CS 61A Lecture Notes Week 5

Topic: Hierarchical data/Scheme interpreter

Reading: Abelson & Sussman, Section 2.2.2–2.2.3, 2.3.1, 2.3.3

Midterm 1 is this week.

• Example: A Calculator Program

Later in the course we'll be studying several variants of a Scheme interpreter written in Scheme. As a first, small step in that direction, here is an interactive calculator that accepts arithmetic expressions in Scheme notation, but without variables, without the ability to define procedures, and with no data types other than numbers. Here's how it works:

```
STk> (load "~cs61a/lib/calc.scm")
STk> (calc)
calc: (+ 2 3)
5
calc: (+ (* 2 3) (* 4 5))
26
calc: foo
Error: calc: bad expression: foo
```

The last example shows that there are no variables in this language.

The entire program consists of three procedures in 30 lines of code. You should find it easy to understand. And yet these three procedures exactly parallel the core procedures in a real Scheme interpreter:

1. The read-eval-print loop: interact with the user.

2. Eval: `(eval expression)` returns the value of the expression.

3. Apply: `(apply function argument-values)` calls the function and returns the value it returns.

Here's the read-eval-print loop (or REPL, pronounced "rep-uhl"):

```
(define (calc)
  (display "calc: ")
  (flush)
  (print (calc-eval (read)))
  (calc))
```

The calls to `display` and `flush` print the user prompt. (`Flush` is used when you want to print something that doesn't end with a newline character, but you want it to print right away. Ordinarily, most programs, including STk, save up the characters you print until you finish a line, then send the entire line to the operating system at once. `Flush` tells STk not to wait for the end of the line.)

The most important line is the `(print (calc-eval (read)))`. `Read` is a Scheme primitive that reads a datum from the keyboard. "Datum" here means a word, a list, or any other single Scheme value. In our case, the things we're reading are Scheme expressions, but `read` doesn't know that; as far as `read` is concerned, `(+ 2 3)` is just a list of three elements, not a request to add two numbers.

The fact that a Scheme expression is just a Scheme datum—a list—makes it very easy to write an interpreter. This is why Scheme uses the `(+ 2 3)` notation for function calls rather than, say, `+(2,3)` as in the usual mathematical notation for functions! Even a more complicated expression such as `(+ (* 2 3) (* 4 5))` is one single Scheme datum, a list, in which some of the elements are themselves lists.

What do we want to do with the thing `read` returns? We want to treat it as an expression in this Scheme-

subset language and *evaluate* it. That's the job of `calc-eval`. (We use the names `calc-eval` and `calc-apply` in this program because STk has primitive procedures called `eval` and `apply`, and we don't want to step on those names. The STk procedures have jobs exactly analogous to the ones in `calc`, though; every interpreter for any Lisp-family language has some form of `eval` and `apply`.)

Once we get the value from `eval`, what do we want to do with it? We want to show it to the user by printing it to the display. That's the job of `print`. So now you understand why it's a "read-eval-print" loop! Read an expression, evaluate it, and print its value.

Finally, the procedure ends with a recursive call to itself, so it loops forever; this is the "loop" part of the REPL.

Notice that `read` and `print` are not functional programming; `read` returns a different value each time it's called, and `print` changes something in the world instead of just returning a value. The body of the REPL has more than one expression; Scheme evaluates the expressions in order, and returns the value of the last expression. (In this case, though, it never returns a value at all, since the last expression is a recursive call and there's no base case to end the recursion.) In functional programming, it doesn't make sense to have more than one expression in a procedure body, since a function can return only one value.

(It's also worth noting, in passing, that the REPL is the `only` non-functional part of this program. Even though we're doing something interactive, functional programming techniques are still the best way to do most of the work of the calculator program.)

The job of `eval` is to turn expressions into values. It's very important to remember that those are two different things. Some people lose midterm exam points by thinking that when you type `'foo` into Scheme, it prints out `'foo`. Of course it really prints `foo` without the quotation mark. `Foo` is a possible Scheme value, but `'foo` really makes sense only as an expression. (Of course, as we've seen, every expression is also a possible value; what expression would you type to Scheme to make it print `'foo`?) What confuses people is that *some* things in Scheme are both expressions and values; a number is a Scheme expression whose value is the number itself. But most expressions have a value different from the expression itself.

Part of what makes one programming language different from another is what counts as an expression. In Scheme, certain kinds of expressions have funny notation rules, such as the extra parentheses around clauses in a `cond` expression. The notation used in a language is called its *syntax*. `Eval` is the part of a Lisp interpreter that knows about syntax. Our simplified calculator language has only two kinds of syntax: numbers, which are *self-evaluating* (i.e., the value is the number itself), and lists, which represent function calls:

```
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((list? exp) (calc-apply (car exp) (map calc-eval (cdr exp))))
        (else (error "Calc: bad expression:" exp))))
```

In real Scheme, there are more kinds of self-evaluating expressions, such as Booleans (`#t` and `#f`) and `"strings in double quotes"`; there are variables, which are expressions; and some lists are special forms instead of procedure calls. So in a Scheme interpreter, `eval` is a little more complicated, but not that much more.

By the way, notice that we're talking about two different programming languages here. I've said that the calculator language doesn't have variables, and yet in `calc-eval` we're using a variable named `exp`. This isn't a contradiction because `calc-eval` isn't itself a program in calculator-language; it's a program in STk, which is a complete Scheme interpreter. The calculator language and Scheme are different enough so that you probably won't be confused about this, but I'm belaboring the point because later on we'll see interpreters for much more complete subsets of Scheme, and you can easily get confused about whether some expression you're looking at is part of the interpreter, and therefore an STk expression, or data given to the interpreter, and therefore a mini-Scheme expression.

The way `calc-eval` handles function calls is the only part of the calculator that is *not* the same as the corresponding feature of real Scheme. That's because in calculator language, numbers are the only data type, and so in particular procedures aren't data. In a real Scheme procedure call expression, the first subexpression, the one whose value provides the procedure itself, has to be evaluated just as much as the argument subexpressions. *Often* the first subexpression is just a variable name, such as `+` or `cdr`, but the expression could be

```
((lambda (x) (+ x 5))  (* 2 3))
```

in which case the first subexpression is a special form, a `lambda` expression. But in calculator language, the first sub-"expression" is always the name of the function, and there are only four possibilities: `+`, `-`, `*`, and `/`. I put "expression" in quotes because these symbols are *not* expressions in calculator language. So the expression in `calc-eval` that handles procedure calls is

```
(calc-apply (car exp) (map calc-eval (cdr exp)))
```

The first argument to `calc-apply` is the *name* of the function we want to call. The rest of the expression `exp` consists of actual argument subexpressions, which we recursively evaluate by calling `calc-eval` for each of them. (Remember that `map` is the list version of `every`; it calls `calc-eval` repeatedly, once for each element of `(cdr exp)`.) When we look at a more-nearly-real Scheme interpreter in another week, the corresponding part of `eval` will look like this:

```
(apply (EVAL (car exp)) (map eval (cdr exp)))
```

`Eval` is the part of the interpreter that knows about the syntax of the language. By contrast, `apply` works entirely in the world of values; there are no expressions at all in the arguments to `apply`. Our version of `apply` has the four permitted operations built in:

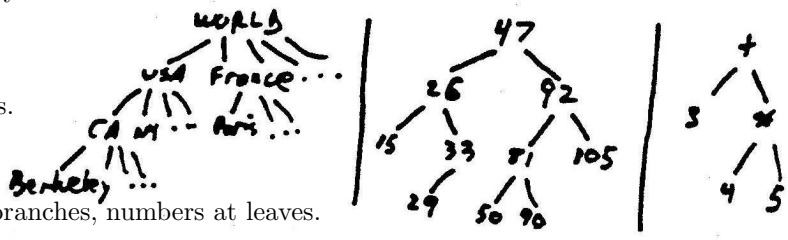
```
(define (calc-apply fn args)
  (cond ((eq? fn '+) (accumulate + 0 args))
        ((eq? fn '-) (cond ((null? args) (error "Calc: no args to -"))
                           ((= (length args) 1) (- (car args)))
                           (else (- (car args) (accumulate + 0 (cdr args))))))
        ((eq? fn '*) (accumulate * 1 args))
        ((eq? fn '/) (cond ((null? args) (error "Calc: no args to /"))
                           ((= (length args) 1) (/ (car args)))
                           (else (/ (car args) (accumulate * 1 (cdr args))))))
        (else (error "Calc: bad operator:" fn))))
```

The associative operations can be done with a single call to the higher order `accumulate` function; the non-associative ones have to deal with the special case of one argument separately. I'm not spending a lot of time on the details because this isn't how real Scheme handles function calls; the real `apply` takes the actual procedure as its first argument, so it doesn't have to have special knowledge of the operators built into itself. Also, the real `apply` handles user-defined procedures as well as the ones built into the language.

• Trees. Big idea: representing a hierarchy of information.

What are trees good for?

- Hierarchy: world, countries, states, cities.
- Ordering: binary search trees.
- Composition: arithmetic operations at branches, numbers at leaves.



The name "tree" comes from the branching structure of the pictures, like real trees in nature except that they're drawn with the root at the top and the leaves at the bottom.

A *node* is a point in the tree. In these pictures, each node includes a *datum* (the value shown at the node, such as `France` or `26`) but also includes the entire structure under that datum and connected to it, so the `France` node includes all the French cities, such as `Paris`. Therefore, **each node is itself a tree**—the terms "tree" and "node" mean the same thing! The reason we have two names for it is that we generally use "tree" when we mean the entire structure that our program is manipulating, and "node" when we mean just one piece of the overall structure. Therefore, another synonym for "node" is "subtree."

The *root node* (or just the *root*) of a tree is the node at the top. Every tree has one root node. (A more general structure in which nodes can be arranged more flexibly is called a *graph*; you'll study graphs in 61B and later courses.)

The *children* of a node are the nodes directly beneath it. For example, the children of the `26` node in the picture are the `15` node and the `33` node.

A *branch* node is a node that has at least one child. A *leaf* node is a node that has no children. (The root node is also a branch node, except in the trivial case of a one-node tree.)

• The `Tree` abstract data type

Lisp has one built-in way to represent sequences, but there is no official way to represent trees. Why not?

- Branch nodes may or may not have data.
- Binary vs. n-way trees.
- Order of siblings may or may not matter.
- Can tree be empty?

We'll get back to some of these variations later, but first we'll consider a commonly used version of trees, in which every tree has at least one node, every node has a datum, and nodes can have any number of children. Here are the constructor and selectors:

```
(make-tree datum children)
(datum node)
(children node)
```

The selector `children` should return a *list of trees*, the children of the node. These children are themselves trees. There is a name for a list of trees: a *forest*. It's very important to remember that Tree and Forest are two different data types! A forest is just a sequence, although its elements are required to be trees, and so we can manipulate forests using the standard procedures for sequences (`cons`, `car`, `cdr`, etc.); a tree is *not* a sequence, and should be manipulated only with the tree constructor and selectors.

A leaf node is one with no children, so its `children` list is empty:

```
(define (leaf? node)
  (null? (children node)) )
```

This definition of `leaf?` should work no matter how we represent the ADT.

The straightforward implementation is

```
;;;;;                          In file cs61a/lectures/2.2/tree1.scm
(define make-tree cons)
(define datum car)
(define children cdr)
```

• Mapping over trees

One thing we might want to do with a tree is create another tree, with the same shape as the original, but with each datum replaced by some function of the original. This is the tree equivalent of `map` for lists.

```
;;;;;                          In file cs61a/lectures/2.2/tree1.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (map (lambda (t) (treemap fn t))
                  (children tree) )))
```

This is a remarkably simple and elegant procedure, especially considering the versatility of the data structures it can handle (trees of many different sizes and shapes). It's one of the more beautiful things you'll see in the course, so spend some time appreciating it.

Every tree node consists of a datum and some children. In the new tree, the datum corresponding to this node should be the result of applying `fn` to the datum of this node in the original tree. What about the children of the new node? There should be the same number of children as there are in the original node, and each new child should be the result of calling `treemap` on an original child. Since a forest is just a list, we can use `map` (not `treemap`!) to generate the new children.

• Mutual recursion

Pay attention to the strange sort of recursion in this procedure. `Treemap` does not actually call itself! `Treemap` calls `map`, giving it a function that in turn calls `treemap`. The result is that each call to `treemap` may give rise to any number of recursive calls, via `map`: one call for every child of this node.

This pattern (procedure `A` invokes procedure `B`, which invokes procedure `A`) is called *mutual recursion*. We can rewrite `treemap` without using `map`, to make the mutual recursion more visible:

```
;;;;;                          In file cs61a/lectures/2.2/tree11.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (forest-map fn (children tree))))

(define (forest-map fn forest)
  (if (null? forest)
      '()
      (cons (treemap fn (car forest))
            (forest-map fn (cdr forest)))))
```

`Forest-map` is a helper function that takes a forest, not a tree, as argument. `Treemap` calls `forest-map`, which calls `treemap`.

Mutual recursion is what makes it possible to explore the two-dimensional tree data structure fully. In particular, note that reaching the base case in `forest-map` does not mean that the entire tree has been visited! It means merely that one group of sibling nodes has been visited (a "horizontal" base case), or that a node has no children (a "vertical" base case). The entire tree has been seen when every child of the root node has been completed.

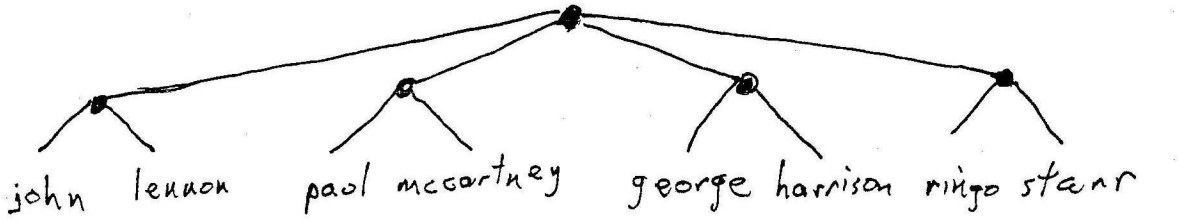
Note that we use `cons`, `car`, and `cdr` when manipulating a forest, but we use `make-tree`, `datum`, and `children` when manipulating a tree. Some students make the mistake of thinking that data abstraction means "always say `datum` instead of `car`"! But that defeats the purpose of using different selectors and constructors for different data types.

• Deep lists

Trees are our first two-dimensional data structure. But there's a sense in which any list that has lists as elements is also two-dimensional, and can be viewed as a kind of tree. We'll use the name *deep lists* for lists that contain lists. For example, the list

```
[[john lennon] [paul mccartney] [george harrison] [ringo starr]]
```

is probably best understood as a sequence of sentences, but instead we can draw a picture of it as a sort of tree:



Don't be confused; this is *not* an example of the Tree abstract data type we've just developed. In this picture, for example, only the "leaf nodes" contain data, namely words. We didn't make this list with `make-tree`, and it wouldn't make sense to examine it with `datum` or `children`.

But we can still use the *ideas* of tree manipulation if we'd like to do something for every word in the list. Compare the following procedure with the first version of `treemap` above:

```
;;;;;                          In file cs61a/lectures/2.2/tree22.scm
(define (deep-map fn lol)
  (if (list? lol)
      (map (lambda (element) (deep-map fn element))
           lol)
      (fn lol)))
```

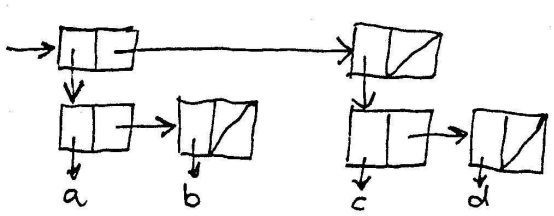
The formal parameter `lol` stands for "list of lists." This procedure includes the two main tasks of `treemap`: applying the function `fn` to one datum, and using `map` to make a recursive call for each child.

But `treemap` applies to the Tree abstract data type, in which every node has both a datum and children, so `treemap` carries out both tasks for each node. In a deep list, by contrast, the "branch nodes" have children but no datum, whereas the "leaf nodes" have a datum but no children. That's why `deep-map` chooses only one of the two tasks, using `if` to distinguish branches from leaves.

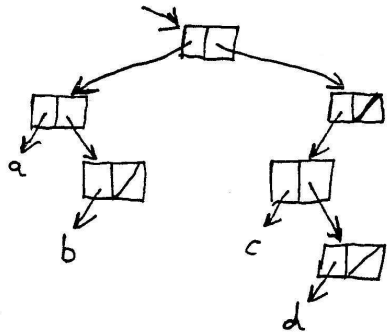
Note: SICP does not define a Tree abstract data type; they use the term "tree" to describe what I'm calling a deep list. So they use the name `tree-map` in Exercise 2.31, page 113, which asks you to write what I've called `deep-map`. (Since I've done it here, you should do the exercise without using `map`.) SICP does define an abstract data type for *binary* trees, in which each node can have a `left-branch` and/or a `right-branch`, rather than having any number of children.

• **Car/cdr recursion**

Consider the deep list `((a b) (c d))`. Ordinarily we would draw its box and pointer diagram with a horizontal spine at the top and the sublists beneath the spine:



But imagine that we grab the first pair of this structure and "shake" it so that the pairs fall down as far as they can. We'd end up with this diagram:



Note that these two diagrams represent the same list! They have the same pairs, with the same links from one pair to another. It's just the position of the pairs on the page that's different. But in this new picture, the structure looks a lot like a binary tree, in which the branch nodes are pairs and the leaf nodes are atoms (non-pairs). The "left branch" of each pair is its `car`, and the "right branch" is its `cdr`. With this metaphor, we can rewrite `deep-map` to look more like a binary tree program:

```
;;;;;                          In file cs61a/lectures/2.2/tree3.scm
(define (deep-map fn xmas)
  (cond ((null? xmas) '())
        ((pair? xmas)
         (cons (deep-map fn (car xmas))
               (deep-map fn (cdr xmas))))
        (else (fn xmas))))
```

(The formal parameter `xmas` reflects the fact that the picture looks kind of like a Christmas tree.)

This procedure strongly violates data abstraction! Ordinarily when dealing with lists, we write programs that treat the car and the cdr differently, reflecting the fact that the car of a pair is a list element, whereas the cdr is a sublist. But here we treat the car and the cdr identically. One advantage of this approach is that it works even for improper lists:

```
> (deep-map square '((3 . 4) (5 6))
((9 . 16) (25 36))
```

• **Tree recursion**

Compare the car/cdr version of `deep-map` with ordinary `map`:

```
(define (map fn seq)
  (if (null? seq)
      '()
      (cons (fn (car seq))
            (map fn (cdr seq)))))
```

Each non-base-case invocation of `map` gives rise to one recursive call, to handle the cdr of the sequence. The car, an element of the list, is not handled recursively.

By contrast, in `deep-map` there are *two* recursive calls, one for the car and one for the cdr. This is what makes the difference between a sequential, one-dimensional process and the two-dimensional process used for deep lists and for the Tree abstraction.

A procedure in which each invocation makes more than one recursive call is given the name *tree recursion* because of the relationship between this pattern and tree structures. It's tree recursion only if each call (other than a base case) gives rise to two or more recursive calls; it's not good enough to have two recursive calls of which only one is chosen each time, as in the following non-tree-recursive procedure:

```
(define (filter pred seq)
  (cond ((null? seq) '())
        ((pred (car seq)) (cons (car seq) (filter pred (cdr seq))))
        (else (filter pred (cdr seq)))))
```

There are two recursive calls to `filter`, but only one of them is actually carried out each time, so this is a sequential recursion, not a tree recursion.

A program can be tree recursive even if there is no actual tree-like data structure used, as in the Fibonacci number function:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

This procedure just handles numbers, not trees, but each non-base-case call adds the results of two recursive calls, so it's a tree recursive program.

• **Tree traversal**

Many problems involve visiting each node of a tree to look for or otherwise process some information there. Maybe we're looking for a particular node, maybe we're adding up all the values at all the nodes, etc. There is one obvious order in which to traverse a sequence (left to right), but many ways in which we can traverse a tree.

In the following examples, we "visit" each node by printing the datum at that node. If you apply these procedures to actual trees, you can see the order in which the nodes are visited.

Depth-first traversal: Look at a given node's children before its siblings.

```
;;;;;                          In file cs61a/lectures/2.2/search.scm
(define (depth-first-search tree)
  (print (datum tree))
  (for-each depth-first-search (children tree)))
```

This is the easiest way, because the program's structure follows the data structure; each child is traversed in its entirety (that is, including grandchildren, etc.) before looking at the next child.

Breadth-first traversal: Look at the siblings before the children.

What we want to do is take horizontal slices of the tree. First we look at the root node, then we look at the children of the root, then the grandchildren, and so on. The program is a little more complicated because the order in which we want to visit nodes isn't the order in which they're connected together.

To solve this, we use an extra data structure, called a *queue*, which is just an ordered list of tasks to be carried out. Each "task" is a node to visit, and a node is a tree, so a list of nodes is just a forest. The iterative helper procedure takes the first task in the queue (the car), visits that node, and adds its children at the end of the queue (using `append`).

```
;;;;;                          In file cs61a/lectures/2.2/search.scm
(define (breadth-first-search tree)
  (bfs-iter (list tree)))

(define (bfs-iter queue)
  (if (null? queue)
      'done
      (let ((task (car queue)))
        (print (datum task))
        (bfs-iter (append (cdr queue) (children task))))))
```

Why would we use this more complicated technique? For example, in some situations the same value might appear as a datum more than once in the tree, and we want to find the *shortest* path from the root node to a node containing that datum. To do that, we have to look at nodes near the root before looking at nodes far away from the root.

Another example is a game-strategy program that *generates* a tree of moves. The root node is the initial board position; each child is the result of a legal move I can make; each child of a child is the result of a legal move for my opponent, and so on. For a complicated game, such as chess, the move tree is much too large to generate in its entirety. So we use a breadth-first technique to generate the move tree up to a certain depth (say, ten moves), then we look for desirable board positions at that depth. (If we used a depth-first program, we'd follow one path all the way to the end of the game before starting to consider a different possible first move.)

For binary trees, within the general category of depth-first traversals, there are three possible variants:

Preorder: Look at a node before its children.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (pre-order tree)
  (cond ((null? tree) '())
        (else (print (entry tree))
              (pre-order (left-branch tree))
              (pre-order (right-branch tree)) )))
```

Inorder: Look at the left child, then the node, then the right child.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (in-order tree)
  (cond ((null? tree) '())
        (else (in-order (left-branch tree))
              (print (entry tree))
              (in-order (right-branch tree)) )))
```

Postorder: Look at the children before the node.

```
;;;;;                          In file cs61a/lectures/2.2/print.scm
(define (post-order tree)
  (cond ((null? tree) '())
        (else (post-order (left-branch tree))
              (post-order (right-branch tree))
              (print (entry tree)) )))
```

For a tree of arithmetic operations, preorder traversal looks like Lisp; inorder traversal looks like conventional arithmetic notation; and postorder traversal is the HP calculator "reverse Polish notation."

• Path finding

As an example of a somewhat more complicated tree program, suppose we want to look up a place (e.g., a city) in the world tree, and find the path from the root node to that place:

```
> (find-place 'berkeley world-tree)
(world (united states) california berkeley)
```

If a place isn't found, `find-place` will return the empty list.

To find a place within some tree, first we see if the place is the datum of the root node. If so, the answer is a one-element list containing just the place. Otherwise, we look at each child of the root, and see if we can find the place within that child. If so, the path within the complete tree is the path within the child, but with the root datum added at the front of the path. For example, the path to Berkeley within the USA subtree is

```
((united states) california berkeley)
```

so we put `world` in front of that.

Broadly speaking, this program has the same mutually recursive tree/forest structure as the other examples we've seen, but one important difference is that once we've found the place we're looking for, there's no need to visit other subtrees. Therefore, we don't want to use `map` or anything equivalent to handle the children of a node; we want to check the first child, see if we've found a path, and only if we haven't found it should we go on to the second child (if any). This is the reason for the `let` in `find-forest`.

```
;;;;;                          In file cs61a/lectures/2.2/world.scm
(define (find-place place tree)
  (if (eq? place (datum tree))
      (cons (datum tree) '())
      (let ((try (find-forest place (children tree))))
        (if (not (null? try))
            (cons (datum tree) try)
            '()))))

(define (find-forest place forest)
  (if (null? forest)
      '()
      (let ((try (find-place place (car forest))))
        (if (not (null? try))
            try
            (find-forest place (cdr forest))))))
```

(Note: In 61B we come back to trees in more depth, including the study of *balanced* trees, i.e., using special techniques to make sure a search tree has about as much stuff on the left as on the right.)

• The `Scheme-1` interpreter

[This topic may be in week 5 or in week 6 depending on the holiday schedule.]

We're going to investigate a Scheme interpreter written in Scheme. SICP has a rather large and detailed Scheme interpreter in Chapter 4, which we'll get to near the end of the semester. But students often find that program intimidating, so we're going to work up to it with a series of three smaller versions that leave out some details and some of the features of real Scheme. This week we'll use the `Scheme-1` interpreter.

We weren't ready for this investigation until we had the idea of lists that contain sublists, because that's what a Scheme program is – a list. That's the point of all those parentheses; the Scheme language can look at a Scheme program as data, rather than as something different from data.

Here's how we use the interpreter:

```
STk> (load "~cs61a/lib/scheme1.scm")
STk> (scheme-1)
Scheme-1: (+ 2 3)
5
Scheme-1: ((lambda (x) (* x 3)) 4)
12
```

To leave `Scheme-1` and return to the STk prompt, just enter an illegal expression, such as `()`.

Why bother? What good is an interpreter for Scheme that we can't use unless we already have another interpreter for Scheme?

- It helps you understand evaluation models.
- It lets us experiment with modifications to Scheme (new features).
- Even real Scheme interpreters are largely written in Scheme.
- It illustrates a big idea: *universality*.

This week's interpreter implements the *substitution* model of evaluation that we learned in Chapter 1 of SICP. In Chapter 3, we'll get to a more complicated but more realistic evaluation model, called the *environment* model.

Universality means we can write *one program* that's equivalent to all other programs. We'll talk more about this when we see SICP's full Scheme interpreter in Chapter 4. This week's interpreter, although universal in principle, doesn't make the point clearly because it's quite difficult to write serious programs in it, mainly because it lacks `define`.

Our Scheme interpreter leaves out many of the important components of a real one. It gets away with this by taking advantage of the capabilities of the underlying Scheme. Specifically, we don't deal with storage allocation, tail recursion elimination, or implementing any of the Scheme primitives. All we *do* deal with is the evaluation of expressions. That turns out to be quite a lot in itself, and pretty interesting.

Here is a one-screenful version of a Scheme interpreter, using the substitution model, with most of the details left out:

```
;;;;;                          In file cs61a/lectures/2.2/tiny.scm
(define (scheme)
  (display "> ")
  (print (eval (read)))
  (scheme) )

(define (eval exp)
  (cond ((self-evaluating? exp) exp)
        ((symbol? exp) (look-up-global-value exp))
        ((special-form? exp) (do-special-form exp))
        (else (apply (eval (car exp))
                     (map eval (cdr exp)) ))))

(define (apply proc args)
  (if (primitive? proc)
      (do-magic proc args)
      (eval (substitute (body proc) (formals proc) args))))
```

Although the versions we can actually run are bigger, this really does capture the essential structure of every Lisp interpreter, namely, a mutual recursion between `eval` (evaluate an expression) and `apply` (apply a function to arguments). To evaluate a procedure call means to evaluate the subexpressions recursively, then apply the `car` (a function) to the `cdr` (the arguments). To apply a function to arguments means to evaluate the body of the function with the argument values in place of the formal parameters.

The `substitute` procedure is essentially the `substitute2` that you wrote in last week's homework, except that it has to be a little more complicated to avoid substituting for quoted symbols and for the formal parameters of a `lambda` inside the body.

What's left out? Primitives, special forms, and a lot of details.

The `Scheme-1` interpreter has only three special forms: `quote`, `if`, and `lambda`. In particular, it doesn't have `define`, so there are no global variables, and we can't give procedures global names. If we need a name for a procedure, we have to use it as an argument to another procedure. In particular, if we want to write *recursive* procedures we have to use a trick that was an extra-for-experts in week 2:

```
Scheme-1: ((lambda (n)
             ((lambda (f) (f f n))   ; the "Y combinator"
              (lambda (fact n)
                (if (= n 0)
                    1
                    (* n (fact fact (- n 1))) )) ))
           5)
120
```