HOMEWORK #3:

Recursive Descent Parser

Due Date: **Wednesday, March the 15th, 11:59:59pm**

The **Progrotron-17** programming language is a dynamically typed, imperative programming language. The language supports 3 basic types: Integers, decimals, and strings; and has common control structures like loops and conditional statements.
Write a **Recursive Descent Parser** using the techniques used in class to recognize valid programs of the **Progrotron-17** Programming Language.

Description:

Programs of the **Progrotron-17** programming language are built from the following tokens:

Tokens:

- Integers are non-empty sequences of digits optionally preceded with either a '+' or '-' sign.
- Decimal numbers are Integers followed by a '.', followed by a non-empty sequence of digits.
- Strings are any **non-space** sequences of characters enclosed in "".
  e.g. `"hello" "abc123"`.
- Keywords are the following strings: `:=`, `+`, `-`, `*`, `/`, `OR`, `AND`, `~`, `(`, `)`, `<`, `>`, `=` , `#`, `&`, `!`, `PRINT`, `IF`, `ELSE`, `FI`, `LOOP`, `POOL`, `FUNC`, `RET`, `BEGIN`, `END.` (Notice: keywords are uppercase)
- Identifiers are sequences of digits or letters. The first character must be a letter, and an identifier cannot be a Keyword.

In **Progrotron-17** tokens are **always** separated by white-spaces.

Grammar:

Programs in the **Progrotron-17** programming language conform to the following EBNF grammar where:

- "FunctionSequence" is the start symbol.
- Terminal symbols are in **bold**.
- Collections of terminal symbols are in **blue**
- Brackets, '[' and ']' , denote an *optional* section of a rule.
- Braces, '{' and '}', denote *repetition* of a rule section (possibly 0 times).

Relation := **<** | **>** | **=** | **#**
AddOperator := **+** | **-** | **OR** | **&**
MulOperator := ***** | **/** | **AND**

Expression := SimpleExpression [Relation SimpleExpression]
SimpleExpression := Term { AddOperator Term }
Term := Factor { MulOperator Factor }
Factor := integer | decimal | string | identifier | **(** Expression **)** | **~** Factor

Assignment := identifier **:=** Expression **!**
PrintStatement := **PRINT** **(** Expression **)!**
RetStatement := **RET** identifier **!**
IfStatement := **IF** **(** Expression **)** StatementSequence [**ELSE** StatementSequence] **FI**
LoopStatement = **LOOP** **(** Expression **)** StatementSequence **POOL**

Statement := Assignment | PrintStatement | RetStatement | IfStatement |
LoopStatement
StatementSequence = Statement { Statement }

ParamSequence := identifier { **,** identifier }
FunctionDeclaration := **FUNC** identifier **(** [ParamSequence] **)** **BEGIN**
StatementSequence **END.**

FunctionSequence := FunctionDeclaration { FunctionDeclaration }

Submission Guidelines:

Submit via '`cssubmit`'. Your main file shall be called "**`progparser`**" regardless of extension. (e.g. if you are programming in C++, your main file should be called "`progparser.cpp`". If you are programming in Java your main file should be called "`progparser.java`"). Your main file should include your **name**. Include any other necessary files in your submission. If you submit a `makefile`, `make` should generate an executable called `progparser.ex`. Your program will then be tested with the command:

```
progparser.ex < inputFileName
```

Input:

Your Parser should accept input from "standard input" and output to "standard output".

Output:

If the input program is valid, output "`CORRECT`", otherwise output "`INVALID!`".

Sample 1:

Input	Output

`FUNC main ( )`
`BEGIN`
`  x := 2 + 2 !`
`  PRINT ( x * 100 ) !`
`END.`
	`CORRECT`

Sample 2:

Input	Output
`FUNC fibo ( n )` `BEGIN`	`CORRECT`

`  x := 1 !`
`  y := 2 !`
`  c := 3 !`
`  LOOP ( c < n )`
`    x := x + y !`
`    y := x - y !`
`    c := c + 1 !`
`  POOL`
`  RET x !`
`END.`	

Sample 3:

Input	Output

`FUNC gcd ( a , b )`
`BEGIN`
`  LOOP ( a # b )`
`    IF ( a > b )`
`      a := a - b !`
`    ELSE`
`      b := b - a !`
`    FI`
`  POOL`
`  RET a !`
`END.`
 	
`FUNC hello ( ) BEGIN`
`  PRINT ( "Hello" & "World!" ) ! END.`
	`CORRECT`

Sample 4:

Input	Output

`void main ( ) {`
`  cout << “Hello World”;`
`}`
	`INVALID!`

Sample 5:

Input	Output

FUNC foo (LOOP)
BEGIN BEGIN !
  x := END. !
  PRINT (RETURN) !
IF
	INVALID!