define-datatype

In object-oriented programming everything that isn't represented by the simple primitive types is represented by an object of a class. Classes have a lot of built-in functionality:

- constructors
- getters
- setters
- recognizers (e.g. Java's instance-of operator)

Because we have a hierarchy of classes, a class can contain objects with different data. For example, a class Obies might contain Students, who have gpas and transcripts and majors, and Faculty, who have salaries, departments and tenure schedules.

Classes have a lot of other functionality, like inheritance. Scheme isn't object-oriented and we don't want it to be. However, we do need more capable data than we get with basic Scheme. In particular, for parsing we want to be able to make various kinds of nodes that hold different sorts of data, and as we walk around a parse tree we need to be able to recognize what sort of node we have at any given point.

It is possible to get this by using "tagged" lists that say what sort of node is being represented. For example, we might have a node that is a list of three elements. The first is a tag to say this node represents a lambda expression. The second is a list of symbols that are the parameters of the expression. The third is an expression that represents the body of the expression.

This can be made to work, but the resulting trees are difficult to visualize and difficult to debug.

A number of people have added datatype mechanisms to Scheme. We are going to use one that was developed in the book Essentials of Programming Languages by Friedman, Wand and Haynes. I will occasionally refer to this as EOPL.

You can access the full text of EOPL online through the library. We will only be working with chapters 2 and 3 and I will try to do everything in class that you need from this.

To use the EOPL system in Racket you need to put the following line at the top of each Scheme file:

```
(require (lib "eopl.ss"  "eopl"))
```

The feature we will use is called <u>define-datatype</u>

Here is an example that constructs binary trees.

```
(define-datatype binaryTree binaryTree?
          (null-node)
          (leaf-node (datum number?))
          (interior-node (key symbol?)
                         (left-child binaryTree?)
                         (right-child binaryTree?)))
```

This makes a datatype called "binaryTree" and a predicate binaryTree? that takes any Scheme expression and says if it was constructed using this define-datatype version of binary trees.

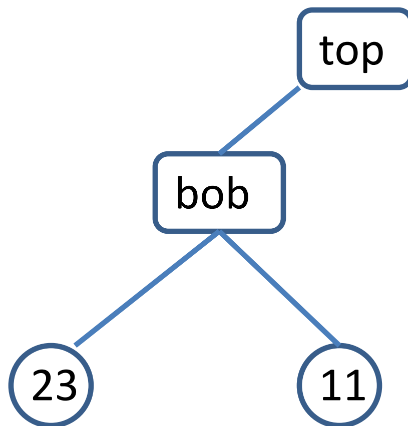
There are three different kinds of binaryTrees:

- null-nodes, which contain no data
- leaf-nodes, which contain numbers
- interior-nodes, which contain symbols and two child subtrees which must themselves be binaryTrees.

define-datatype creates a constructor procedure for each kind of binaryTree. For example, (leaf-node 5) creates a leaf-node object with datum 25.

Here is a short tree built from these functions:

```
(define T1 (leaf-node 23))
(define T2 (leaf-node 11))
(define T3 (interior-node 'bob T1 T2))
(define T4 (interior-node 'top T3 (null-node)))
```



There is also a cases expression that gives you access to the variant fields of an object constructed with define-datatype. The format of this is

```
(cases  type  object
        (variant1 (data fields) exp1)
        (variant2 (data fields) exp2)
        etc. )
```

For example, we could write a function that sums the leaf-nodes of our binaryTrees with

```
(define sum (lambda (tree)
    (cases binaryTree tree
        (null-node () 0)
        (leaf-node (v) v)
        (interior-node (sym left right) (+ (sum left) (sum right))))))
```

Note that we get for free a predicate of the top-level datatype but not for the variant types. The variant-type predicaters are easy to build if you need them:

```
(define leaf-node? (lambda (x)
                (cond
                    [(not (binaryTree? x)) #f]
                    [else (cases binaryTree x
                                  (null-node () #f)
                                  (leaf-node (v) #t)
                                  (interior-node (a b c) #f))])))
```

Note that we can only construct objects of the variant types, not of the top-level type. As we have defined it, we can construct a null-node, a leaf-node and an interior-node, which are the three kinds of binaryTrees, but we can't construct a generic binaryTree. The top-level datatypes are like abstract types in Java; they provide a unifying container for all of the variants but you can't construct one of them.