (map atomcount s)) '()))
  ))
1 ]=> (atomcount '(a b))

;Value: 2
1 ]=> (atomcount '((1) (2 3 (4)) (((5)))))

;Value: 5

Using eval to Evaluate Expressions

1 ]=> (append '(a) '(b))
;Value 13: (a b)
1 ]=> (cons 'append '((a) (b)))
;Value 14: (append (a) (b))

1 ]=> (eval (cons 'append '((a) (b))) '())
;Unbound variable: b
...
1 ]=> (cons 'append '( '(a) '(b)))
;Value 15: (append (quote (a)) (quote (b)))

1 ]=> (eval
    (cons 'append '( '(a) '(b))) '())
;Value 16: (a b)

Too complicated!!
Applying Procedures with \texttt{apply}

\begin{verbatim}
1 ]=> (apply + '(1 2 3)) ;Value: 6
1 ]=> (apply append '((a) (b))) ;Value 5: (a b)

1 ]=>
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else
         (apply + (map atomcount s)))))

;Value: atomcount
1 ]=> (atomcount 'a (b) c)) ;Value: 3
\end{verbatim}

\textbf{Higher-order Procedures: reduce}

\begin{verbatim}
(define (reduce op 1 id)
  (if (null? 1)
    id
    (op (car 1)
      (reduce op (cdr 1) id))))
\end{verbatim}

A binary \rightarrow n-ary procedure.

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

\textbf{NOTE:} \texttt{reduce} is not equivalent to \texttt{apply}.

\textbf{Higher-order Procedures: reduce}

\begin{verbatim}
(reduce + '(1 2 3) 0) \Rightarrow 6:
(reduce + '(1 2 3) 0)
(+ 1 (reduce + '(2 3) 0))
(+ 1 (+ 2 (reduce + '(3) 0)))
(+ 1 (+ 2 (+ 3 (reduce + '() 0))))
(+ 1 (+ 2 (+ 3 0)))
\end{verbatim}

\textbf{Note}: (+ 1 2 3) \Rightarrow 6

\begin{verbatim}
(reduce / '(24 6 2) 1) \Rightarrow 8:
(reduce / '(24 6 2) 1)
(/ 24 (reduce / '(6 2) 1))
(/ 24 (/ 6 (reduce / '(2) 1)))
(/ 24 (/ 6 (/ 2 (reduce / '() 1))))
(/ 24 (/ 6 (/ 2 1)))
\end{verbatim}

\textbf{Note}: (/ 24 6 2) \Rightarrow 2

\textbf{Higher-order Procedures: reduce}

\begin{verbatim}
Given union, which takes two lists representing sets and returns their union:
1 ]=> (apply union '((1 3)(2 3 4))) ;Value 21: (1 2 3 4)
1 ]=> (apply union '((1 3)(2 3)(4 5))) ;The procedure #\text{[compound-procedure union]}
;has been called with 3 arguments;
;it requires exactly 2 arguments.
1 ]=> (reduce union '((1 3)(2 3)(4 5)) ')()
;Value 22: (1 2 3 4 5)
\end{verbatim}

\textbf{Question}: How would you have to change \texttt{reduce} to be able to take intersection as its function argument?