CS428/584 Final Project: *C--* **Due 12/17**

NOTE: You may work in pairs on this project.

Implement an interpreter for the C-- language: a statically-scoped block structured language that supports functional-style programming, calls by value and reference, and limited forms of higher-order function.

C-- Grammar: (http://www.mathcs.emory.edu/~jlu/cs428/c--/)
(Note: The plural form of any non-terminal N means 0 or more occurrences of N.)

Program → Definitions “main” “(“ “)”

Definition → “var” ID ”:” Expression “;”
Definition → “fun” ID “(“ Formals “)” ”:” Expression “;”
Definition → “proc” ID “(“ Formals “)” Block “;”
Formal → ID | “var” ID

Expression → Term Expression’
Expression’ → Wop Term Expression Expression’ | ε
Term → Factor Term’
Term → Sop Factor Term Term’ | ε
Factor → ID | Num | “(“ Expression “)” | FCall | Ternary | “-” ID
Ternary → “if” “(“ Boolean “)” Expression “else” Expression
FCall → ID “(“ Actuals “)”
Actual → Expression

Sop → * | /
Wop → + | -

Boolean → Expression Cop Expression
Cop → < | <= | > | >= | != | ==

Statement → Assignment | While_Loop | IF_Stmt | Block | PCall | Print | Break

Assignment → ID “=” Expression “;”
While_Loop → “while” “(“ Boolean “)” Statement
IF_Stmt → “if” “(“ Boolean “)” Statement
IF_Stmt → “if” “(“ Boolean “)” Statement “else” Statement
Block → “{“ Local_Defs Statements “}”
PCall → ID “(“ Actuals “)” “;”
Print → “print” Expression “;”
Break → “break” “;”

Local_Def → “var” ID: Expression “;”

Remarks on Syntax:

1. ε is the “empty string”.
2. Some syntax requirements not reflected in the grammar including
    a. A formal var parameters must be passed an ID.
    b. Number of formals must agree with the number of actuals.
    c. There must be at least one procedure: “main()”.

Example Program:

```
// Given an “input” n, the following program prints the number of totatives of n (a totative
is
// a positive integer, less than n, that is relatively prime to n).

        fun gcd(p,q): if (q == 0): p else: gcd(q, p%q);
        proc inc(var y) { y = y + 1; }
        proc main() {
          var count: 0;
          var n: 20;
          var i: 1;
S1:       while (i <= n) {
            var flag: gcd(i,n);
S2:         if (flag == 1)
S3:           inc(count);
S4:         inc(i);
          }
S5:       print count;
        }
S0:     main();
```


Structure of the Interpreter:

The structure combines most elements of the interpreters for FP and While, with extensions to execute *procedures*, and modifications to accommodate pass by reference and local variable definitions in blocks.

1. Procedures are abstractions for statements. Similar to functions in FP, they are operationalized via closures. To make procedures useful (beyond printing), parameters may be passed by *reference*. This enables the body of the procedure to modify the *r-values* of variables passed by the caller.
2. To model reference parameters, we need to model a variable’s value (r-value) and its reference (l-value). The is most simply done by splitting the “store” into two lists: the first (env) maps variables to references, and the second maps references to values.

3. When executing blocks, local variables may cast “shadows” on variables (and generally names) in outer scopes. One way to understand a block is as an “anonymous” procedure, and the local variables are its “parameters”. The combination of the body (i.e., the statements) of a block and its local variable declaration forms a closure that is called when execution flows to the block. The defining environment of the closure *is* also its calling environment.
4. The implication of the “block as a closure” view is that every statement should also be treated as a closure, with its own defining environment that provides a window into the “correct” references in the store. As an example, if a program fragment contains a block B followed by a statement S. The environment at the start of the fragment may be modified upon entry into a block. But the modification is only temporary -- execution of S should resume with the original environment when B completes.

At the top-level, the four main functions are eval_defs, eval_expr, eval_bool, and exec_stmt. The division between the environment and store means an extra parameter for each of these functions. For eval_expr and eval_bool, that’s almost the only change necessary.

For eval_defs and exec_stmt, changes are more extensive.

1. For eval_defs, add a new case for procedure.
2. For exec_stmt, there are new cases for PCall (procedure call) and Print. The case for Print is to simplify the process of seeing the results of computation. In pseudo Scheme:

((print? stmt) (display (eval_expr (expr_part stmt) env store defs)))

Here is a summary of the parameters.

- env: the association between variable, function, procedure names to references
- store: the association between variable references and their values
- defs: the association between function and procedure references and their closures
- rest: a list of “statement closure”s that represents the flow of program execution; this is used only by exec_stmt
- continuation: a list of statement closures that correspond to “break” points of while loops (but see discussion below)

(Optional) The number of parameters we need to pass to exec_stmt can be reduced by removing “continuation”. This was used in the interpreter for “While” to make explicit the difference between normal and abnormal flow of execution, but it isn’t necessary. Abnormal flow of execution for “break” and “continue” can be determined from the parameter “rest”, by searching for the next “while” statement.

Garbage Collection (optional):

Procedure calls and entries into blocks with local variables allocate storage that are not automatically deallocated. We can garbage collect by removing all store elements that are unreachable via identifiers in the environment. We need a function, “cleanup”, that takes an “env” and a “store”, and returns a reduced store that does not contain any key that is not the value of some pair in “env”. The code for “cleanup” and a helper function “rassoc” appears at the end.

There are different strategies for when this is done:

- Every statement: add a call to “cleanup” at the beginning of exec_stmt. (Incorrect!)
- When the “store” becomes too full. This requires some monitoring of the size of “store” and a notion of large and small.
- At strategic locations: the most reasonable places to garbage collect would be after exiting from a loop or a procedure.

```
(define cleanup
  ;; clean up the store based on the env
  (lambda (env store)
    (fold-left (lambda (lst s) (if (pair? (rassoc (car s) env)) (cons s lst) lst)) '() store)))

(define rassoc
  ;; rassoc -- reverse assoc that finds a pair based on the value of each key-value pair.
  (lambda (v alist)
    (cond ((null? alist) '())
          ((eq? (cadr (car alist)) v) (car alist))
          (else (rassoc v (cdr alist))))))
```