Worked on by Tyrone Shaffer

***** Problems *****

Does not generate intermediate code.

How to use:

First, use the lexical analyzer to process your code, written in a text file that must be named input.txt. Grammar rules will be in a separate section below. The lexical analyzer is listed as LexicalAnalyzer.c and can be run using any IDE of your personal choosing. The output will consist of three text files: lexemelist.txt, lexemetable.txt, and cleaninput.txt

lexemelist.txt: A one lined document containing all the tokens and lexemes that will be interpreted by the parser and checked for syntactical correctness.

lexemetable.txt: A table that is easier to read, listing each token and its associated lexeme for debugging purposes.

cleaninput.txt: A version of the original input file without comments.

Following that, run the Parser.c file which will take the previously mentioned lexemelist.txt as input. It will then output two additional files: mcode.txt and syntaxcheck.txt.

mcode.txt: Contains the interpreted lexemelist as machine code that will be utilized by the Virtual Machine to simulate stack processing. (Currently not outputted by Parser.txt)

syntaxcheck.txt: States errors the parser might have found or a declaration that the input is, indeed, syntactically correct. To ensure proper processing, make sure that the file states the input is syntactically correct before running the mcode generated in the virtual machine.

Lastly, run the VMAssignment.c file, which will take the specially formatted machine code in the mcode.txt file and generate one last file, stacktrace.txt.

stacktrace.txt: Shows the status of the stack following each instruction contained within mcode.txt and the end result of the program that was typed.

How to use the PL/0 language:

Programs consists of blocks that must start with a begin and end statement. The “main” part of the program is always listed after all the “functions”, named procedures, and has a period following the end to denote the end of the program. Procedure blocks, on the other hand, have their end followed by a semicolon.

Example:

var x, y; ← note: variables are contained within the scope of their block (main block for these)
procedure a;
var x, y;
begin
x := y+1;
end; ← note the semicolon at the end of the procedure block
begin
x := 1;
y := 2; ← note to assign values, it must always be through the := symbols versus simply a =
call a;
end. ← note the period at the end of the main block

const, var, and procedures all must be followed by an identifier of some sort that must begin with a letter but can be followed by a letter or number afterwards. They cannot have names more than 12 characters long.

Valid Examples:

const x1 := 2; ← note that constants must always have a numeric value and be assigned a value at declaration.

var x; ← variables may be assigned with or without a value;

procedure summation;

Invalid Examples:

const 1 := 1;

var;

procedure AddAllTheThings;

"if" statements must always be followed by a condition and a "then" block which will detail what happens after the condition is met. Else statements are optional and follow a similar pattern to a then block

Example:
if x = 2
    then ← Indented for readability.
    x := 4;
else
    then
    x := 5;

If statements may be followed by the following types of relational operations:
"="|“<>"|"<"|"<="|">"|">=“

The following mathematical operations are also supported by this compiler:
+, -, *, /

No variable can also exceed a value of 32767. There cannot be more than 100 const, var, or procedures in any given input file, and there cannot be more than 500 lines of code generated at any one time.

To call procedures, the command “call” must be followed by a valid procedure identifier.

Example:

```
procedure a;
var x;
begin
x := 4;
end;
begin
call a;
end.
```

Similar to if statements, “while” loops are expected to be followed by a “do” statement that states what occurs during each iteration until the condition set by the while is met.

Example:

```
var x;
Begin
x:= 0;
while x < 5
do
x := x+1;
end.
```