Interpreter, Part 2

Submit Assignment

Due Friday 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.

In this homework, you will expand on the interpreter of part 1 adding code blocks as well as "goto" type constructs: break, continue, (true) return, and throw. We still assume all variables store either an integer or a boolean value. *For those wanting an extra challenge*: you are to again assume that expressions can have side effects. Specifically, you should assume that any expression can include an assignment operator that returns a value.

Please note: a portion of your grade in this project will be correcting the errors you had in Part 1.

The Language

The parser you used in part 1 supports all of the language features used in this assignment. Here are the new language constructs you need to implement:


```
break;              =>   (break)
continue;           =>   (continue)
throw e;            =>   (throw e)

if (i < j) {        =>   (if (< i j) (begin (= i (+ i 1)) (= j (+ j 1))))
  i = i + 1;
  j = j - 1;
}

try {                      =>   (try body (catch (e) body) (finally body))
  body
}
catch (e) {
  body
}
finally {
  body
}
```

Note that either the finally or the catch block may be empty:

```
try {                    =>  (try body (catch (e) body) ())
  body
}
catch (e) {
  body
}
```

Please note:

- As with C and Java, a block of code can appear anywhere and not only as the body of an if statement or a loop.
- As with C and Java, the break and continue apply to the immediate loop they are inside. There are no labels in our interpreter, and so there will be no breaking out of multiple loops with one break statement.
- As there is no type checking in our language, only one catch statement per try block is allowed.

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.

part2tests.html

General Guidelines

You do not have to stick to strict functional programming style, but you should avoid global variables and heavy use of let because they will make your life harder. You also should not use set! (except for the recommended state change below).

As with the last project, 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.

Also as before, the launching point of your interpreter should be 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 initialized.

Implementing the "Goto" constructs

You need to use continuations to properly implement return, break, continue, and throw. For each, you have two options. You can make your interpreter tail-recursive with continuation passing style (note that for this version only the M_state functions must be tail recursive, but you will need the M_value and M_boolean functions tail recursive in part 3 of the interpreter) or you can use call/cc. Both techniques are equally challenging. You are also welcome to use cps for some of the constructs and call/cc for others.

The Program State

To implement blocks, you need to make the following **required** change to the state/environment. In addition, because this interpreter does not require a lot of new features from the previous one, there is a **recommended** change to the state that may help reduce the work required when we get to Part 3 of the interpreter.

The required change: Your state must now be a list of *layers*. Each layer will contain a list of variables and bindings similar to the basic state of part 1. The initial state consist of a single layer. Each time a new block is entered, you must "cons" a new layer to the front of your state (but use abstraction and give the operation a better name than "cons"). Each time a variable is declared, that variable's binding goes into the top layer. Each time a variable is accessed (either to lookup its binding or to change it), the search must start in the top layer and work down. When a block is exited, the layer must be popped off of the state, deleting any variables that were declared inside the block.

A reminder about a note from part 1: 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 recommended change: In Part 3 of the interpreter, you will need to implement function/method calls and global variables. Thus, even if you are not doing the extra coding challenge, you will need to handle functions that produce side effects. If you would like a simpler way to deal with side effects, I recommend the following break from strict functional style coding. Instead of binding each variable to its value, we will bind the variable to a *box* that contains its value. You can think of a box as a pointer to a memory location, and thus the values stored in the environment will be pointers to the actual data (similar to how Java implements non-primitive types). Using boxes, you will not need separate M_value and M_state functions for handling function calls. Instead, the function/method call M_value mapping will be able to change the values of global variables. The Scheme commands are:
(box v): places *v* into a *box* and returns the box
(unbox b): returns the value stored in box *b*
(set-box! b v): changes the value stored in box *b* to value *v*.
Note that the set-box! command does not return a value. You should embed it in a begin function. Scheme begin takes one or more expressions and returns the value of the last expression. For example, (begin (set-box! b v) #t) will return #t.

Interpreter Part 2

Criteria	Ratings						Pts
Part 1 Performance	20.0 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.*	16.0 pts Good *Clearly separated state, "M_state", and "M_value" functions that correctly return states and values. The functions either have minor errors.*	12.0 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.*	8.0 pts Poor *While there may be 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.*	0.0 pts No Marks		20.0 pts
Abstraction	5.0 pts Good Abstraction *Uses abstraction through out.*	4.0 pts Good abstraction but hardcodes 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	20.0 pts Excellent functional style	16.0 pts Good functional style *Mostly uses good functional style, but overuses let or begin.*	13.0 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)*	10.0 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.*		20.0 pts
Return continuation	10.0 pts Full Marks *Correct creation of a continuation for return. Either call/cc or tail recursion is used. The continuation is used everywhere. The proper value is returned.*	9.0 pts Excellent *Same as 10 but a couple minor errors like typos, missing continuations, or functions that are not tail recursive.*	8.0 pts Good *Uses a continuation for return, but the wrong continuation is used, the continuation is missing in several places where it is needed, multiple M_state functions are not tail recursive, or the call/cc is in the wrong location.*	7.0 pts Reasonable *A correct return that returns a state instead of a value or a continuation return with significant problems with the continuations or the tail recursion.*	4.0 pts Minimal *Some attempt at implementing a continuation for return, but does not demonstrate understanding of how to use continuations. For example, call/cc used many places instead of the correct place, or a normal continuation created but tail recursion is not correctly done anywhere.*	0.0 pts No Marks *Does not use a continuation for return.*	10.0 pts

Criteria	Ratings						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
State Layers	10.0 pts Full Marks *The state now has layers. The new variables only go in the top layer. The look up traverses all the layers in order and returns/updates the first match found.*		7.0 pts Reasonable *The state has layers, and they mostly work, but there are some minor errors that cause a variable not to be found or scope to not be properly implemented.*		4.0 pts Minimal *The state has layers, but there are significant errors that prevents proper scoping in the interpreter.*	0.0 pts No Marks *No attempt to add layers to the state.*	10.0 pts
Blocks	10.0 pts Full Marks *When entering a block a layer is added to the state, and the layer is removed when leaving. The block code is separate from the other M_state functions. The code works correctly in all situations.*	9.0 pts Excellent *Same as 10 but did not properly modify the state for one way the execution can leave the block.*	8.0 pts Good *Implements blocks separate from the other M_state functions. Demonstrates the proper way to add and remove the top layer when entering and leaving, but multiple places are not implemented correctly.*	7.0 pts Reasonable *Implements blocks but does not add/remove layers in most of the ways execution can enter or leave. Or implements the block in the M_state of other statement types like while or if.*	4.0 pts Minimal *Some attempt to implement code blocks but significant errors such as failing to pop the top layer in any way that the execution can leave the block.*	0.0 pts No Marks *No attempt to implement code blocks.*	10.0 pts
Break and continue	10.0 pts Full Marks *Correct creation of a continuations for break and return. Either call/cc or tail recursion is used. The continuations are created in the M_state function for while loop. The continuations are included everywhere they are needed. The*	9.0 pts Excellent *Same as 10 but a couple minor errors like typos, missing continuations, or functions that are not tail recursive*	8.0 pts Good *Uses a continuation for break and continue, the continuations are created in the correct place in the code, and the proper use of tail recursion or call/cc, but there are errors: mistakes in the continuation, multiple places missing tail*	7.0 pts Reasonable *Continuations for break and return but there are significant errors such as the continuations are not created in the correct location, the continuations do not return the proper thing, there is a significant number of places with missing*	4.0 pts Minimal *Some attempt at implementing a continuation for break or continue, but does not demonstrate understanding of how to use continuations. For example, call/cc used many places instead of the correct place, or a normal continuation created but tail recursion*	0.0 pts No Marks *No attempt to use continuations for break or continue.*	10.0 pts

Criteria	Ratings						Pts
try/catch	10.0 pts Full Marks *A correct continuation for throw is created using call/cc or tail recursion. The continuation is used everywhere needed. All execution paths out of try/catch correctly implement the finally.*	9.0 pts Excellent *Same as 10 but a couple small errors such as a few missing continuation locations, missing tail recursion, finally does not execute on a couple of execution paths.*	8.0 pts Good *Uses a continuation for throw, the continuation is created in the correct place in the code with proper use of tail recursion or call/cc, but there are errors: such as catch works but finally does not, significant missing tail recursion, or a significant number of places that are missing the continuation.*	7.0 pts Reasonable *Continuation throw is created along with code for try/catch but there are significant errors such as the continuation not created in the correct location, the continuation does not return the proper thing, there is a significant number of places with missing continuations or missing tail recursion.*	4.0 pts Minimal *Some attempt at implementing try/catch, but does not demonstrate understanding of how to use continuations. For example, call/cc used many places instead of the correct place, or a normal continuation created but tail recursion is not correctly done anywhere, or the continuation shows no understanding of how try/catch works.*	0.0 pts No Marks *No attempt to use a continuation for throw or no attempt to write the M_state for try/catch.*	10.0 pts
						Total Points: 100.0	