# Problem Set 0: Scheme Warm-Up

*6.034 Fall 2004*

`http://www.ai.mit.edu/courses/6.034f/`

**Due: Tuesday, September 14th, 2004**

The purpose of this problem set is to familiarize you with this term's problem set system and to serve as a diagnostic for programming ability and facility with MIT Scheme. 6.034 uses MIT Scheme for all of its problem sets and you will be called on to understand the functioning of large systems, as well as to write significant pieces of code yourself.

While coding is not, in itself, a focus of this class, artificial intelligence is a hard subject full of subtleties. As such, it is important that you be able to focus on the problems you are solving, rather than the mechanical code necessary to implement the solution. If you struggle with the problems in this problem set, you will likely struggle with all subsequent problem sets as well, and 6.034 will be very difficult for you.

If Scheme doesn't come back to you by the end of this problem set, we recommend that you seek extra help through the Course 6/HKN tutoring program (`http://hkn.mit.edu/act-tutoring.html`), which matches students who want help with students who've taken and done well in a class. The department pays the tutor, and the program comes highly recommended.

## Scheme References

Some resources to help you knock the rust off of your Scheme:

- Your trusty 6.001 book, *Structure and Interpretation of Computer Programming*, now available online at: `http://mitpress.mit.edu/sicp/full-text/book/book.html`
- The MIT Scheme Reference Manual and User's Manual are available at `http://www.gnu.org/software/mit-scheme/index.html#Documentation`
- Many copies of emacs include this documentation, accessible through the `M-x info` command
- Course 6/HKN tutoring program `http://hkn.mit.edu/act-tutoring.html`

## Problem Set Logistics

The first thing you need to do is set up a 6.034 section in your Athena files. From any Athena terminal run this command: `add 6.034; /mit/6.034/bin/6.034-setup`

This script will:

1. Create a `6.034-psets` directory in your home directory. Warning: this may destroy any pre-existing directory with the same name!
2. Set AFS permissions on `/6.034-psets` to allow your TA to access your problem sets. Modifying these permissions may affect your grade.
3. Create subdirectories for each problem set.

Each problem set will involves you producing a file called ps*n*`.scm` (e.g. Problem Set 1 is ps1.scm) This is the file which your TA will use to grade your problem set.

Every problem set comes in four pieces:

1. The problem set proper
2. A scheme file called `psN.scm` with skeleton code to enter your problem set work into. This is the file which the TA will test.
3. Supplementary code used by the problem set (e.g. `match.scm` for this problem set.)
4. A set of public test cases which you can use to check your work.

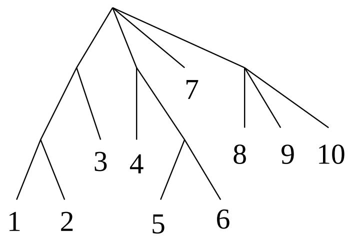

Figure 1: Example Tree

To run the public tests, run scheme and load the tester `(load "tester.scm")`, then run the tests with `(test-file "ps0" "ps0-publictest")`.

Note that looking at the public test cases is a good way to get a feel for what you're supposed to be doing! If the problem seems ambiguous, check the public test cases first: they will likely contain an example that resolves the ambiguity.

In addition, there will be a set of hidden test cases which your TA will use in grading your problem sets. These will be released only after the problem set has been graded.

Your grade will be determined by your performance on the public test cases (1/3) and the hidden test cases (2/3).

## 1 Warm-Up Stretch

Write the following functions:

1. cube: e.g. $1^3 = 1$, $2^3 = 8$, $3^3 = 27$, ...
2. factorial: e.g. $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, $5! = 120$, ... (you can assume you will be given only non-negative integers)
3. count-pattern: count how many times pattern occurs in a list of symbols, e.g. `(count-pattern '(a b) '(a b c e b a b f))` should return 2 and `(count-pattern '(a b a) '(g a b a b a b a))` should return 3.

## 2 Expression Depth

One way to measure the complexity of a mathematical expression is the depth of the expression describing it in Scheme. Write a program that finds the depth of an expression.

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`

## 3 Tree Reference

Your job is to write a procedure that is analogous to list-ref, but for trees. This "tree-ref" procedure will 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 in Figure 1, represesented by Scheme list: `(((1 2) 3) (4 (5 6)) 7 (8 9 10))`

To select the element 9 out of it, we'd normally need to do something like `(second (fourth tree))`

Instead, we'd prefer to do `(tree-ref tree (list 3 1))` (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 tree (list 1 1 1))`.

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

If the index specifies an element that doesn't exist in the tree, then it's OK for it to return an error.

## 4 Matching

*Throughout the semester, you will need to understand, manipulate, and extend complex algorithms implemented in Scheme. You may also want to write more functions than we provide in the skeleton file for a problem set.*

*In this problem, you will work with a small matcher system. If you have trouble with this problem, you will have a hard time learning important course material by means of problem sets.*

In this problem, you will work with the simple matcher in the `match.scm` file. You job will be to extend the matcher program with a new type of variable that can match sequences as well as single elements in a list.

The matcher is given two lists as input, one (the pattern) contains variables and the other (the data) does not, and it returns a list of variable bindings that would make the pattern match the data. We denote simple variables by a list whose first element is the symbol `?` and whose second element is the name of the variable, e.g. `(? x)` represents the variable $x$. For example:

- `(match '(a (? x) c) '(a b c))` → `(bindings (x b))`

Note that the return value is a list, whose car is the symbol bindings and whose cdr is a list of lists (possibly null). The first element of the component lists is the name of a variable and the second element of these lists is the matched value. Here are some more examples of successful matches:

- `(match '(a ((? x) c) d) '(a (b c) d))` → `(bindings (x b))`
- `(match '(a ((? x) c) (? y)) '(a (b c) c))` → `(bindings (y c) (x b))`
- `(match '(a (b c) d) '(a (b c) d))` → `(bindings)`

If a variable with the same name occurs more than once in a pattern, it must match equal parts of the data:

- `(match '(a (? x) c (? x) e) '(a b c b e))` → `(bindings (x b))`
- `(match '(a (? x) c (? x) e) '(a b c d e))` → `()`
- `(match '(a (? x) c (? y) e) '(a b c d e))` → `(bindings (y d) (x b))`

The code also supports nameless variables `(? _)`, which simply match but whose matching values are not returned and not constrained to be equal:

- `(match '(a (? _) c (? _) e) '(a b c b e))` → `(bindings)`
- `(match '(a (? _) c (? _) e) '(a b c d e))` → `(bindings)`

A simple variable can match any component of the data list, but it will not match a sublist of the data:

- `(match '(a (? x) d) '(a b c d))` → `()`

What we want to do in this problem is extend the matcher to handle a new class of variable – called "segment variables" – that can match sublists of the data:

- `(match '(a (* x) d) '(a b c d))` → `(bindings (x (b c)))`
- `(match '(a (* x) d) '(a d))` → `(bindings (x ()))`
- `(match '((* x) a (* x) c) '(a c))` → `(bindings (x ()))`

- `(match '((* x) a (* x) c) '(b a b c))` → `(bindings (x (b)))`
- `(match '((* x) a (* x) c) '(b a d c))` → `()`
- `(match '((a (* x)) d e) '((a b c) d e))` → `(bindings (x (b c)))`

Note that the binding value of a segment variable is always a list, possibly a null list. Therefore,

- `(match '((? x) a (* x) c) '(b a b c))` → `()`
- `(match '((? x) a (* x) c) '((b) a b c))` → `(bindings (x (b)))`

Note also that there is, in general, no unique assignment to the segment variables to produce a match. We just want to produce *some* assignment if one exists.

Also, we treat the case of standalone segment variables that are not part of a list, that is `(match '(* x) ...)`, as immediate failures since it is not clear what it should mean. On the other hand, `(match '(? x) ...)`, always succeeds.

Your job is to extend the matcher to support segment variables. Most of the work has already been done for you. What remains is to read and understand how the matcher works, then implement the missing functionality needed for the `match-segment-variable` function.

## 5 Survey

We are always working to improve the class. Most problem sets will have at least one survey question at the end to help us with this. Your answers to these questions are purely informational, and will have no impact on your grade.

Please fill in the answers to the following questions as definitions of the appropriate variables in `ps0.scm`:

- When did you take 6.001?
- How many hours did 6.001 projects take you?
- How well do you feel you learned the material in 6.001?
- How many hours did this problem set take you?