Interpreter, Part 3

Due Apr 1 by 11:59pm **Points** 100 **Submitting** a file upload

*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. You do **not** have to keep the same group as the previous interpreter parts.*

Solutions to Part 2

Here is solution code for the interpreter, part 2. These solutions do not use boxes and do not support side effects. They are the same except that one has the M_state functions tail recursive (but not the M_value functions) and uses (lambda (v) v) type continuations, and the other uses "normal" recursion and call/cc for the continuations.

Both solutions are written to work with racket. If you are using scheme instead of racket, you need to remove #lang racket from the top of the file and change to (load "simpleParser.scm") the (require "simpleParser").

interpreter2-callcc-no-boxes.rkt

interpreter2-tail-recursion-no-boxes.rkt

A New Parser

This interpreter needs a new parser: functionParser.rkt

As with the previous parser, this one is written for racket, and you will need to comment/uncomment some lines to use it with scheme.

The same lex.scm file will work with the new parser.

The Language

In this homework, you will expand on the interpreter of part 2 adding function definitions. We still assume all variables store integers and boolean. Likewise, all functions will only return integers and boolean.

While normal C does not allow nested functions, the gcc compiler *does* allow nested functions as an extension to C, so let's implement them!

For those seeking a small extra challenge: try implementing both the *call-by-reference* and the *call-by-value* parameter passing styles.

An example program that computes the greatest common divisor of two numbers is as follows:

```
var x = 14;
var y = 3 * x - 7;
function gcd(a,b) {
  if (a < b) {
    var temp = a;
    a = b;
    b = temp;
  }
  var r = a % b;
  while (r != 0) {
    a = b;
    b = r;
    r = a % b;
  }
  return b;
}
function main () {
  return gcd(x,y);
}
```

Here is another example program that uses recursion:

```
function factorial (x) {
  if (x == 0)
    return 1;
  else
    return x * factorial(x - 1);
}

function main () {
  return factorial(6);
}
```

Note that only assignment statements are allowed outside of functions. Functions do not have to return a value. The parser will have the following additional constructs:

```
function a(x, y) {          =>    (function a (x y) ((return (+ x y)))
  return x + y;
}

function main () {          =>    (function main () ((var x 10) (var y 15) (return (funcall gcd x y))))
  var x = 10;
  var y = 15;
  return gcd(x, y);
}
```

The final value returned by your interpreter should be whatever is returned by main.

Nested functions can appear anywhere in the body of a function. Any name in scope in a function body will be in scope in a function defined inside that body.

```
function main() {
  var result;
  var base;

  function getpow(a) {
     var x;

     function setanswer(n) {
        result = n;
     }

     function recurse(m) {
       if (m > 0) {
         x = x * base;
         recurse(m-1);
       }
       else
         setanswer(x);
     }

     x = 1;
     recurse(a);
  }
  base = 2;
  getpow(6);
  return result;
}
```

Function calls may appear on the right hand side of global variable declaration/initialization statements, but the function (and any functions that function calls) must be defined before the variable declaration. Otherwise, functions that are used inside other functions do not need to be defined before they are used.

If you want the additional challenge, we will use a similar style as C++ for call-by-reference:

```
function swap(&x, &y) {     =>  (function swap (& x & y) ((var temp x) (= x y) (= y temp)))
  var temp = x;
  x = y;
  y = temp;
}
```

It is an error to use call-by-reference on anything other than a variable. For example, if the program contains swap(x, x + 10) with the above definition of swap, you should give an error because x + 10 is not a variable.

Sample Programs

Here are some 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 test your interpreter. **In particular, there are no tests here using boolean values. Make sure your functions can take booleans as inputs and return booleans.**

part3tests.html

What your code should do

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 returned by main. You are to maintain an environment/state for the variables and return an error message if the program attempts to use a variable before it is declared, attempts to use a variable before it is initialized, or attempts to use a function that has not been defined.

Some hints

Terminology In this interpreter, we will be talking about *environments* instead of *states*. The state consists of all the active bindings of your program. The environment is all the active bindings that are in scope.

1. Note that the base layer of your state will now be the global variables and functions. You should create an outer "layer" of your interpreter that just does M_state functions for variable declarations and function definitions. The declarations and assignments should be similar to what you did in your part 2 interpreter. The functon definitions will need to bind the function closure to the function name where the closure consists of the formal parameter list, the function body, and a function that creates the function environment from the current environment.

2. Once the "outer" layer of your interpreter completes, your interpreter should then look up the main function in the state and call that function. (See the next step for how to call a function).

3. You need to create a M_value function to call a function. This function should do the following: (a) create a function environment using the closure function on the current environment, (b) evaluate each actual parameter in the current environment and bind it to the formal parameter in the function environment, (c) interpret the body of the function with the function environment. Note that interpreting the body of the function should be, with one change, *exactly* what you submitted for *Interpreter, Part 2*. Also note that if you are using boxes, you should not have to do anything special to deal with global variable side effects. If you are not using boxes, you will need to get the final environment from evaluating the function body and copy back the new values of the global variables to the current environment/state.

4. Change the M_state and M_value functions for statements and expressions, respectively, to expect function calls.

5. Test the interpeter on functions without global variables, and then test your functions using global variables. One tricky part with the functions is that, unlike the other language constructs we have created, function calls can be a statement (where the return value is ignored), and an expression (where the return value is used). You need to make sure both ways of calling a function works.

6. Since exceptions can happen anywhere that a function call can occur, you may discover more places that need the throw continuation. If you used call/cc for throw, then you should only need minimal modifications from what you did in your interpreter from part 2. If you used tail recursion for throw, you will need to make the M_value functions tail recursive for throw to work correctly.

Interpreter Part 3

Criteria	Ratings						Pts
Abstraction	**5.0 pts** **Good Abstraction** Uses abstraction throughout.	**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/cons** Accessing the state in the M_ functions uses cars and cdrs instead of good abstraction			5.0 pts
Functional Coding	**15.0 pts** **Excellent functional style**	**12.0 pts** **Good functional style** Mostly uses good functional style, but overuses let or begin.	**10.0 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. (set-box! is allowed for the state values)	**8.0 pts** **Poor functional style** The code uses an iterative style throughout such as a list of statements executed sequentially.	**0.0 pts** **Violates functional coding** Significant use of set!, define inside of functions, global variables, or anything else that is grossly non-functional.		15.0 pts
Readibility	**5.0 pts** **Full Marks** Nicely readible code: good indentation, well organized functions, well named functions and parameters, clear comments.	**3.0 pts** **Reasonable** Reasonably readible code. Except for a few places there is good commenting, organization, indentation, short lines, and good naming.	**0.0 pts** **No Marks** Hard to read and follow the code due to poor organization or indentation, poorly named functions or parameters, and/or a lack of commenting.				5.0 pts
M_value for functions	**25.0 pts** **Full marks** (a) Correctly creates new environment for the function that implements static scoping. (b) Correctly evaluates and binds the parameters. (c) Creates the proper continuations for the function call. (d) Evaluates the function body, and (d) handles return correctly.	**23.0 pts** **Excellent** Has all the necessary parts and logic, but a few small errors.	**21.0 pts** **Good** Has all the necessary parts, but there is a significant error with one of the parts. For example, does not have static scoping, evaluates the parameters in the wrong environment, or does not set up the continuations correctly.	**16.0 pts** **Poor** Has most of the necessary steps, but has significant errors in multiple steps, or is completely missing one of the necessary steps.	**10.0 pts** **Minimal** Some valid logic implementing a function call, but none of the necessary steps are correct.	**0.0 pts** **No Marks** No reasonable attempt at creating a M_value for function calls.	25.0 pts
Functions in expressions and statements	**10.0 pts** **Full Marks** Function calls work correctly as both statements and as expressions. The environment is updated correctly in all cases.	**9.0 pts** **Very good** Function calls are implemented for both statements and expressions, but the interpreter is missing a place where function calls can occur.	**8.0 pts** **Good** Function calls are implemented for statements and expressions, but there is a significant error such as the return value not being ignored when the function call is a statement.	**7.0 pts** **Poor** Function calls are implemented in only one place.	**3.0 pts** **Minimal** Function calls are implemented, but not at the correct place in the code.	**0.0 pts** **No Marks** Does not have function calls implemented.	10.0 pts
Interpreter "layers"	**15.0 pts** **Full Marks** The interpreter has two "layers" with the outer layer reading in global variables and function definitions, the interpreter looks up and executes main, and returns the value. The "inner layer" correctly handles function bodies including nested functions.	**14.0 pts** **Very Good** The interpreter has "layers", looks up and runs main after running the "outer layer", deals with nested functions in the "inner layer", but there are small errors.	**12.0 pts** **Good** The interpreter has layers that separate handling the global variables and function definitions from interpreting the function bodies, but there are significant errors.	**10.0 pts** **Poor** There is some attempt at dividing the interpreter into layers, but there are things that should be done in one layer that are missing or done in the wrong layer or done in both layers.	**5.0 pts** **Minimal** The interpreter does not have layers. Instead the same M_state recursion is used for global variables and function definitions as it is for function bodies. The interpreter does lookup and run main, though it may not run main correctly.	**0.0 pts** **No Marks** The interpreter has only one layer. The interpreter does not find and run the main function.	15.0 pts
M_state for function definitions	**10.0 pts** **Full Marks** The function name is correctly placed in the state with a correct closure.	**9.0 pts** **Excellent** The function name is bound to a correct closure in the state, but the routine has a small error in the M_state function.	**7.0 pts** **Okay** The function name is bound to a function closure in the state, but the closure does not correctly set the scope for nested functions.	**5.0 pts** **Poor** The function name is bound to a closure in the state, but the closure is not correct.	**0.0 pts** **No Marks** The function name is not bound to a closure in the state.		10.0 pts
Loops, conditionals, etc.	**5.0 pts** **Full Marks** Loops, conditionals, assignment all still work correctly.	**3.0 pts** **Some mistakes** Separated the function code from the interpreter part 2 code, but something done broke a M_state or M_value from part 2 of the interpreter.	**0.0 pts** **No Marks** Did not successfully separate the function implementation from the rest of the language features, and now much is broken.				5.0 pts
Global variables	**5.0 pts** **Full Marks** Global variables are correctly modified and updated when used in a function.	**3.0 pts** **Uses but does not update globals** Global variables are used, but the values are not correctly updated and maintained.	**0.0 pts** **No Marks** The functions cannot use global variables.				5.0 pts
Throw/catch	**5.0 pts** **Full Marks** Throw can correctly work across functions.	**3.0 pts** **Good** Throw correctly exits functions, but the environment in the catch or finally is not correct.	**0.0 pts** **No Marks** Throw does not leave the function (for example, failed to pass the throw continuation where needed or failed to make M_value tail recursive).				5.0 pts

Total Points: 100.0