CS 280
Fall 2023
Programming Assignment 2

November 2nd, 2023

Due Date: Wednesday November 15, 2023, 23:59
Total Points: 20

In this programming assignment, you will be building a parser for a Simple Pascal-Like programming language. The syntax definitions of the programming language are given below using EBNF notations. Your implementation of a parser to the language is based on the following grammar rules specified in EBNF notations.

```
1.  Prog ::= PROGRAM IDENT ; DeclPart CompoundStmt .
2.  DeclPart ::= VAR DeclStmt; {  DeclStmt ; }
3.  DeclStmt ::= IDENT {, IDENT } : Type [:= Expr]
4.  Type ::= INTEGER | REAL | BOOLEAN | STRING
5.  Stmt ::= SimpleStmt | StructuredStmt
6.  SimpleStmt ::= AssignStmt | WriteLnStmt | WriteStmt
7.  StructuredStmt ::= IfStmt | CompoundStmt
8.  CompoundStmt ::= BEGIN Stmt {; Stmt } END
9.  WriteLnStmt ::= WRITELN (ExprList)
10. WriteStmt ::= WRITE (ExprList)
11. IfStmt ::= IF Expr THEN Stmt [ ELSE Stmt ]
12. AssignStmt ::= Var := Expr
13. Var ::= IDENT
14. ExprList ::= Expr { , Expr }
15. Expr ::= LogOrExpr ::= LogAndExpr { OR LogAndExpr }
16. LogAndExpr ::= RelExpr {AND RelExpr }
17. RelExpr ::= SimpleExpr  [ ( = | < | > ) SimpleExpr ]
18. SimpleExpr :: Term { ( + | - ) Term }
19. Term ::= SFactor { ( * | / | DIV | MOD ) SFactor }
20. SFactor ::= [( - | + | NOT )] Factor
21. Factor ::= IDENT | ICONST | RCONST | SCONST | BCONST | (Expr)
```

The following points describe the programming language. Note that not all of these points will be addressed in this assignment. However, they are listed in order to give you an understanding of the language semantics and what to be considered for implementing an interpreter for the language in Programming Assignment 3. These points are:

Table of Operators Precedence Levels

Precedence	Operator	Description	Associativity
1	Unary +, -, not	Unary plus, minus, and not (complement)	Right-to-Left
2	*, /, div, mod	Multiplication, Division, integer division, and modulus	Left-to-Right
3	+, -	Addition, and Subtraction,	Left-to-Right
4	<, >, =	• Less than, greater than, and equality	(no cascading)
5	and	• Logical Anding	Left-to-Right
6	or	• Logical Oring	Left-to-Right

1. The language has four types: integer, real, boolean, and string.
2. The precedence rules of operators in the language are as shown in the table of operators' precedence levels.
3. The PLUS, MINUS, MULT, DIV, IDIV, and MOD operators are left associative.
4. An IfStmt evaluates a logical expression (Expr) as a condition. If the logical condition value is true, then the If-clause statement is executed, otherwise they are not. An else clause for an IfSmt is optional. Therefore, If an Else-clause is defined, the Else-clause statement is executed when the logical condition value is false.
5. A writeln statement evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline. While a write would print the values without a newline.
6. The ASSOP operator (:=) in the AssignStmt assigns a value to a variable. It evaluates the Expr on the right-hand side and saves its value in a memory location associated with the left-hand side variable (Var). A left-hand side variable of a Numeric type must be assigned a numeric value. Type conversion must be automatically applied if the right-hand side numeric value of the evaluated expression does not match the numeric type of the left-hand side variable. While a left-hand side variable of string or Boolean type must be assigned a value of the same type as of the left-hand side variable.
7. The binary operations of numeric operators as addition, subtraction, multiplication, division, integer division, and modulus are performed upon two numeric operands. While the binary logic operators are performed upon two Boolean operands.
8. Similarly, relational operators (=, <, and >) operate upon two compatible type operands. The evaluation of a relational expression produces either a logical true or false value. For all relational operators, no cascading is allowed.

9. The unary sign operators (+ or -) are applied upon unary numeric type operands only. While the unary not operator is applied upon a Boolean type operand only.
10. It is an error to use a variable in an expression before it has been assigned.

Parser Requirements:

Implement a recursive-descent parser for the given language. You may use the lexical analyzer you wrote for Programming Assignment 1, OR you may use the provided implementation when it is posted. The parser should provide the following:

- The results of an unsuccessful parsing are a set of error messages printed by the parser functions, as well as the error messages that might be detected by the lexical analyzer.
- If the parser fails, the program should stop after the parser function returns.
- The assignment does not specify the exact error messages that should be printed out by the parser; however, the format of the messages should be the line number, followed by a colon and a space, followed by some descriptive text. Suggested messages might include "Missing semicolon at end of Statement.", "Incorrect Declaration Statement.", "Missing Right Parenthesis", "Undefined Variable", "Missing END", etc.
- If the scanning of the input file is completed with no detected errors, the parser should display the message (DONE) on a new line before returning successfully to the caller program.

Provided Files

You are given the header file for the parser, "parser.h" and **an incomplete file for the "parser.cpp", called "GivenParserPart.cpp". You should use "GivenParserPart.cpp" to complete the implementation of the parser.** In addition, "lex.h", "lex.cpp", and "prog2.cpp" files are also provided. The descriptions of the files are as follows:

"parser.h"

"parser.h" includes the following:

- Prototype definitions of the parser functions (e.g., Prog, Stmt, etc.)

"GivenParserPart.cpp"

- A map container that keeps a record of the defined variables in the parsed program, defined as: `map<string, bool> defVar;`
  - The key of the `defVar` is a variable name, and the value is a Boolean that is set to true when the first time the variable has been initialized, otherwise it is false.
- A function definition for handling the display of error messages, called `ParserError`.
- Functions to handle the process of token lookahead, GetNextToken and PushBackToken, defined in a namespace domain called *Parser*.
- Static int variable for counting errors, called `error_count`, and a function to return its value, called `ErrCount()`.
- Implementations of some functions of the recursive-descent parser.

"prog2.cpp"

- You are given the testing program "prog2.cpp" that reads a file name as an argument from the command line. The file is opened for syntax analysis, as a source code for your parser.
- A call to Prog() function is made. If the call fails, the program should stop and display a message as "Unsuccessful Parsing ", and display the number of errors detected. For example:

```
Unsuccessful Parsing
Number of Syntax Errors: 3
```

- If the call to Prog() function succeeds, the program should stop and display the message "Successful Parsing ", and the program stops.

Vocareum Automatic Grading

- You are provided by a set of 19 testing files associated with Programming Assignment 2. Vocareum automatic grading will be based on these testing files. You may use them to check and test your implementation. These are available in compressed archive as "PA 2 Test Cases.zip" on Canvas assignment. The testing case of each file is defined in the Grading table below.
- The automatic grading of a clean source code file will be based on checking against the output message:

```
(DONE)
Successful Parsing
```

- In each of the other testing files, there are one or more syntactic errors. The parser would detect the first discovered syntactic error and returns back unsuccessfully. The automatic grading process will be based on the statement number at which this first syntactic error has been found, and the number of associated error messages with this syntactic error.
- You can use whatever error message you like. There is no check against the contents of the error messages.
- A check of the number of errors your parser has produced and the number of errors printed out by the program are made.

Submission Guidelines

- Submit your "parser.cpp" implementation through Vocareum. The "lex.h", "parser.h", "lex.cpp" and "prog2.cpp" files will be propagated to your Work Directory.
- **Submissions after the due date are accepted with a fixed penalty of 25%. No submission is accepted after Saturday 11:59 pm, November 18, 2023.**

Grading Table

Item	Points
Compiles Successfully	1
testprog1: Incorrect Type	1
testprog2: Variable Redefinition	1
testprog3: Missing Program Name.	1
testprog4: Missing a Comma in Declaration Statement	1
testprog5: Invalid Operator Symbol	1
testprog6: Missing end of compound statement	1
testprog7: Missing Semicolon	1
testprog8: Missing left Parenthesis in write or Writeln Statement	1
testprog9: Missing Right Parenthesis	1
testprog10: Missing Assignment Operator	1
testprog11: Incorrect initialization expression	1
testprog12: Missing begin of compound statement	1
testprog13: Missing operand after an operator	1
testprog14: Undeclared Variable	1
testprog15: Missing program Keyword	1
testprog16: Non-recognizable Declaration Part	1
testprog17: Syntax Error in IF Statement	1
Testprog18: Illegal relational expression	1
testprog19: Clean Program	1
Total	20