

EECS 345: Programming Language Concepts

Programming Project, Part 1

Due Friday, February 12

For this and all programming project's, you are welcome to work in groups of up to three. The names of all group members should appear at the top of the file, and every member should submit the project on blackboard. All team members are responsible for understanding the code submitted in their name.

In this homework, you are to create an interpreter for a very simple Java/C-ish language. The language has variables, assignment statements, mathematical expressions, comparison operators, boolean operators, if statements, while statements, and return statements.

An example program is as follows:

```
var x;  
x = 10;  
var y = 3 * x + 5;  
while (y % x != 3)  
    y = y + 1;  
if (x > y)  
    return x;  
else if (x * x > y)  
    return x * x;  
else if (x * (x + x) > y)  
    return x * (x + x);  
else  
    return y - 1;
```

Note that braces, { and }, are not implemented.

The following mathematical operations are implemented : +, -, *, /, % (including the unary -), the following comparison operators are implemented: ==, !=, <, >, <=, >=, and the following boolean operators: &&, ||, !. Variables may store values of type `int` as well as `true` and `false`. You do not have to detect an error if a program uses a type incorrectly, but it is not hard to add the error check.) Note that you do not have to implement short-circuit evaluation of && or ||.

For those seeking an extra challenge: The parser supports nested assignment statements as well as assignments inside expressions. Try writing your interpreter so that assignment operators return a value as well as initialize a variable:

```
var x;  
var y;  
x = y = 10;  
if ((x = x + 1) > y)  
    return x;  
else  
    return y;
```

General guidelines

You are to write your interpreter in Scheme using the functional programming style. For full marks, you should not use variables, only functions and parameters.

Your program should clearly distinguish, by naming convention and code organization, functions that

are doing the `M_state` operations from ones doing the `M_value` and `M_boolean` operations. You do not have to call them `M_`, but your naming convention should be consistent.

A parser is provided for you called `simpleParser.scm`. You will also have to get the file `lex.scm`. You can use the parser in your program by including the line `(load "simpleParser.scm")` at the top of your homework file. The command assumes `simpleParser.scm` is in the same directory as your homework file. If it is not, you will have to include the path to the file in the load command.

To use the parser, type the code into a file, and call `(parser "filename")`. The parser will return the parse tree in list format. For example, the parse tree of the above code is:

```
((var x) (= x 10) (var y (+ (* 3 x) 5)) (while (!= (% y x) 3) (= y (+ y 1))) (if (> x y) (return x) (if (> (* x x) y) (return (* x x)) (if (> (* x (+ x x)) y) (return (* x (+ x x))) (return (- y 1)))))
```

Formally, a parse tree is a list where each sublist corresponds to a statement. The different statements are:

variable declaration	<code>(var variable)</code> or <code>(var variable value)</code>
assignment	<code>(= variable expression)</code>
return	<code>(return expression)</code>
if statement	<code>(if conditional then-statement optional-else-statement)</code>
while statement	<code>(while conditional body-statement)</code>

You should write a function called `interpret` that takes a filename, calls `parser` with the filename, evaluates the parse tree returned by `parser`, and returns the proper value. You are to maintain a state for the variables and return an error message if the user attempts to use a variable before it is declared. You can use the Scheme function `(error ...)` to return the error.

The State

Your state needs to store binding pairs, but the exact implementation is up to you. I recommend either a list of binding pairs (for example: `((x 5) (y 12) ...)`), or two lists, one with the variables and one with the values (for example: `((x y ...) (5 12 ...))`). The first option will be simpler to program, but the second will be more easily adapted supporting objects at the end of the course. The exact way you decide to implement looking up a binding, creating a new binding, or updating an existing binding is up to you. It is not essential that you be efficient here, just do something that works. With such a simple language, an efficient state is unneeded.

What you *do* have to do is use abstraction to separate your state from the rest of your interpreter. As we increase the number of language features we have in future parts of the project, we will need to change how the state is implemented. If you correctly use abstraction, you will be able to redesign the state without changing the implementation of your interpreter. In this case, that means that the interpreter does not know about the structure of the state. Instead, you have generic functions that the interpreter can call to manipulate the state.

Finally...

If you are using DrRacket, you will probably need to change the language to one of the more advanced teaching languages (if you want more descriptive error messages). The language "Pretty Big" will work, but it does not give very informative error messages.

Please save your interpreter as a Scheme file with either the `.scm` or `.rkt` extension.