For this and all programming project's, you are to work in groups of up to three students. The names of all group members should appear at the top of the file, and every member should join the same group on Canvas. All team members are expected to contribute to the project, and all students are responsible for understanding the code submitted in their name. If you insist on working alone, you must still join a group of size one.

The Language

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: &&, ll, !. 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). You do not have to implement short-circuit evaluation of && or ||, but you are welcome to do so.

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;
```


Sample Programs

Here are some more sample programs in this simple language that you can use to test your interpreter. Please note that these programs cover most of the basic situations, but they are *not* sufficient to completely test your interpreter. Be certain to write some of your own to fully tests your interpreter.

part1tests.html

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.

You should use good style, indentation and proper commenting so that the code you submit is easy to read and understand.

Using the Parser

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.

If you are using racket instead of scheme, you should use (require "simpleParser.scm") at the top of your file instead of the load function, and you will need to uncomment some lines in the simpleParser.scm and lex.scm files.

To parse a program in our simple language, type the program 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)) (return (+ 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	(var *variable*) or (var *variable value*)
assignment	(= *variable expression*)
return	(return *expression*)
if statement	(if *conditional then-statement optional-else-statement*)
while statement	(while *conditional body-statement*)

Your Interpreter Program

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.

Returning a Value

Your interpreter needs to return the proper value. How you achieve this is up to you, and we will later learn the proper way to handle the return. A simple solution that is sufficient for this project is to have a special variable called *return* in the state that you assign to the return value. When you are done interpreting the program your code can then lookup *return* in the state, get, and return that value.

Finally...

If you are using DrRacket, you will need to choose an appropriate language. The parser will run in both R5RS Scheme and PrettyBig. If you are using racket, you will need to uncomment a few lines in both the simpleParser.scm and lex.scm files.

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

Rubric

Interpreter Part 1

Interpreter Part 1							
Criteria	**Ratings**						
This criterion is linked to a Learning OutcomePerformance	50.0 pts Full Marks *Works on all operators and statements. Clearly delineates state functions, "M_state" functions, and "M_value" functions. The value functions correctly return the value of an expression. The state functions correctly determine the new state. State functions correctly insert, update, and lookup.*	45.0 pts Great *Same as 50 with only minor errors.*	40.0 pts Good *Clearly separated state, "M_state", and "M_value" functions that correctly return states and values. The functions either have minor errors throughout or significant errors or omissions that only hurt a few cases.*	30.0 pts Okay *The code shows some understanding of the differences between state, "M_state", and "M_value" functions but there are significant errors that cover many cases.*	20.0 pts Poor *While there is some understanding of interpreter structure, the interpreter is mixing up where a state should be returned and where a value should be returned. There are significant errors that affect most cases.*	10.0 pts Minimal *Does not demonstrate an interpreter structure but has some thing such as a state or basic expressions with no variables*	0.0 No Ma

Interpreter Part 1

Criteria	Ratings			
This criterion is linked to a Learning OutcomeAbstraction	5.0 pts Good Abstraction *Uses abstraction through out.*	4.0 pts Good abstraction but the initial state *Uses abstraction throughout but hardcodes '() or '(()()) for the state instead of an abstraction*	2.0 pts Missing some abstraction *Accessing elements of the statements uses cars and cdrs instead of well-named functions.*	0.0 pts Over use of car/cdr/ *Accessing the state the M_ functions us cars and cdrs instea good abstraction*
This criterion is linked to a Learning OutcomeFunctional Coding	30.0 pts Excellent functional style	24.0 pts Good functional style *Mostly uses good functional style, but occasionally does things that are not purely functional such as using let or begin (except for handling the side effect challenge)*	20.0 pts Mostly functional *Uses the functional style, but also has very non-functional coding such as set!, global variables, or define used other than to name a function.*	15.0 pts Poor functional style *The code uses an iterative style throughout such as a list of statements executed sequentially.*	0.0 pts Violates functional co *Significant u of set!, defin inside of functions, gl variables, o anything els that is gross non-function*
This criterion is linked to a Learning OutcomeReturn values	10.0 pts Full Marks *Correctly returns integers or true and false.*	7.0 pts Close *Returns values but either may return fractions or floating point instead of integers or returns #t and #f instead of true and false.*	3.0 pts Returns the wrong thing *Returns a state instead of a value*	0.0 pts No Marks *Does not return a va or a state*
This criterion is linked to a Learning OutcomeReadibility	5.0 pts Full Marks *Nicely readible code: good indentation, well organized functions, well named*	3.0 pts Reasonable *Reasonably readible code. Except for a few places there is good commenting,*	0.0 pts No Marks *Hard to read and follow the code due to poor organizat or indentation, poorly name*	

Interpreter Part 1			
Criteria	**Ratings**		
	functions and parameters, clear comments.	*organization, indentation, short lines, and good naming.*	*functions or parameters, a a lack of commenting.*
Total Points: 100.0			

Submission

Turned In!

Feb 18 at 10:30pm
Submission Details
Download interpreter.scm
Grade: 97 (100 pts possible)
Graded Anonymously: no

View Rubric Evaluation

Comments:
No Comments