Massachusetts Institute of Technology

16.410-13 Principles of Autonomy and Decision Making

Assignment #3, tutorial. Due: LEC #6

Problem PS.3.1.1: List Creation

Write a simple Scheme function, fill-list, that takes as parameters an integer, n, (which is greater than or equal to 1) and an element that can be of any type. The function should create a list of length n, with each item in the list being the element. For example,

```
(fill-list 1 'x) => (x)
(fill-list 5 'x) => (x x x x x)
(fill-list 2 '(a b)) => ((a b) (a b))
```

Problem PS.3.1.2: Tree Creation

Write a simple Scheme function, fill-tree, that takes as parameters an integer, depth, (which is greater than or equal to 2) and an integer, node-index. The function should create a binary tree of the specified depth. The format of the tree should be (node-index (left-child right-child)), where left-child and right-child may themselves be trees (this is a recursive definition). At the deepest level of the tree, left-child and right-child are just node indices. For example,

```
(fill-tree 2 1) => (1 (2 3))
(fill-tree 3 1) => (1 ((2 (3 4)) (3 (4 5))))
(fill-tree 4 1) => (1 ((2 ((3 (4 5)) (4 (5 6)))) (3 ((4 (5 6)) (5 (6
7))))))
```

Problem PS.3.1.3: Functional Composition

Write a simple Scheme program that determines the maximum depth of functional composition of a mathematical expression written in Scheme notation. For example,

```
(depth 'x) => 0
(depth '(expt x 2)) => 1
(depth '(+ (expt x 2) (expt y 2))) => 2
(depth '(/ (expt x 5) (expt (- (expt x 2) 1) (/ 5 2)))) => 4
```

Problem PS.3.1.4: Indexing Trees

It would be handy to have a procedure that is analogous to `list-ref`, but for trees. It would take a tree and an index, and return the part of the tree (a leaf or a subtree) at that index. For trees, indices will have to be lists of integers. Consider the tree

```
   (((1 2)
     3)
```

```
(4
 (5 6))
7
(8 9 10))
```

To select the element 9 out of it, we need to do something like

```
(second (fourth tree))
```

Instead, we'd prefer to do `(tree-ref (list 3 1) tree)` (note that we're using zero-based indexing, as in `list-ref`, and that the indices come in top-down order; so an index of (3 1) means you should take the fourth branch of the main tree, and then the second branch of that subtree). As another example, the element 6 could be selected by `(tree-ref (list 1 1 1) tree)`.

Also, it's okay for the result to be a subtree, rather than a leaf. So `(tree-ref (list 0) tree)` should return `((1 2) 3)`.

Write the `tree-ref` procedure.

```
(define tree-ref
 (lambda (index tree) your_code_here))
```

Problem PS.3.2.1: Survey

We're always trying to improve 16.410. One of the factors that has a strong impact on what we teach and how we teach it are assumptions about the background of students who take the course. To help us in planning for future years, we'd like you to take a couple of minutes to tell us about your background before starting 16.410. This will be used purely for course development purposes and will have no impact whatsoever on your grade. Thanks!

1. What's your year?
2. How would you describe yourself as a programmer before starting the course?
3. Have you programmed in Basic?
4. Have you programmed in C?
5. Have you programmed in C++?
6. Have you programmed in Java?
7. Have you programmed in Scheme (or Lisp)?
8. How big is the largest program you have written (from scratch)?
9. Have you had a job (including UROP) as a programmer? Sorry, HTML does not count.
10. Have you taken a probability course, e.g. 6.041?
11. Have you taken an algorithms course, e.g. 6.046?

Problem PS.3.2.2: Hours

We want to understand how much time it took students to answer the questions on the problem sets.

Approximately how many hours did you spend really working on this problem set?