Datatypes

What do we need to implement a datatype in Scheme?

At a minimum we need 3 kinds of procedures:

a) Recognizers: Is this thing an object of type X?
b) Constructors: Create an object of type X.
c) Accessors: Get field Y from an object of type X.

Of course, if we will work with data functionally, we don't need procedures to *set* the fields of a data element.

If you are only working with one datatype you can probably afford to skip the recognizers; everything has the same type. For our interpreter project we will parse Scheme expressions into trees, and then evaluate the tree. Each kind of expression needs a different type of tree. For example an *if* expression has a condition and two branches. A *let* expression has a binding list and a body. A *cond* expression has a list of pairs where the first element of each pair is a condition and the second element is an expression to evaluate if the condition is true, and so forth. When evaluating the tree it is important to know what kind of expression a particular node represents, which will tell us what kinds of children the node has and what to do with them.

For example, suppose we want to create a set datatype. We will represent a set with elements 'a 'b 'c as a list: '(set a b c). The empty set will be '(set).

This leads almost immediately to a whole series of definitions:

Constructor:

```
(define makeSet (lambda args
        (cons 'set (removeDuplicates args))))
```

where (removeDuplicates lat) is a simple function that strips duplicate entries from lat.

Recognizer:

```
(define set? (lambda (x)
               (cond
                 [(list? x) (if (null? x) #f (eq? (car x) 'set))]
                 [else #f])))
```

Note that it is important that your recognizers not crash if they are applied to an argument that is very much not a set, such as an atom or a null list.

Accessors

```
(define members (lambda (x)
                  (cdr x)))
```

Then we might have a host of utility functions:

```
(define contains? (lambda (set a)
                    (member? a (members set))))

(define addElement( lambda (a set)
                   (if (contains? set a)
                              set
                              (apply makeSet (cons a (members set))))))
```

```
(define union (lambda (set1 set2)
                (foldr addElement set2 (members set1))))


(define intersection (lambda (set1 set2)
                       (foldr (lambda (x y) (if (contains? set2 x)
                                                (addElement x y)
                                                y))
                              (makeSet)
                              (members set1))))
```

```
(define subset? (lambda (set1 set2)
            (foldr (lambda (x y) (if (not y) #f (contains? set2 x)))
                #t
                (members set1))))

(define sameSet? (lambda (set1 set2)
             (and (subset? set1 set2) (subset? set2 set1))))
```