

# CS51 Assignment 3:

## $\lambda!$

Due: Friday, February 27th, 2009, 5:00PM

*Total Points: 56 (including 10 style points)*

In this assignment, you will discover the magic of higher-order functions. You will implement analogues of `fold` and `map` for binary trees, and put them to various uses, including a solver for a simple game. *For this assignment, you should strive to maximize elegance rather than speed.*

## 1 Higher-Order Functions

*(9 points total)*

One of the most powerful aspects of Scheme – as well as other functional programming languages – is that its functions are first-class values. This means that you can treat functions the same way you treat any other datatype in Scheme. Beyond simply calling functions, this means that functions can take functions as arguments and can even return functions. Functions that do either of these things are called *higher-order functions*.

### 1.1 Lambda

Scheme provides a special form, `lambda`, that allows us to create anonymous functions, or functions that don't have names. Up until now, we've written our functions

```
(define (foo bar) bar)
```

which creates a named function `foo`. However, this is really syntactic sugar<sup>1</sup> for

```
(define foo (lambda (bar) bar))
```

In both expressions, we've created a function and bound it to `foo`. In the first expression, however, we created and named the function all in one step. In the second expression, the process of giving the function a name is separate from the process of creating the function. In fact, it's best to think of functions as entities that *may* be bound to a symbol. When we

---

<sup>1</sup>That is, a convenient, but equivalent, alternative syntax.

write `(define foo 5)` we are binding the symbol `foo` to the number 5; we are *not* giving the number 5 the name `foo`. Try to think about functions the same way.

So, what's the syntax of `lambda`? It's just `(lambda (<arguments>) <expression>)`. A function that returns its only argument is just:

```
(lambda (x) x)
```

`lambda` becomes really powerful when we return functions from other functions. Within the body of the `lambda`, we can refer to variables that are in the scope of the function. A trivial example is creating a function that returns the same value for all inputs. We can write `create-constant-function`, which returns such a function, like this:

```
(define (create-constant-function n)
  (lambda (x) n))

((create-constant-function 5) false) => 5
((create-constant-function 100) 1) => 100
((create-constant-function false) empty) => false
```

This probably seems a little weird, but you'll get the hang of it.

### Exercise 1.

9 points

Now you get to write your own higher-order functions!

- (a) [2 points] Write `n-adder`, which takes an argument `n` and returns a function that takes a single argument `x` and returns the sum of `x` and `n`.
- (b) [2 points] Write `negate-pred`, which takes a predicate `f` that takes a single argument and returns a function that computes the negation of `f`. That is, `(f 5)` is true if and only if `((negate-pred f) 5)` is false.
- (c) [2 points] Write `my-compose`, which takes two functions `f` and `g` (each taking a single argument), and returns the composition `f . g`: i.e., a function that applies `g` to its argument, then applies `f` to the result.
- (d) [3 points] Write `my-compose-list`, which takes a list of functions (`f1 ... fn`), and returns the composition `f1 . ... . fn`. You should do this using `fold` over lists (called `foldr` in the incarnation of Scheme we are currently using), rather than making an explicit recursive call. You may also find the identity function `id` useful.

## 1.2 Map, Fold, and Filter

By now you've seen `map`, `fold`, and `filter` on lists. Combined with `lambda`, we can do some pretty amazing things with very few lines of code. Throughout the rest of the term, think about how you can use these functions — your code will be clearer, cleaner, and more concise! In this assignment, we'll look at `map` and `fold` on trees.

## 2 Tree Folding

(22 points total)

*I think that I shall never see  
a poem lovely as a naturally recursive data structure.*  
– Joyce Kilmer, “Trees” (paraphrase)

We’ve included a tree module similar to the one from Assignment 1. It provides the constructors `make-branch` (building a tree from two subtrees), and `empty-tree` (the empty tree), and includes the usual predicates (`tree?`, `empty-tree?`). In addition, it provides the functions `tree->list` and `list->tree`, which convert between trees and nested lists for ease of testing (and the associated predicate `tree-list?`, which tests whether a nested list actually represents a tree). More interestingly, it also provides `tree-fold`, which is the tree analogue of the `fold` function for lists that we saw in lecture. Before we get to `tree-fold`, though, let’s examine more closely the code in the tree module.

### Exercise 2.

4 points

For these questions, please limit your answers to at most two sentences each.

(a) [2 points] On Assignment 1, you wrote a recursive `binary-tree?` predicate that verified that a tree was a valid binary tree. Why is it valid for us to instead use a simple `tree?` predicate that only checks one level deep?

(b) [2 points] What does the `takes-n-args` function do? (The online PL/T Scheme documentation may be helpful here.) Why might it be useful? Why isn’t it available outside the module (i.e. why didn’t we provide it)?

### Exercise 3.

18 points

Now we’ll write some tree operations using `tree-fold`. On lists, the function you passed to `fold` took two arguments — an element, and the result of the fold on the rest of the list. Folding trees is a little different because each node on the tree has two children. So, the function you pass to `tree-fold` should have three arguments: the value at a node, the result of the fold on the left child, and the result of the fold on the right child. Like the list fold, we have to specify a “zero” value. This will represent the result of the fold on the `empty-tree`. In other words,

```
> (tree-fold f 42 empty-tree) => 42
```

regardless of what `f` is.

(a) [3 points] Write `tree-sum` using `tree-fold`. `tree-sum` takes a tree and returns the sum of all the values in the tree. Assume that the value at every node is a number.

(b) [3 points] Write `tree-mirror` using `tree-fold`. `tree-mirror` takes a tree and returns the reflection of the tree. If the path from the root to a node was (left, right, left) in the tree, the path to the same node in the reflection will be (right, left, right).

(c) [3 points] Write `tree-map` using `tree-fold`, placing your definition in the module itself. Also add a contract to the module for `tree-map`.

- (d) [3 points] Write `tree-square` using `tree-map`. `tree-square` takes a tree and returns the same tree with every value squared. Assume that the value of every node is a number.
- (e) [3 points] Write `tree->ordered-list` using `tree-fold`. `tree->ordered-list` takes a tree and returns a list of all the elements in the tree in left-to-right order, such that if the tree were a binary search tree, the numbers would be sorted low-to-high. (Don't worry about efficiency.)
- (f) [3 points] Above, you wrote an implementation of `tree-map`. Why isn't there a natural way to write `tree-filter`?

### 3 The Tree Game

(14 points total)

Games are especially suited to the magic of higher-order functions, since a strategy in a game is essentially a function from game states to actions. In this section, we will be studying a simple two-player game that is played on a binary tree. Both players begin with zero score. The player whose turn it is adds the number at the root of the tree to their score, then chooses one of the two subtrees; it is then the other player's turn to move on the chosen subtree, and the game continues in this fashion until it terminates at an empty tree. The goal is to get more points than the other player.

#### Exercise 4.

14 points

Here is a possible strategy for the Tree Game: if the tree `t` is empty (i.e. the game is over), the player acknowledges this by returning "F" for "finished". Otherwise, the player always chooses the left branch ("L") (rather than the right branch ("R")). This is a somewhat silly strategy; let's write a less-silly one.

- (a) [3 points] Write `tgame-greedy-strategy`, a greedy strategy for The Tree Game. On a game tree `t`, a player playing this strategy should choose the branch that minimizes the score the other player will receive on the very next turn (consider this amount zero if there would be no next turn). If the value is the same for both subtrees, you can return either one. (If the game is already over, the strategy should return "F".) You may find the helper function `tgame-immed-val` useful.
- (b) [3 points] We would like to play some sample games to test these strategies against each other. We have provided a series of functions for testing strategies (both for correctness and for performance), but we've left the implementation of one of them as an exercise. Fill in the missing function `tgame-score`, which given two strategies (that of the player to move, and that of the opponent), determines the net score for the player to move (i.e., final score minus the opponent's final score).
- (c) [3 points] We can do better than a greedy strategy. In particular, suppose for the moment that we know our opponent's strategy. Then we can predict how our opponent will respond to any potential situation for the rest of the game. If we also assume that we ourselves will continue to choose the best response to the opponent's strategy for the rest of the game, then given any candidate move, we can completely determine how the game will

play out if we make that move; we can then use this information to select the move that will maximize our score.

Write `tgame-best-response-strategy`, which, when given the opponent's strategy, returns a strategy that will yield the highest possible score. (Hint: you can write this in terms of `tgame-score`, but you need to use recursion in an unusual way. Again, don't worry about efficiency.)

(d) [2 points] Using `tgame-test-strategies`, run the strategies you have written against each other, and describe the results qualitatively. You should have tests for each pairing of strategies written so far.

(e) [3 points] Using `tgame-best-response-strategy` is great if we know our opponent's strategy. But what if we're playing an arbitrary opponent? A standard technique in game theory is to assume *perfect play*: that is, we play a strategy that maximizes our score, under the assumption that the opponent will play a strategy that maximizes their score (under the assumption that we will play a strategy that maximizes our score, and so on).

Write `tgame-perfect-strategy`, which maximizes the current player's score assuming perfect play on the part of the opponent. (Hint: this should be short, elegant, and written in terms of `tgame-best-response-strategy`.)

(f) [0 points] (*Food for thought – not for credit.*) You may have noticed that we had to symmetrize the tests we run from `tgame-test-strategies`, to compensate for bias in the games. Which way might these randomly-generated games be biased, and why? You may want to experiment with various game tree depths.

(g) [0 points] (*Food for thought – not for credit.*) Throughout this assignment, we have repeatedly emphasized that we are looking for elegance, not efficiency. Which parts of the code we have written might be inefficient, and why? How might we make them more efficient without sacrificing elegance? (You may want to look up *memoization*.)

## 4 Finishing Up

(1 point total)

### Exercise 5.

1 point

Please tell us how much time you spent on this problem set! You may use this space to add any thoughts or questions you want to send to your TF as well, if you so desire.