CS 280
Spring 2023
Programming Assignment 2

March 9, 2023

Due Date: Sunday, April 2nd, 2023, 23:59
Total Points: 20

In this programming assignment, you will be building a parser for the Simple Perl-Like (SPL) programming language. The syntax definitions of SPL 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 ::= StmtList
2.  StmtList ::= Stmt ;{ Stmt; }
3.  Stmt ::= AssignStme | WriteLnStmt | IfStmt
4.  WriteLnStmt ::= writeln (ExprList)
5.  IfStmt ::= if (Expr) '{' StmtList '}' [ else '{' StmtList '}' ]
6.  AssignStmt ::= Var = Expr
7.  Var ::= NIDENT | SIDENT
8.  ExprList ::= Expr { , Expr }
9.  Expr ::= RelExpr  [(-eq|==) RelExpr ]
10. RelExpr ::= AddExpr  [ ( -lt | -gt | < | > ) AddExpr ]
11. AddExpr :: MultExpr { ( + | - | .) MultExpr }
12. MultExpr ::= ExponExpr { ( * | / | **) ExponExpr }
13. ExponExpr ::= UnaryExpr { ^ UnaryExpr }
14. UnaryExpr ::= [( - | + )] PrimaryExpr
15. PrimaryExpr ::= IDENT | SIDENT | NIDENT | ICONST | RCONST | SCONST
    | (Expr)
```

The following points describe the SPL 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 +, -	Unary plus, and minus,	Right-to-Left
2	^	Exponent	Right-to-Left
3	*, /, **	Multiplication, Division, and string repetition	Left-to-Right
4	+, -, . (Dot)	Addition, Subtraction, and String concatenation	Left-to-Right
5	<, > -gt, -lt	• Numeric Relational • String Relational	(no cascading)
6	== -eq	• Numeric Equality • String Equality	(no cascading)

1. The language has two types: Numeric, and String.
2. The SPL language does not have explicit declaration statements. However, variables are implicitly declared as Numeric type by a variable name starting with "$", or as String type by a variable name starting with "@".
3. All SPL variables must first be initialized by an assignment statement before being used.
4. The precedence rules of operators in the language are as shown in the table of operators' precedence levels.
5. The PLUS, MINUS, MULT, DIV, CAT, and SREPEAT operators are left associative.
6. An IfStmt evaluates a logical expression (Expr) as a condition. If the logical condition value is true, then the StmtList in the If-clause are executed, otherwise they are not. An else clause for an IfSmt is optional. Therefore, If an Else-clause is defined, the StmtList in the Else-clause are executed when the logical condition value is false.
7. A WritelnStmt evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.
8. 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. While a left-hand side variable of a String type must be assigned a string value. Type conversion must be automatically applied if the right-hand side value of the evaluated expression does not match the type of the left-hand side variable.
9. The binary operations of numeric operators as addition, subtraction, multiplication, and division are performed upon two numeric operands. While the binary string operator for concatenation is performed upon two string operands. If one of the operands does not match the type of the operator, that operand is automatically converted to the type of the operator.
10. Similarly, numeric relational and equality operators (==, <, and >) operate upon two numeric type operands. While, string relational and equality operators (-eq, -lt, -gt) operate upon two string type operands. The evaluation of a relational or an equality expression, produces either a true or false value. If one of the operands does not match the type of the operator, that operand is automatically converted to the type of the operator. For all relational and equality operators, no cascading is allowed.
11. The exponent operator is applied on a numeric type operand, and the exponent value must be a numeric type value. No automatic conversion to numeric type operand is applied in case of

an expression with exponent operator. Note that, exponent operators follow right-to-left association.

12. The binary operation for string repetition (**) operates upon a string operand as the first operand, where the second operand must be a numeric expression of integer value.
13. The unary sign operators (+ or -) are applied upon unary numeric type operands only.
14. 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". You should use "parser.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, StmtList, Stmt, etc.)

"Parser.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 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 is one syntactic error at a specific line. The automatic grading process will be based on the statement number at which this 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.

<u>Submission Guidelines</u>

- Submit your "parse.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 Wednesday 11:59 pm, April 5, 2023.**

Grading Table

Item	Points
Compiles Successfully	1
testprog1: Undefined variable	1
testprog2: Missing semicolon	1
testprog3: Missing operand after string operator	1
testprog4: Missing left parenthesis in writeln Statement or if statement condition	1
testprog5: Missing right parenthesis in writeln Statement or if statement condition	1
testprog6: Missing left brace in if statement clause	1
testprog7: Missing right brace in if statement clause	1
testprog8: Missing right brace in else-clause	1
testprog9: Missing assignment operator	1
testprog10: Illegal Equality Expression	1
testprog11: Illegal relational expression	1
testprog12: Missing comma in expression list	1
testprog13: Missing operand after numeric operator	1
testprog14: Missing right parenthesis after expression	1
testprog15: Invalid variable name	1
testprog16: Clean program 1 with variables definitions	1
testprog17: Clean Program 2	1
testprog18: Clean Program 3	1
testprog19: Clean Program 4	1
Total	20