Section Notes 2

CS51—Spring, 2009

Week of February 8, 2009

Outline

1. Review
2. Abstraction
3. Trees
4. Reading and writing Scheme
5. Style
6. Answers

1 Review

By now you should be familiar with the following concepts:

- The Scheme evaluation rule and exceptions to it (special forms)
- Side effects, including variable assignment, and the fact that we are programming without them
- How to write a simple recursive function

2 Abstraction

We hope that you will take a great deal away from CS51. However, *abstraction* is a central concept that stands out among all the others we will study this semester.

SEGWAY

An *abstraction* in programming is a tool used to generalize code to make it more broadly applicable. One of the first forms of abstraction that you encountered as programmers was the "function". A function parameterizes a section of code so that the same code can be applied to different inputs. While a parameterized function is abstract in that it can be applied to different arguments, it is also abstract in that the details of the function's implementation are hidden.

As an example, if you took CS50, you will remember the Spell Checker problem set where you were asked to design and implement functions that would be used to check for spelling errors in an input file. Each student's "check" function conformed to the same contract - take a string as an argument and return true if the word appeared in the dictionary file and false otherwise. The competition to produce the fastest implementation of these functions was possible because the function interface effectively abstracted away the particular implementation.

Why do we care about abstraction? The simple answer is that it makes our jobs as programmers easier. It's much harder to reason about complex systems than about individual units of programs that have been broken down into pieces with well-defined boundaries. Functional interfaces and contracts are one such boundary, but they are not the only kind that come up in software design.

2.1 Data Abstraction

Besides functional or control abstraction, as described above, CS51 introduces *data abstraction.* Data abstraction separates the conceptual properties of a datatype from its implementation. An example of an *Abstract DataType* or *ADT* is a *Set.* Conceptually, a Set is a collection of unique values and nothing more. This abstract definition provides no guidance as to how a Set should be represented on a computer. In Scheme, one might consider storing the contents of a given Set in a list, however this is certainly not the only possible concrete implementation of this type, and is probably a rather poor choice of representation. Why might a list be a poor choice? Later in this section we will discuss binary trees, which is one of the many possible concrete datatypes that can be used to represent the Set ADT in Scheme.

2.2 Representing Data Types in Scheme

Scheme provides us with primitive types including numbers, strings, and lists, but it also provides us with a mechanism for defining our own structured types. Much like the struct that you have seen in C, Scheme provides a mechanism for defining a type with named components. For example, if you wanted to write a program that would store and manipulate student information, you might want the following structure definition in your program:

```
(define-struct student (name gradyear house concentration gpa))
```

Once you have introduced this new concrete type in Scheme, you automatically get some help in the form of functions that create, check, and interrogate students[1]. For our example type, these generated functions are:

```
make-student
student?
student-name
student-gradyear
student-house
student-concentration
student-gpa
```

From this representation of a student record, you could design a whole program that allows TFs and Professors to store and retrieve information about their students. Note, however, that this structure is still

[1]Note: no students were harmed in the making of these notes.

somewhat abstract in that it does not tie us to a specific representation of the fields. For an example, think about the various ways that you might choose to store the gpa for a student. Over the next few months we will learn more about how PLT Scheme helps us enforce data *encapsulation*, or the strict separation of a datatype's interface from its implementation.

For more information on data abstraction and how to work with structures in Scheme, check out Chapter 6 of "How To Design Programs" (http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-9.html#node_chap_6) or Chapter 5 of "Guide: PLT Scheme" (http://docs.plt-scheme.org/guide/define-struct.html).

2.3 Variables as Abstractions

The Scheme special form `let` allows you to introduce local variables. Creating local names for values lets you avoid typing a common subexpression multiple times, and lets the interpreter avoid evaluating it multiple times.

The *bindings* you create with `let` remain active within the `let` block, but nowhere else. Only the `expression` below would have access to these particular values for `sym1` and `sym2`:

```
(let  (  [sym1  expr1]
         [sym2  expr2]
         ...             )
      expression)
```

`let` introduces a variety of interesting technicalities. For instance, the semantics of `let` say that the symbols get their values simultaneously—in other words, you can't express the value of one in terms of another.

Before moving on to talk about some concrete data types, it is worth noting that Scheme's `let` expression provides a form of abstraction. Recall that `let` can be used to associate identifiers with the result of evaluting expressions. What is the result of evaluating the following expression?

```
(let ([size (if (< 1 0) "Small" "X-Large")]
      [color "Chartreuse"]
      [material "Wool"])
   (list size color material))
```

While the example above is trivial, it demonstrates that the final list expression can be written without any knowledge of how size, color or material are computed. A more complicated example might replace the list expression with an expression that computes the price of a sweater based on these three parameters. What might that code look like? Is there any limit to the complexity that we can put in our `let` identifier definitions?

3 Trees

3.1 Definition

A tree is a connected graph that does not contain a cycle, and a binary tree is a tree data structure in which each node contains at most 2 children.

A tree is a data structure in-and-of-itself; however, trees can also be used to represent other Abstract Data Types. Dictionaries are often represented as trees, since trees naturally allow for ordering data. Trees can also be used to represent other undirected graphs (by picking a root node in a graph) in order to accomplish Breadth-First Search or Depth-First Search (explained in a following section).

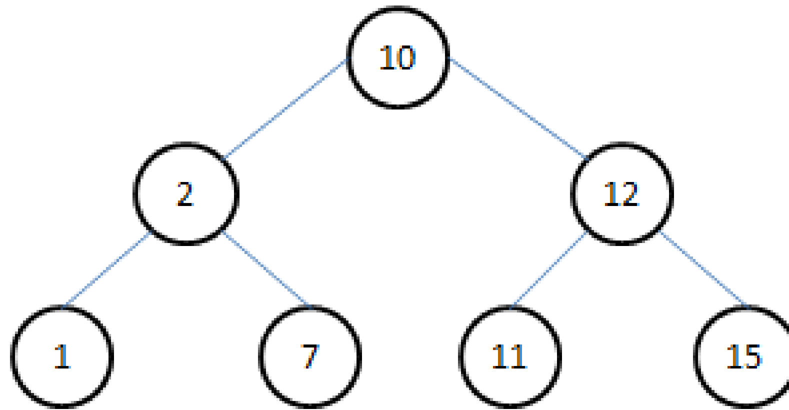

Figure 1: A properly-formed binary search tree

3.2 Trees in Scheme

How can we depict the preceding binary tree as a list in Scheme?

How can we depict the same graph as a recursively-defined binary tree in Scheme? You can think of it in terms of list, or a Scheme structure:

```
(define-struct tree (value left right))
```

.

3.3 Inserting Elements in Trees

The tree provided in figure 1 is called a binary search tree because its nodes obey a set of simple rules. What would the binary search tree look like with the element 9 inserted? What would the Scheme data structure look like?

Try inserting the elements 3, 4, and 14.

3.4 Searching Trees

Breadth-first search is a graph-searching algorithm that starts at the root and searches all of the root's child nodes. Then it explores all the child nodes of those nodes, and repeats, until the item is found, or the graph has been exhaustively searched.

Depth-first search is a graph-searching algorithm that starts at the root and explores all the way down a branch before backtracking. In backtracking, we return to the most recently visited node that has other children and explore all the way down this next branch; we repeat until the item is found, or the graph has been exhaustively searched.

For some graphs, such as binary serach trees (see Figure 1), the search process should be simpler. In what way can we use the proper ordering of the tree to find the element we are searching for?

4 Reading and writing scheme functions

4.1 A mystery function

Try to figure out what this function does. You may want to consider the following:

- What type of object does the function evaluate to?
- What do the various cases mean?
- How are subproblem results being combined, and what does this imply?

```
(define (enigma baz qux)
  (cond [(null? qux) 0]
        [(equal? baz (car qux)) (+ 1 (enigma baz (cdr qux)))]
        [(pair? (car qux)) (+ (enigma baz (car qux))
                              (enigma baz (cdr qux)))]
        [else (enigma baz (cdr qux))]))
```

4.2 Mystery function challenge

This one's a little tricky. Try applying the following strategy:

- Don't get bogged down in details. What does the function do in the most general case?
- There appear to be two recursive cases. Is this essential to the function or perhaps somewhat misleading?

```
(define (mystery foo bar)
  (cond [(and (null? foo) (null? bar)) '()]
        [(null? foo) (append bar foo)]
        [(null? bar) (cons (car foo)
                           (mystery bar (cdr foo)))]
        [else (cons (car foo)
                    (cons (car bar)
                          (mystery (cdr foo)
                                   (cdr bar))))]))
```

5 Style notes

Starting now, we will be watching your coding style like hawks, and so should you. Prudent and consistent style is a big help when you read over code, debug it or try to remember how it works.

A few important points to remember:

- Use `cond` instead of nested `if`
- Remember that a function you use might already return the value you need, eliminating the need to check its value:

  ```
  (if (= (+ (square a) (square b)) (square c)) #t #f)
  ```

  becomes simply:

  ```
  (= (+ (square a) (square b)) (square c))
  ```

  Think functionally. Every expression already yields a value, so there is rarely a need to explicitly `return` one object or another.
- Learn to use `and` and `or`
- Be sure to use `cons` when all you need to do is add an item to the front of a list. Never use the construct `(append (list x) ...)`. Append works to join two lists together; however if the item being added is not originally a list, it shouldn't be made into a list, then appended.
- Use line breaks and indentation well. Line up the arguments to a function so each one starts at the same horizontal position. Avoid very long and very short lines in general. Be sure to use an editor like Dr. Scheme, emacs, or vim that can intelligently indent Scheme for you.

6 Answers

6.1 Trees

- From 3.2: (list 1 2 7 10 11 12 15)
- From 3.2: (list (10 (2 (1 empty empty) (7 empty empty)) (12 (11 empty empty) (15 empty empty))))
- From 3.3: (list (10 (2 (1 empty empty) (7 empty (9 empty empty))) (12 (11 empty empty) (15 empty empty))))