6.034 Artificial Intelligence, Fall 2006
Prof. Patrick H. Winston

Problem Set 0

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 DrScheme. 6.034 uses DrScheme 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 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, 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 resources

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

- Your trusty 6.001 book, *Structure and Interpretation of Computer Programming*, available online
- The standard Scheme documentation, which is called "R5RS"
- Course 6/HKN tutoring program

1. 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 these commands:

```
add 6.034
/mit/6.034/bin/6.034-setup
```

This script creates a `6.034-psets` directory in your Athena home directory, with a subdirectory for each problem set, and sets the permissions on them so that your TA can access the files in it. Changing these permissions may affect your grade.

In order to get credit for a problem set, you **must** copy the files containing any code you wrote into the appropriate directory. This is how you submit your solutions.

1.1. DrScheme

There are many versions of Scheme out there. 6.001 used different ones in different places. In order for the code in this class to work, though, we need to standardize on one version of Scheme.

This class will be done entirely using the DrScheme environment. You may have used DrScheme to do your projects in 6.001.

You should **not** use 6.001 Scheme or MIT Scheme to do the problem sets! If you do, you will get strange error messages, and the TAs will not be able to help you.

You run DrScheme on Athena like this:

```
add drscheme
drscheme &
```

You can also download it to run it on your own computer.

Note: If you're running a version of DrScheme you already have, or which is provided to you by a Linux distribution like Ubuntu, be sure it's at least version 350 (the current version you can download is 352). Older versions won't be able to run the tester code.

"The great thing about standards is there are so many to choose from."

If you're running DrScheme for the first time, it will ask you to choose a dialect of Scheme. We're going to use PLT Scheme, also known as MzScheme. To set this up:

- Go to the **Language** menu and select **Choose Language...**
- In the list of languages, click on **PLT**.
- A few variants will drop down under PLT. Choose the **Textual** variant.

1.2. Getting the problem set code

The code can be downloaded from the assignments section.

1.3. Answering questions

The main file of this problem set is called `ps0.scm`. Open that file in DrScheme. The file contains a lot of (define) statements that you need to fill in with your solutions.

The first thing to fill in is a multiple choice question. The answer should be extremely easy. Many problem sets will begin with some simple multiple choice questions to make sure you're on the right track.

1.4. Run the tester

Every problem set comes with a file called `tester.scm`. This file checks your answers to the problem set. For problems that ask you to provide a function, the tester will test your function with several different inputs and see if the output is correct. For multiple choice questions, the tester will tell you if your answer was right. Yes, that means that you never need to submit wrong answers to multiple choice questions.

- Open the file `tester.scm` in DrScheme and click "Run".
- It should output the results of a lot of tests in your DrScheme window. You should pass one test (your answer to the multiple choice question), and fail the others, because you haven't solved those problems yet.

You should run the tester early and often, and definitely make sure you pass all the tests before you submit a problem set. Think of it as being like the "Check" button from 6.001. It makes sure you're not losing points unnecessarily.

2. Scheme programming

Now it's time to write some Scheme.

2.1. Warm-up stretch

Write the following functions:

- `(cube n)`, which takes in a number and returns its cube. For example, (cube 3) => 27.
- `(factorial n)`, which takes in a non-negative integer *n* and returns *n!*, which is the product of the integers from 1 to *n*. (0! = 1 by definition.)

  We suggest that you should write your functions so that they raise nice clean errors instead of dying messily when the input is invalid. For example, it would be nice if factorial rejected negative inputs right away; otherwise, you might loop forever. You can signal an error like this: `(error "factorial: input must not be negative")`
  Error handlng doesn't affect your problem set grade, but on later problems it might save you some angst when you're trying to track down a bug.

- `(count-pattern pattern lst)`, which counts the number of times a certain pattern of symbols appears in a list, including overlaps. So `(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.

2.2. 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`

2.3. 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, represented by this Scheme list: `(((1 2) 3) (4 (5 6)) 7 (8 9 10))`

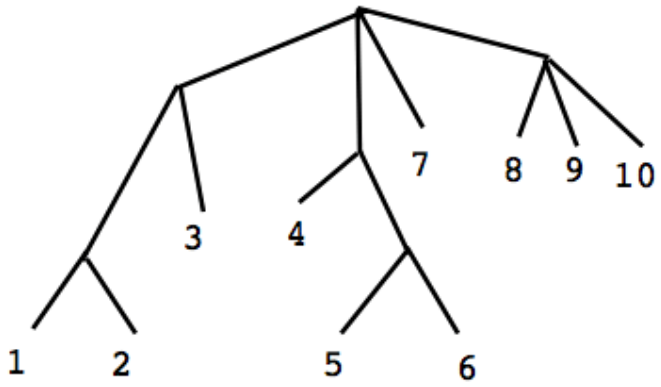

Figure 1: Example Tree

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. So `(tree-ref tree (list 0))` should return `((1 2) 3)`.

3. 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 implement a simple pattern matching system, of the kind that is used in the rule-based expert systems you'll learn about next week.

The matcher compares the corresponding pieces of two lists (which may have other lists inside them -- that is, they may be trees). In the first list, called the *pattern*, some of these pieces may be *variables* -- they can match any value as long as they do so consistently. The other list has no variables, and is called the *datum*.

When a variable is matched to a value, that value needs to be stored to make sure that later instances of that variable match the same value. These connections between variables and values are stored in a structure called the *bindings*, and variables that have been matched are said to be *bound*.

The matcher is outlined in the `match.scm` file. The parts that are left for you to write are:

- Matching variables against their corresponding values, and adding the appropriate bindings.
- Matching entire lists by ensuring that their corresponding pieces match

The code for this matcher is in `match.scm`; you will finishing it by completing the two functions at the end of the file, `match-variable` and `match-lists`.

The code starts by defining some abstractions:

- A **variable expression** indicates that a certain part of the pattern is a variable. It's represented by a two element list whose first element is the symbol `?` and whose second element is the variable's name. So the variable *x* has the expression `(? x)`.
- A **binding** gives a value for a variable. It's represented by a two-element list, whose first element is the variable's name (not the whole expression) and whose second element is the value. So if the variable *x* gets bound to the list `(3 4)`, then the binding that says so looks like this: `(x (3 4))`
- A **set of bindings** stores multiple bindings, associating multiple variables with their values. When a match succeeds, it returns the set of bindings that makes it work. A set of bindings has the symbol `bindings` as its first element, and the rest of the list consists of the bindings themselves.

  An example of a set of bindings looks like this: `(bindings (x (3 4)) (y b) (z 3))`. The variables are named *x*, *y*, and *z*, and their respective values are `(3 4)`, the symbol `b`, and the number `3`.

Some examples of matching:

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

The tester will make sure that your code matches these test cases and some other ones. The tests we run after you submit your problem set will be fairly similar. We don't intend to surprise you with "trick test cases", so don't worry.

4. 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 in the definitions at the end of 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?

5. When you're done

Remember to run the tester, and then to copy your code into your `6.034-psets/ps0` directory on Athena if you haven't already.

6. Errata

- Wednesday, Sept. 6: One of the test cases was wrong. On line 44 of `tester.scm`, `(? x)` was typoed as `(?x)`. Also, one function to fill in was named `match-variables` when the rest of the code expects it to be named `match-variable`. The code has now been fixed. If you downloaded the problem set on Wednesday, you can re-download `tester.scm` and `match.scm`, or you can just fix the typoes.

- Thursday, Sept. 7: Test case 27 was too picky; it should accept the bindings no matter what order they're in. You can ignore that test case, or you can download the new `tester.scm` and `tester-utils.scm`.
- Sunday, Sept. 10: In one place, the problem set said that 6.034 uses MIT Scheme. It should have said DrScheme.