

Representing Data Abstractions

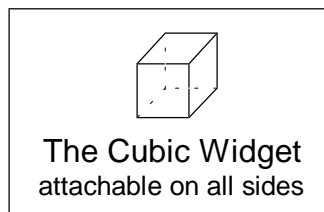
What is a data abstraction?

A data abstraction is a condition where higher level interfaces are used to interact with a lower level data structure. This can be related to the difference between a high-level language versus a low-level language. The high-level language has many predefined data structures and methods of interacting with them, whereas a low-level language only gives you the most basic forms of data structures. The interfaces allow the high-level user to use the data structure without needing to know or care about the underlying low-level data structures that make it up. All the user needs to know is how the given interfaces work, and the assurance that they will always work as specified with the correct data.

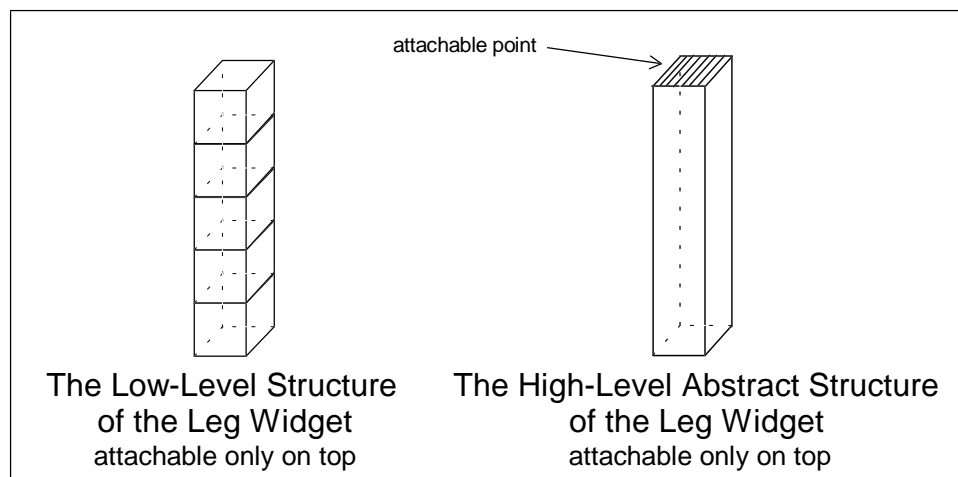
Representing a data abstraction.

Data abstractions are made so we don't have to worry about the low-level data structures that make up our high-level data structure. Likewise, when representing these high-level data structures by pictures or words, we don't want to draw it in respect to its low-level components.

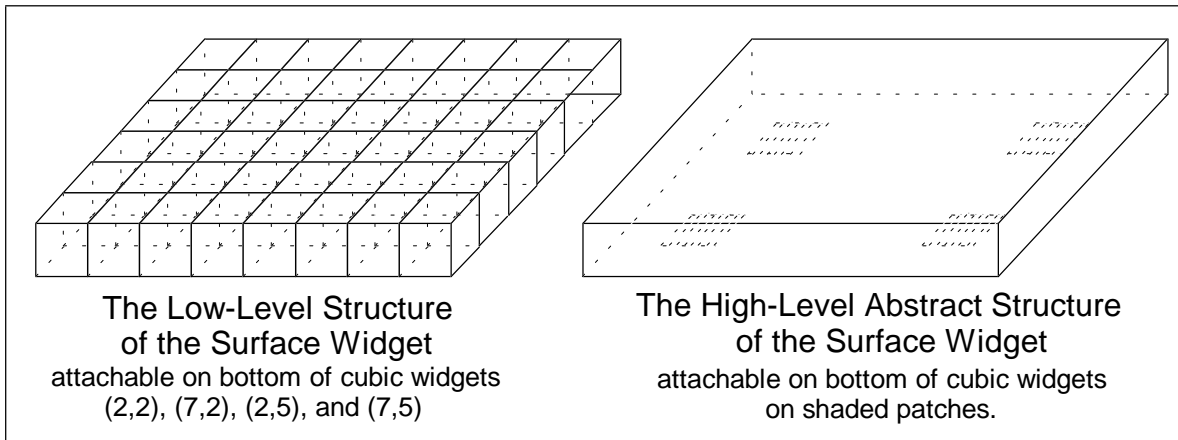
An example of this picture abstraction would be the manufacturing of widgets for industrial purposes. Widgets are created by combining multiple widgets to perform particular tasks and functions. One basic widget is a cubic block that can be connected to other widgets on any of its six sides. A pictorial representation is shown below.



Now we want to build a new object using the cubic widget. Our goal is to make a *leg* widget created by stacking 5 widgets on top of each other. The leg is also defined to have only one attachable surface on the top. The low-level and high-level abstractions are given below.

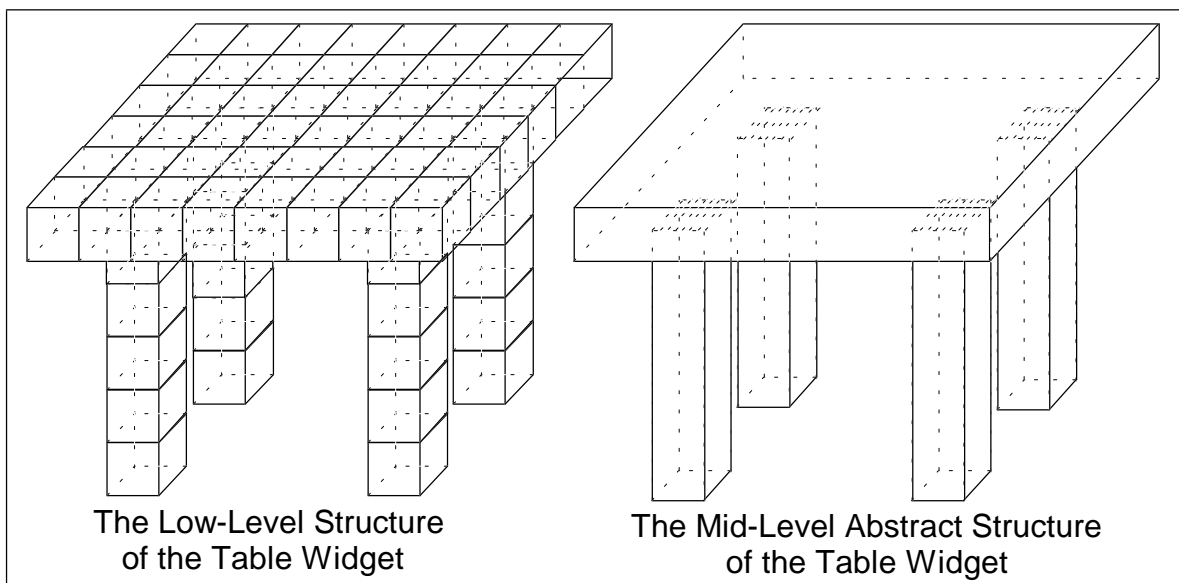


We'd also like to build a surface using the cubic widgets. This surface would be 1 deep, 6 wide, and 8 long with four attachment points on its bottom, on the cubic widgets at (2,2), (7,2), (2,5), and (7,5). The low-level and high level abstractions are given below.

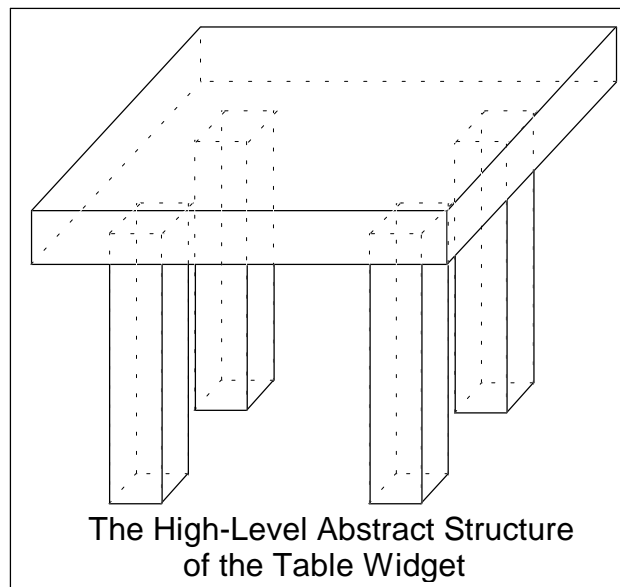


Both of these objects have high-level and low-level pictorial representations. Notice how the high-level pictures show all the information we need to know about the objects, while the low-level pictures look confusing and actually show us more than we really care about. The low-level pictures may also tempt us to misuse our new widgets. Recall that the basic widget is attachable to other widgets on all sides, whereas our new widgets only have particular attachment points. If we tried attaching blocks to any location on our new widgets other than the specified points, we would be violating our abstraction. Just looking at the low-level model seems to suggest you can do it, and you can, but only if you use the new widgets improperly (in the real world, this can cause things to break). This is why we always should distinguish new abstractions by drawing pictorial representations that clearly show the characteristics of the abstraction rather than the lower-level elements of the abstraction.

Using the proper attachment points we can make a table out of the new widgets. Notice how straight forward it is to build the table using the high-level abstract pictures, while far less obvious given the low-level pictures.



While we are on the topic of widget abstraction, you should also realize that we can make a high-level abstract picture for the table widget. Notice how it's cleaner representing the table at the highest level of abstraction. The table has no attachment points.



Scheme's Basic Widgets

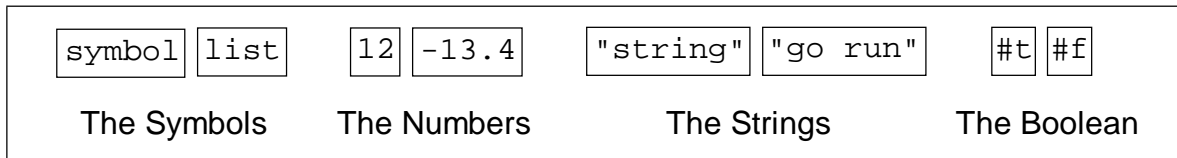
Common Characteristics

Scheme widgets (also known as data types) are similar to the widgets described above. Some of them are not constructed to point to other Scheme widgets and are unchangeable. These are sometimes called *atoms*. Others are constructed to point to other Scheme widgets and can be changed by redirecting their pointers (called mutation)¹. The function of pointers in Scheme is the same as someone actively keeping an eye on someone else. If we asked that person who they were watching, they could tell us. The person being watched may not know who's *pointing* at him unless he was explicitly told to *point* back. Also notice that multiple people can all point at the same person. Likewise, multiple Scheme widgets can point at a single Scheme widget.

¹ If mutation hasn't been covered yet, don't worry, just skip the parts you don't understand. Please do not use mutators in class unless they have been formally introduced by the reading assignments or lecture.

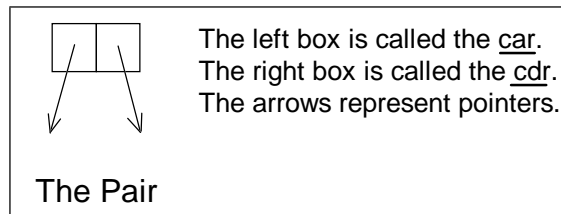
Symbols, Numbers, Strings, Boolean

Symbols, numbers, strings, and boolean values are a few² of Scheme's widgets that cannot point at any other Scheme widget, and are unchangeable. They have specific values and are considered to be atomic in nature. They normally appear as shown below. It isn't required that you put them in boxes, but that helps give them the appearance of being one whole widget, rather than many.



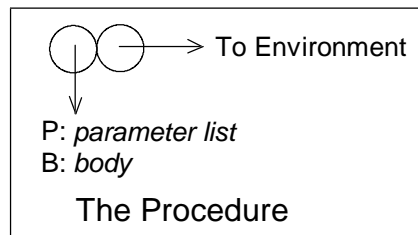
The Pair

One of Scheme's widgets that has pointers is called a *pair*. The pair has two parts that point to other Scheme widgets. The first pointer is called the *car* of the pair, and the second pointer is called the *cdr* of the pair. The pictorial representation of a pair is formed by making two boxes and putting them together. The pictorial representation of the pointers are arrows protruding from the center of their respective box.



The Procedure

One of Scheme's most complex widgets is the procedure. Not only are they constructed by one of Scheme's special forms, but they themselves are constructors of other special Scheme widgets called *frames*. Frames are beyond the scope of this document and will be covered at the same time as environment diagrams. The pictorial representation of a procedure is formed by putting two circles together (also known as a double bubble), each with a pointer pointing outwards. One of the circles always points to the parameters and body of the procedure, while the other points to an environment with the created frame³.



² There are more of them, but these are the ones you'll probably see most often.

³ When ignoring the frames and environments, and just using substitution, it's only necessary to worry about the *parameter list* and *body* of the procedure.

Constructors

Common Characteristics

In order to correctly create widgets, machines must be built to make them all compatible with the same interfaces. The constructor is made to create the new data based on the blue prints of the respective data type. If a constructor wasn't made, the data would have to be hand assembled, which simply looks messy (imagine the task of assembling the table out of the cubic widgets rather than ordering it out of a catalog) and is prone to error.

Numbers, Strings, Boolean

Numbers, strings, and boolean are the simplest widgets to create in Scheme. You type them in and they are created instantaneously. An unevaluated number is a number. An unevaluated string is a string. An unevaluated boolean is a boolean. When you evaluate one of these, they have the unique property of evaluating to themselves (similar to reconstructing themselves), so we call them self-evaluating.

$12.3 \Rightarrow 12.3$	$'12.3 \Rightarrow 12.3$	$(\text{quote } 12.3) \Rightarrow 12.3$
$"abc" \Rightarrow "abc"$	$'"abc" \Rightarrow "abc"$	$(\text{quote } "abc") \Rightarrow "abc"$
$\#t \Rightarrow \#t$	$'\#t \Rightarrow \#t$	$(\text{quote } \#t) \Rightarrow \#t$

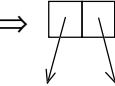
Symbols

Symbols are tricky widgets to make because you must insure that they are unevaluated. An evaluated symbol is immediately replaced with the value it is bound to within the current environment⁴. The unevaluated symbol stays a symbol, so that is how they are constructed.

$\text{symbol} \Rightarrow \text{whatever } \text{symbol} \text{ is bound to in the current environment}$
 $'\text{symbol} \Rightarrow \text{symbol}$
 $(\text{quote } \text{symbol}) \Rightarrow \text{symbol}$

The Pair

The pair's constructor has the appearance of most user defined constructors. This is because the constructor is a procedure rather than a special form. The symbol constructor described above (the quote) is a special form, and the procedure constructor described below is a special form. The pair's constructor is bound to the symbol *cons* in the global environment⁵.

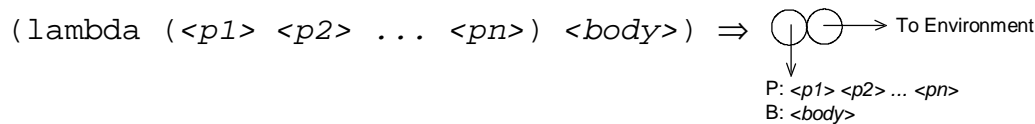
$(\text{cons } \langle \text{car} \rangle \langle \text{cdr} \rangle) \Rightarrow$ 
 $\langle \text{car} \rangle \quad \langle \text{cdr} \rangle$

⁴ This return value gives it the appearance of being similar to a pair because it seems to point at another widget, but it's not that simple of a relationship. We treat this occurrence as a special case. The evaluation of a symbol more closely resembles the operation of a *selector* for values within the *current environment* (a group of linked frames).

⁵ The global environment is where all Scheme primitive procedures and constants are defined.

The Procedure

The procedure is constructed with a special form to specify the parameters and body. The environment to which the procedure points is determined by when and where the procedure is created during evaluation⁶. The special form is detected by the first symbol in the evaluated list being *lambda*. $\langle p1 \rangle$ through $\langle pn \rangle$ are all symbols making up the formal parameters, and $\langle body \rangle$ is a list of unevaluated expressions (with possible *define* expressions at the top).



Selectors

Common Characteristics

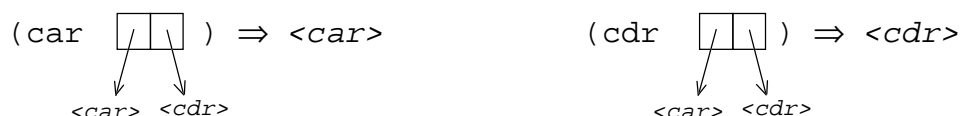
Selectors are special devices used to retrieve other widgets from widgets. For Scheme widgets, this can be done by returning a widget pointed to by the widget of the selector. However, what a selector returns doesn't necessarily have to be immediately pointed to by the widget, but the widget could be used to find or create the new widget. Allowing selectors to create new widgets gives flexibility to the low-level structure of widgets because as long as the selectors return the correct value, the underlying structure of the data doesn't matter. Selectors are designed to retrieve a widget from a single widget, so they traditionally take as a parameter the single widget and whatever extra information is needed to determine what is sought.

Symbols, Numbers, Strings, Boolean

Not all widgets *need* selectors. Symbols⁷, numbers, strings⁸, and boolean don't point to any widgets and represent themselves. This makes them end-of-the-line type widgets because you can use them to identify what to select in other widgets and have them returned by other widgets, but they don't need information selected from themselves to be useful. They can be used at face value.

The Pair

Most user defined selectors tend to be modeled similar to the pair's. The pair uses two procedures, one that returns the widget in the *car* of the pair, and the other that returns the widget in the *cdr* of the pair. The procedures are defined in the global environment as *car* and *cdr* respectively. They each take one parameter, a pair, because that's all they need to return the correct values.



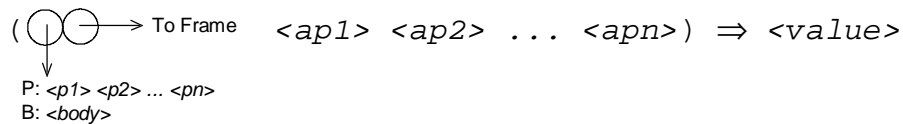
⁶ When covering the environment diagram, the frame the procedure points to is discovered to be the current environment of evaluation.

⁷ See footnote 4 about why symbols don't point to widgets.

⁸ Strings actually have *selectors* for substrings, but these aren't necessary for the using them.

The Procedure⁹

Procedures also have their own type of selector. Calling a procedure with specific actual parameters can be thought as specifying what needs to be returned by the procedure. By the general definition that I've given for selectors¹⁰, the procedure would appear to have a type of selector operation. This operation is built into the interpreter and is invoked whenever the procedure is applied to actual parameters $\langle ap1 \rangle$ through $\langle apn \rangle$.¹¹



Predicates

Common Characteristics

Predicates are useful when you want to find out some information about your widgets. They always return boolean values¹². For Scheme predicates, there is a convention of appending a ? to the end of the representing symbols. This reminds us that the symbol is representing something that is going to evaluate to a #t or #f value.

Equality Predicates

A common question about two objects are their equality. This becomes a complicated issue in Scheme. They have three predicates for testing equality bound to the symbols *equal?*, *eqv?*, and *eq?* in the global environment. *Equal?* is not picky and if things look the same, it will return #t¹³. *Eqv?* is pickier than *Equal?*, and *Eq?* is the pickiest of the three. The best place to find out how these predicates determine equality is by looking in the *Revised Report on the Algorithmic Language Scheme*.

⁹ Using the definition that I've given for selectors you could say procedures have selectors, but I've never heard of calling a procedure as a type of selector before (except in *Message Passing*).

¹⁰ I haven't checked to see if my definition is flawless, but it would seem to fit with the discussion of selectors in *SICP*.

¹¹ Remember that procedures can also have *no* parameters.

¹² Since Scheme assumes anything not #f is #t, they sometimes return other values other than #t.

¹³ None of the equality predicates are required to tell if two procedures look the same or are the same, so their use with procedures is undefined.

Symbols, Numbers, Strings, Boolean, Pairs, Procedures

There are numerous predicates for numbers and strings which can be found in the *Revised Report on the Algorithmic Language Scheme*, but the most popular predicates are probably the ones that tell you what type of widget you are dealing with. They consist of the name of the data type followed by a question mark.

```
(symbol? <widget>) ⇒ #t if <widget> is a symbol, else #f
(number? <widget>) ⇒ #t if <widget> is a number, else #f
(string? <widget>) ⇒ #t if <widget> is a string, else #f
(boolean? <widget>) ⇒ #t if <widget> is boolean, else #f
(pair? <widget>) ⇒ #t if <widget> is a pair, else #f
(procedure? <widget>) ⇒ #t if <widget> is a procedure, else #f
```

Constants

Common Characteristics

Constants are widgets that have a known structure. They are most often used as special widgets that have special meanings for a particular type of widget. Symbols, numbers, strings, and boolean are all constants because they have values that will never change. There are no constant pairs or procedures that can be counted on to always have the same value¹⁴.

Mutators

Common Characteristics

Mutators modify the value of a particular widget. This usually entails redirecting a pointer from pointing at one widget to another widget. For Scheme mutators, there is a convention of appending an *!* to the end of the representing symbols¹⁵.

Symbols, Numbers, Strings, Boolean

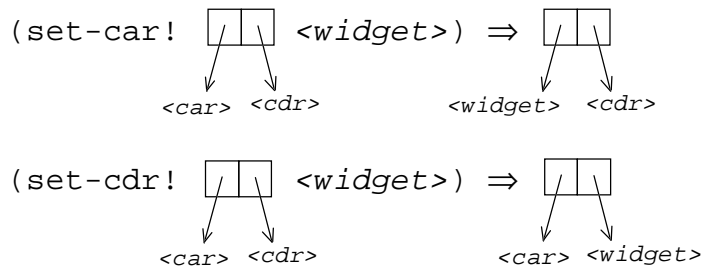
Symbols, numbers, strings, and boolean lack mutators. They are an example of unmutable Scheme widgets. They don't have any pointers to change and are also constants as described above.

¹⁴Someone told me that there is a null procedure () that produces the empty list, but I've also read that all expressions in parenthesis had to have at least one element to be without error (*Revised⁴ Report on the Algorithmic Language Scheme*, section 4.1.3. "Procedure Calls").

¹⁵The exclamation point, *!*, is usually pronounced "BANG!".

The Pair

The pair is a mutable widget. It has two pointers that can be shifted about to point at different widgets. The primitive procedures that mutate the pointers are bound to the symbols *set-car!* and *set-cdr!* in the global environment. *Set-car!* changes the car pointer and *set-cdr!* changes the cdr pointer.



The Procedure

A procedure's formal parameters and body, and the pointer to the frame cannot be mutated, which means that the procedure cannot be directly mutated. The procedure has mutable state by changing the values in the environment to which it points. In fact anything that changes a value bound to a symbol within a procedure's connected environment will change the effective data set the procedure uses (because a procedure doesn't have to use all of the symbols in its environment, there is a possibility this won't change how it evaluates).

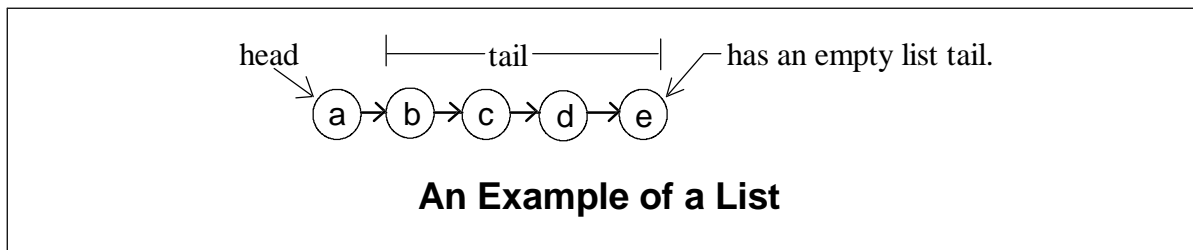
This state is changed by mutating the environment which is done through either the *define* or *set!* special forms. The *define* special form only binds and re-binds symbols in the current frame. It also has many limitations on where it can be used in your program¹⁶. The *set!* special form only re-binds symbols in the current environment. It can be used anywhere a normal expression can be used. Because I'm not covering the frame widgets, I can't show a pictorial example of how *define* and *set!* work.

¹⁶ The limitations of the *define* special form seem more restrictive than necessary, yet they are there (*Revised⁴ Report on the Algorithmic Language Scheme*, Section 5.2. "Definitions").

Scheme List¹⁷ Example

The Abstract List

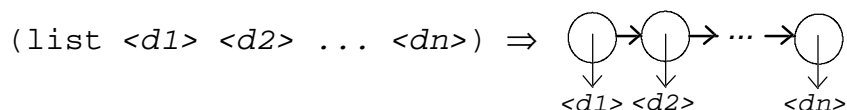
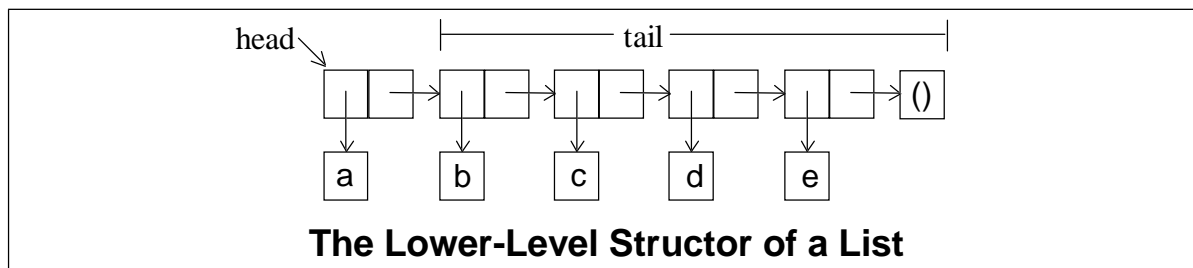
When coming up with a higher level abstraction, first you must make a picture to represent your new data type (widget). Lists consist of two parts, a *head* and a *tail*. The *head* contains a piece of datum and the *tail* is a sublist containing more data. An *empty list* is a list without any data or sublists. Now let's draw our picture¹⁸.



After drawing our picture, we are now ready to make the list widget out of Scheme's basic widgets. We will be developing the list by imitating the different interfaces described above.

The Constructor

There needs to be a way to create our list. Scheme makes this easy by providing a primitive procedure to do this for us. It takes any number of arguments, so we can make lists of any sizes. This provided constructor is bound to the symbol *list* in the global environment. The Scheme list is constructed using the pair widget because it has two pointers, one to point at the datum, and the other at the tail. The last tail in the list, `()`, is a Scheme invention called the *empty list*. It's declared to not be a pair, but still a list. It's not a symbol or a boolean value.



¹⁷The normal definition of a list is a sequence of elements. Scheme uses a special type of list called a linked-list. Lists do not have to be linked so that the *head* points to the *tail*.

¹⁸The pictures I draw are not special, they are only examples of how I imagine the data types *could* be drawn. Notice that I don't point at the datum widgets using pointers, but put them inside the *heads*. This is to make the diagram simpler, not because it is the best representation.

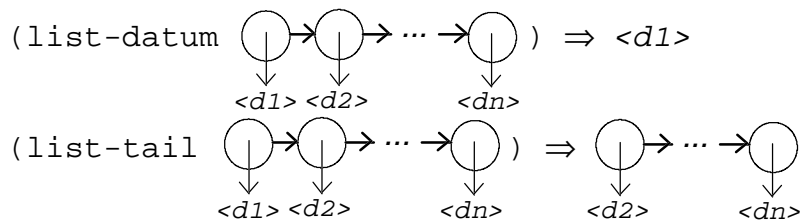
The Constants

The constant for lists is the *empty list*. It cannot be changed and is always the same widget that represents the *empty list*. I choose to draw an empty list as a circle without any datum pointer or tail pointer, because then it resembles a list with the properties of an empty list.

The Selectors

In order to get at the datum and tail at the head of the list, we need selectors. Scheme's implementation uses crude selectors, namely *car* and *cdr*. The selector *car*, which normally gets the left pointer in a pair, now returns the datum, and *cdr*, which normally gets the right pointer in a pair, now returns the tail. This choice of using the low-level selectors to interface with a list, rather than making high-level selectors, is known as a data abstraction violation. To make a list more kosher, I suggest defining new selectors in the global environment, *list-datum* and *list-tail*.

```
(define (list-datum list) (car list))
(define (list-tail list) (cdr list))
```



The Predicates

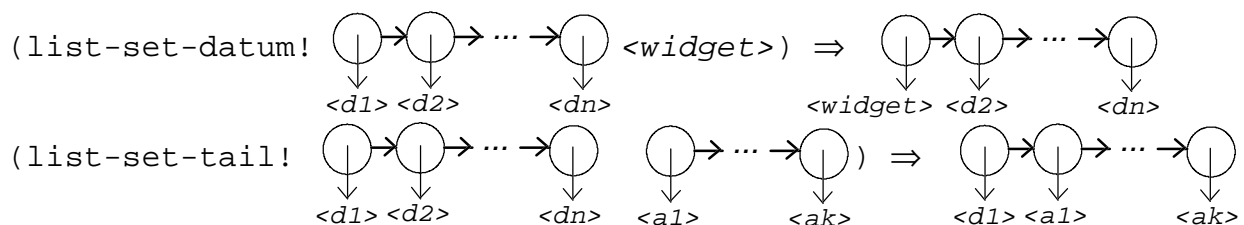
Scheme provides two general predicates for use with lists. They are bound to the symbols *list?* and *null?*. *List?* returns #t if the given widget is a list. *Null?* returns #t if the given widget is an empty list.

```
(list? <widget>) => #t if <widget> is a list, else #f
(null? <widget>) => #t if <widget> is an empty list, else #f
```

The Mutators

The mutators for lists are *set-car!* and *set-cdr!*, yet another data abstraction violation. To remedy this, I suggest defining more abstract mutators in the global environment (just as we did with the selectors) *list-set-datum!* and *list-set-tail!*.

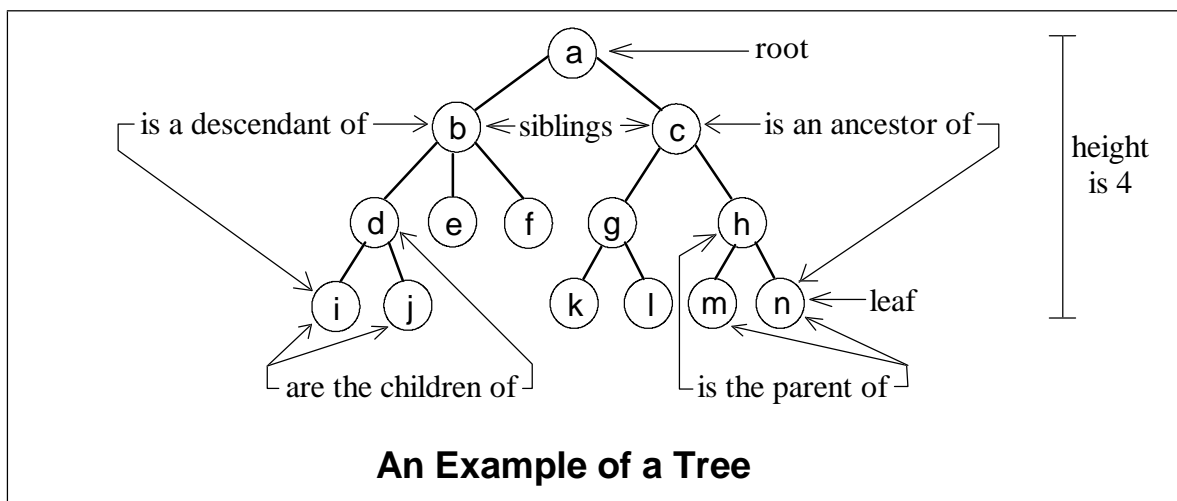
```
(define (list-set-datum! list widget) (set-car! list widget))
(define (list-set-tail! list widget) (set-cdr! list widget))
```



General Tree Example

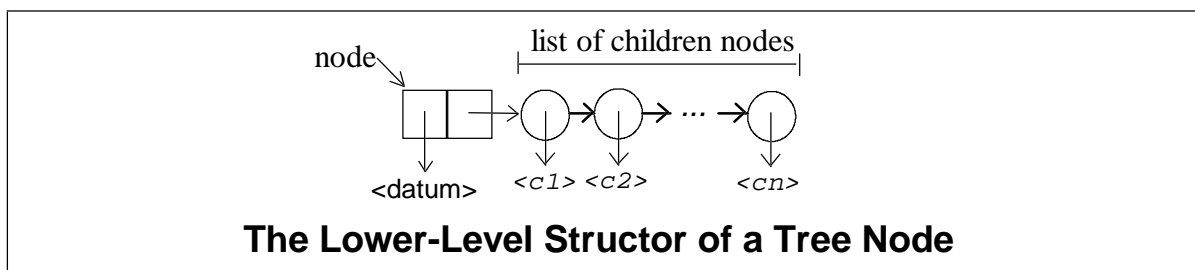
The Abstract Tree

Because we are developing a higher level abstraction, we must once again make a picture to represent our new data type (widget). Trees consist of nodes connected together in a hierarchical fashion with a *parent* node directly above *children* nodes. At the very top of the tree is one node, called the *root* which has no parents. At the bottom of the tree are nodes without children called *leaves*¹⁹. The *height* of a tree is the longest path of nodes from the root to a leaf. *Siblings* are nodes who share the same parent node. A node *a* that can be reached from following a path starting at node *b* going to the root is called an *ancestor* of *b*. A node *a* that can be reached from following a path start at node *b* going to a leaf is called a *descendant* of *b*. A tree is made up of *subtrees* that are connected to the root. Now let's draw our picture.



The Constructor

After drawing our picture, we are now ready to make a tree widget out of Scheme's basic widgets. Looking at the picture, we see that the tree is basically built from nodes which all share the same characteristics of having parents and children. To make our tree simpler, I have chosen to design it so the nodes only have pointers to their children, not to their parent. Each node also must point to a datum widget. What would allow us to have one pointer to the datum and multiple pointers for all the possible children? One possibility is a pair whose car points at the datum and whose cdr points at a list of children.



¹⁹Choosing to mix normal trees and family trees makes nomenclature entertaining.

Notice how I purposely avoided drawing the list as a bunch of pairs. Not only would that representation have been a data abstraction violation since I said I was going to use a list, but it would have looked like I was using a bunch of pairs instead of a list. Now it should be fairly straight forward to write the constructor for a tree²⁰.

```
(define (make-tree datum . children)
  (cons datum children))
```

Another constructor could take the datum and a list of children. This may seem like a lazy way of making a constructor because we just put the list in the pair, but notice that if we decided to use something other than a list, we could easily convert that list into something else. We are also not violating any data abstractions because we explicitly tell the user that it takes a *list of children nodes* as input and not something different.

```
(define (make-tree2 datum list-of-children)
  (cons datum list-of-children))
```

The Constants

We need a constant to represent an *empty tree*, just like we needed a constant for lists which represented an empty list. Once we define what this constant is, it should never change, and it'll always represent an empty tree. A picture of an empty tree should not be the same as a list because a list is not a tree. To differentiate between the two, I suggest that an empty tree be drawn as a circle with a "T" drawn in it. An empty tree node cannot be a leaf because it doesn't perform the function of a real node having a datum.



The Abstract Empty Tree Node

```
(define empty-tree 'the-empty-tree)
```

²⁰I use the '.' in the formal parameter list to indicate that I have a variable number of values after that point. All of those values are placed in a list which works nicely with our lower-level data structure.

The Selectors

There are various selectors we can write for our tree. One is for getting the datum and the other the children or a child. I'll define three selectors, *tree-datum*, *tree-children-list*, and *tree-child*. *Tree-datum* will return the data of the current tree's root node. *Tree-children-list* will return a list of all the children of current tree's root node. *Tree-child* returns a child of the tree's root node given an index number which searches the children going from left to right. It also returns an empty tree node if no child is found at the given index.

```
(define (tree-datum tree) (car tree))
(define (tree-children-list tree) (cdr tree))
(define (tree-child tree index)
  (define (index-list i list-of-trees)
    (cond ((null? list-of-trees) empty-tree)
          ((= i 0) (list-datum list-of-trees))
          (else (index-list (- i 1) (list-tail list-of-trees)))))
  (index-list index (tree-children-list tree)))
```

The Predicates

There are also a few predicates needed to help finish the abstract interface to our tree widget. We can make a *tree?* predicate which parallels with Scheme's *list?* predicate, and an *empty-tree?* predicate which parallels with Scheme's *null?* predicate. The *tree?* predicate should make sure the structure of the widget is the same as a tree. If it isn't then it's not a tree. The *empty-tree?* just checks to see if the widget is the empty tree constant.

```
(define (tree? widget)
  (define (list-of-trees? list-of-widgets)
    (if (null? list-of-widgets) #t
        (and (tree? (list-datum list-of-widgets))
              (list-of-trees? (list-tail list-of-widgets)))))
  (cond ((empty-tree? widget) #t)
        ((not (pair? widget)) #f)
        ((not (list? (cdr widget))) #f)
        (else (list-of-trees? (cdr widget)))))
(define (empty-tree? widget) (equal? widget empty-tree))
```

The Mutators

Now we want ways to mutate the parts of our tree. To keep the mutators simple, I suggest making a mutator that makes the node point to a different datum widget, and one to change the children. These will closely resemble the constructors. *Tree-datum!* will change the datum of the node. *Tree-children!* will accept a variable number of children as an argument and change the children to be those given. *Tree-children-list!* will accept a list of children and make all of the children in the node be those in the list.

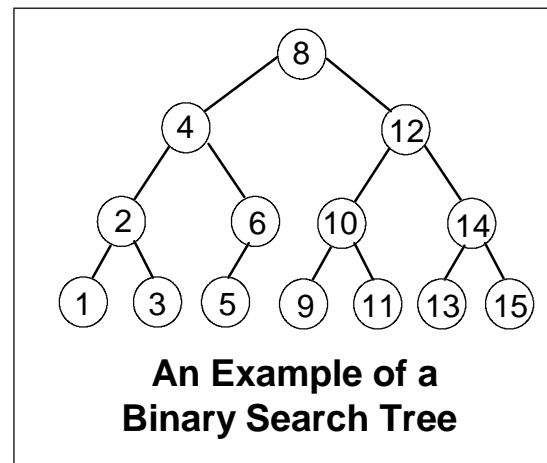
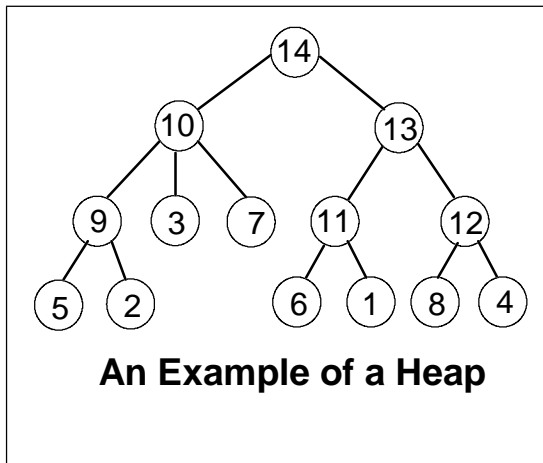
```
(define (tree-datum! tree datum) (set-car! tree datum))
(define (tree-children! tree . children)
  (set-cdr! tree children))
(define (tree-children-list! tree list-of-children)
  (set-cdr! tree list-of-children))
```


The Tree Hierarchy

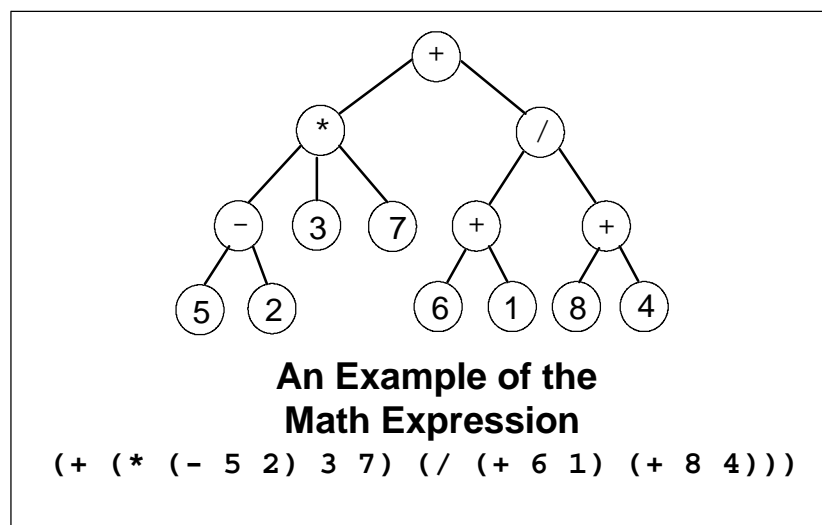
What Trees Represent

Trees are commonly used to represent a hierarchy of information. The hierarchy is often used to take advantage of classifying items by starting at a base followed by each subtree being more specific in meaning than its parent. An example would be the classification of living organisms. Starting at the top you have the broad class of living organisms. As you move down, the organisms are continuously subdivided into more and more groups.

Another type of hierarchy resembles something that's in a particular order in respect to the root node. Two examples of this are the *heap* and *binary search trees*. The *heap* is composed of each parent of the tree having a number of greater value than its children. The parent nodes of a *binary search tree* all having descendants with datums less in value than its own datum down its left branch, and greater in value than its own datum down its right branch.

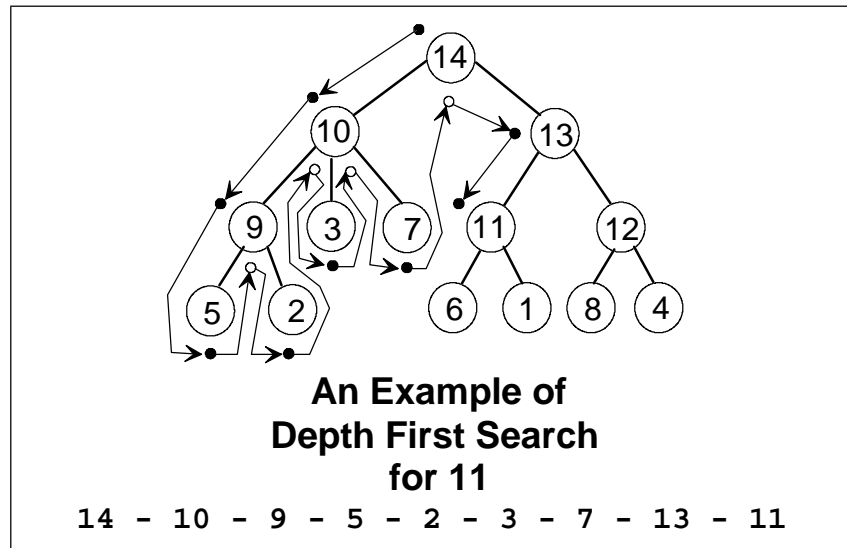


Trees can also be used to represent the order of operations. For example, the tree below could be used to perform a series of arithmetic operations to simplify a mathematical expression:

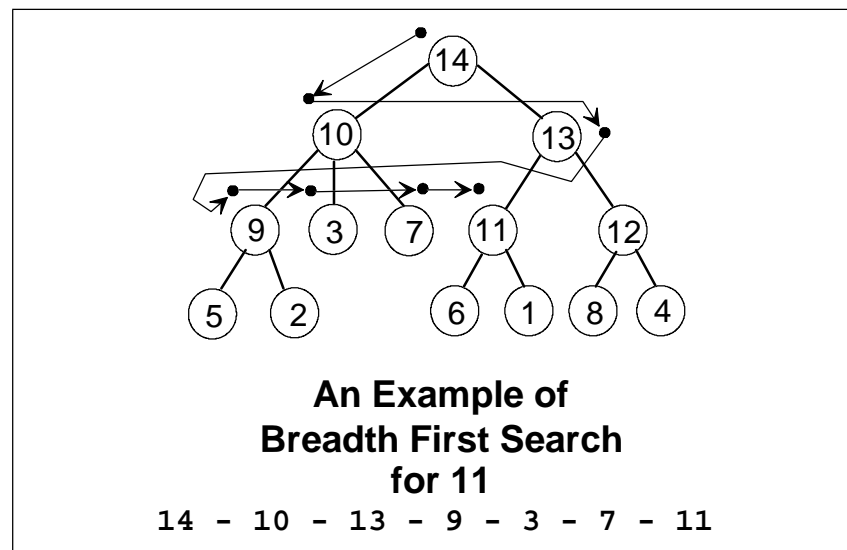


Searching Trees

There are two common methods of searching trees, the *depth-first search* and the *breadth-first search*. Depth-first search simply checks the current node and if it doesn't match the search requirements, chooses to descend down a branch to continue the search. If all the nodes in that branch have been exhausted, the search continues down a new branch until all of the branches have been searched. When nothing has been found in the tree, the search fails.



Breadth-first search searches all nodes at a particular height in the tree before continuing on to a lower depth.

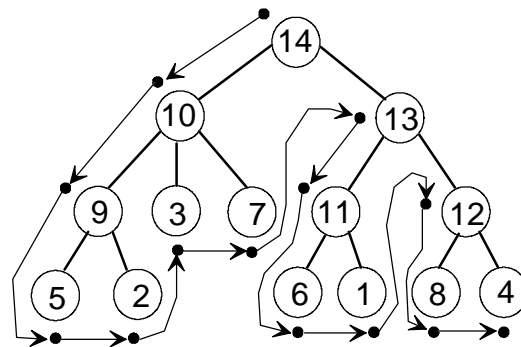


Operating on Trees

There are three general orders of operating on trees – *preorder*, *postorder*, and *inorder*. These describe the order in which the nodes are visited and operated upon. Their descriptions are as follows:

Preorder - Works on general trees.

1. Operate on root node.
2. Operate on subtrees.

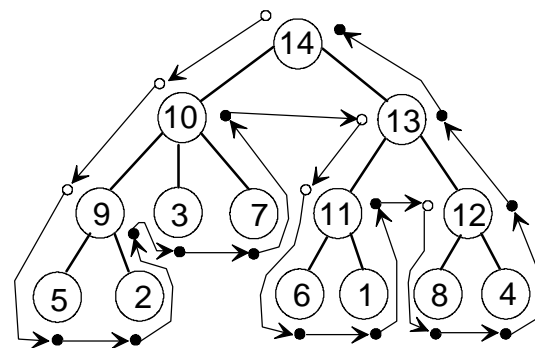


**An Example of
Preorder Operation**

14 - 10 - 9 - 5 - 2 - 3 - 7 - 13 - 11 - 6 - 1 - 12 - 8 - 4

Postorder - Works on general trees.

1. Operate on subtrees.
2. Operate on root node.

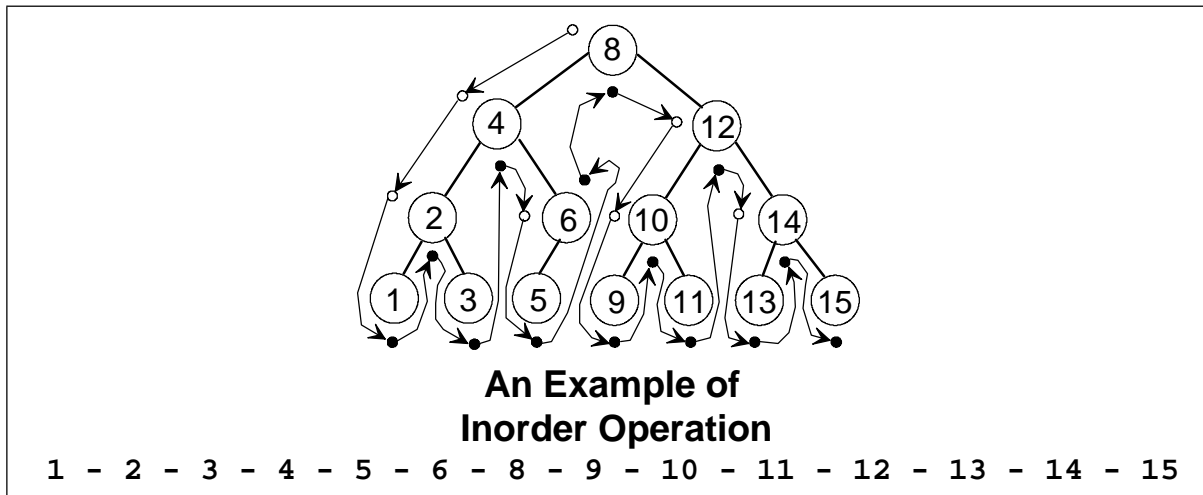


**An Example of
Postorder Operation**

5 - 2 - 9 - 3 - 7 - 10 - 6 - 1 - 11 - 8 - 4 - 12 - 13 - 14

Inorder - Works on binary trees.

1. Operate on left subtree.
2. Operate on root node.
3. Operate on right subtree.



As an example of each operation, I'll collect the node values into a list using each of the orders. Notice how I explicitly make sure the operations are done in the correct order by using *let* special forms.

```
(define (append-lists lists)
  (if (null? lists) '()
      (append (car lists) (append-lists (cdr lists)))))

(define (preorder tree)
  (if (empty-tree? tree) '()
      (let ((root (list (tree-datum tree)))
            (subtrees (map preorder (tree-children-list tree))))
        (append root (append-lists subtrees)))))

(define (postorder tree)
  (if (empty-tree? tree) '()
      (let ((subtrees (map postorder (tree-children-list tree)))
            (root (list (tree-datum tree))))
        (append (append-lists subtrees) root))))

(define (inorder binary-tree)
  (if (empty-tree? tree) '()
      (let ((left (inorder (tree-child binary-tree)))
            (root (list (tree-datum tree)))
            (right (inorder (tree-child binary-tree))))
        (append left root right)))))
```


Data Abstraction Techniques

Manifest Data Types

Data has a *manifest type* if its type can be clearly known and tested. Primitive Scheme widgets have manifest types. These are numbers, boolean, strings, symbols, procedures, and pairs. The other complex data types that we created, lists and trees, do not have manifest types, because they may *appear* to be the correct type, but we don't know for certain.

A method of creating our own manifest types is through data tagging. By tagging each type of data with a name, we can easily distinguish between types without accidentally confusing two types that *look* the same. To help us work with tags, we can create constructors to wrap data and selectors to obtain the tags and data.²¹

Constructor:

```
(define (attach-type type contents)
  (cons type contents))
```

Selectors:

```
(define (type datum)
  (if (pair? datum) (car datum) 'primitive))

(define (contents datum)
  (if (pair? datum) (cdr datum) datum))
```

To help this discussion of types, I'll redefine the tree constructor and predicate so that it becomes a manifest type, and make a new type of tree, a binary tree.

Binary Tree Constructor:

```
(define (make-btree datum left right)
  (attach-type 'btree (list datum left right)))
```

Binary Tree Constant:

```
(define empty-btree 'the-empty-btree)
```

Binary Tree Selectors:

```
(define (btree-datum btree) (car btree))
(define (btree-left btree) (cadr btree))
(define (btree-right btree) (caddr btree))
(define (btree-children-list btree) (cdr btree))
(define (btree-child btree index)
  (cond ((= index 0) (btree-left btree))
        ((= index 1) (btree-right btree))
        (else empty-btree)))
```

Binary Tree Predicates:

```
(define (btree? widget) (eq? (type widget) 'btree))
(define (empty-btree? widget) (equal? widget empty-btree))
```

²¹I've chosen to return type *primitive* for non-tagged types to make numbers, boolean, strings, and symbols easy to use with this new paradigm. SICP chose a better, more complex alternative.

General Tree Constructor:

```
(define (make-tree datum . children)
  (attach-type 'tree (cons datum children)))
```

General Tree Predicates:

```
(define (tree? widget) (eq? (type widget) 'tree))
(define (empty-tree? widget) (equal? widget empty-tree))
```

Explicit Dispatch

Explicit dispatch is where procedures are made to determine what type is being used and act accordingly. We have two types of trees, both with different types of selectors. It would be nice to have a general set of tools to treat both of them same. Here's how that would be done using explicit dispatch:

```
(define (get-datum datum)
  (let ((widget (contents datum)))
    (cond ((tree? datum) (tree-datum widget))
          ((btree? datum) (btree-datum widget)))))

(define (get-children-list datum)
  (let ((widget (contents datum)))
    (cond ((tree? datum) (tree-children-list widget))
          ((btree? datum) (btree-children-list widget)))))

(define (get-child datum index)
  (let ((widget (contents datum)))
    (cond ((tree? datum) (tree-child widget index))
          ((btree? datum) (btree-child widget index)))))
```

The main features of explicit dispatch are given below:

- Operators check type.
- Operators determine how to evaluate correctly.
- Code forced to be grouped by operator.

Data-Directed Programming

Data-directed is where the operators are stored in a 2-Dimensional table, organized by type, and a generic application procedure can correctly choose and carry out the proper action. For storing information in the table, we are introduced to two new Berkeley Scheme procedures *put* and *get*. Put is a mutator, and get is a selector. Their prototypes are as follows:

```
(put <type> <op> <item>) ⇒ An unspecified value.
(get <type> <op>) ⇒ The item stored at the given location in the table.
```


To apply the operators, we need to make a generic application procedure. It must take the operator, determine the type of datum being used, and apply the correct procedure. A simple version would be written as follows:

```
(define (apply-generic op . data)
  (let ((widget-types (map type data))
        (widgets (map contents data)))
    (let ((proc (get widget-types op)))
      (apply proc widgets)))))
```

The *apply-generic* procedure is made to work with procedures of multiple parameters. It handles this by using a list of the parameter types as the acceptable data type of the operator. The *apply* procedure used in the definition simply applies the given procedure to the list of actual parameters²². Now all we need to do is put the procedures in the correct locations within the table:

```
(put '(tree) 'get-datum tree-datum)
(put '(tree) 'get-children-list tree-children-list)
(put '(tree primitive) 'get-child tree-child)

(put '(btree) 'get-datum btree-datum)
(put '(btree) 'get-children-list btree-children-list)
(put '(btree primitive) 'get-child btree-child)
```

A curiosity about data-directed programming is that you aren't forced to group the definitions in any particular way. Above they are grouped by type, but we could also group them by operators.

```
(put '(tree) 'get-datum tree-datum)
(put '(btree) 'get-datum btree-datum)

(put '(tree) 'get-children-list tree-children-list)
(put '(btree) 'get-children-list btree-children-list)

(put '(tree primitive) 'get-child tree-child)
(put '(btree primitive) 'get-child btree-child)
```

The main features of data-directed programming are given below:

- Data determines type.
- Table used to choose correct operator based on actual parameter types.
- Code easily organized by type or operator.

²²Also known as arguments.

Message Passing

Message passing is where the operators are built into the data itself. The easiest way to represent message passing data types is by using procedures. To make the implementation more closely related to SICP, I'll also introduce a new way of making data have manifest type. First we'll change the constructors to create data which takes messages.²³

General Tree Constructor:

```
(define (make-tree datum . children)
  (lambda (op . params)
    (cond ((eq? op 'type) 'tree)
          ((eq? op 'empty?) #f)
          ((eq? op 'get-datum) datum)
          ((eq? op 'get-children-list) children)
          ((eq? op 'get-child)
           (let ((index (car params)))
             (if (>= index (length children))
                 empty-tree
                 (list-ref children (car params)))))))
```

Binary Tree Constructor:

```
(define (make-btree datum left right)
  (lambda (op . params)
    (cond ((eq? op 'type) 'btree)
          ((eq? op 'empty?) #f)
          ((eq? op 'get-datum) datum)
          ((eq? op 'get-children-list) (list left right))
          ((eq? op 'get-child)
           (let ((index (car params)))
             (cond ((= index 0) left)
                   ((= index 1) right)
                   (else empty-btree)))))))
```

Now we should also redefine the predicates. At the same time, defining constants which are procedures may be more helpful than the plain symbolic one.

General Tree Constant:

```
(define (empty-tree op)
  (cond ((eq? op 'type) 'tree)
        ((eq? op 'empty?) #t)))
```

General Tree Predicates:

```
(define (tree? widget)
  (and (procedure? widget) (eq? (widget 'type) 'tree)))

(define (empty-tree? widget)
  (and (procedure? widget) (widget 'empty?)))
```

²³I've chosen to use *list-ref* and *length* since they are Scheme primitives with the following prototypes:

```
(list-ref <list> <k>) ⇒ returns the kth element in list
(length <list>) ⇒ the number of elements in list
```


Binary Tree Constant:

```
(define (empty-btree op)
  (cond ((eq? op 'type) 'btree)
        ((eq? op 'empty?) #t)))
```

Binary Tree Predicates:

```
(define (btree? widget)
  (and (procedure? widget) (eq? (widget 'type) 'btree)))

(define (empty-btree? widget)
  (and (procedure? widget) (widget 'empty?)))
```

As a way to show that message passing data can be interfaced with similarly to the data-directed style, the *apply-generic* procedure can be redefined as:

```
(define (apply-generic op . data)
  (let ((widget (car data))
        (params (cdr data)))
    (apply widget (append (list widget) params))))
```

The main features of message passing are given below:

- Data determines type.
- Data has the correct operator and applies it for a correct result.
- Code forced to be grouped by type.