EECS 345: Programming Language Concepts

Interpreter Project, Part 2

Due Friday, March 4

For this and all Interpreter Project's, you are welcome to work in groups, but each person is expected to submit and be responsible for their own interpreter.

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. and continue and blocks. 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 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.

Style 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 only the M_state functions must be tail recursive) 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`.