CS 280
Spring 2023
Programming Assignment 3

April 13, 2023

Due Date: Sunday, April 30, 2023, 23:59
Total Points: 20

In this programming assignment, you will be building an interpreter for our SPL programming language. The grammar rules of the language and its tokens were given in Programming Assignments 1 and 2. You are required to modify the parser you have implemented for the SPL language to implement an interpreter for it. The specifications of the grammar rules are described in EBNF notations as follows.

```
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. These points that are related to the language syntactic rules were implemented in Programming Assigning 2. However, the points related to the SPL language semantics are required to be implemented in the SPL language interpreter. 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.

Interpreter Requirements:

Implement an SPL interpreter for the language based on the recursive-descent parser developed in Programming Assignment 2. You need to complete the implementations of the *Value* class member functions and overloaded operators. You need to modify the parser functions to include the required actions of the interpreter for evaluating expressions, determining the type of expression values, executing the statements, and checking run-time errors. You may use the parser you wrote for Programming Assignment 2. Otherwise you may use the provided implementations for the parser when it is posted. Rename the "parser.cpp" file as "parserInt.cpp" to reflect the applied changes on the current parser implementation for building an interpreter. The interpreter should provide the following:

- It performs syntax analysis of the input source code statement by statement, then executes the statement if there is no syntactic or semantic error.
- It builds information of variables types in the symbol table for all the defined variables.
- It evaluates expressions and determines their values and types. **You need to implement the member functions and overloaded operator functions for the Value class.**
- The results of an unsuccessful parsing and interpreting are a set of error messages printed by the parser/interpreter functions, as well as the error messages that might be detected by the lexical analyzer.
- **Any failures due to the process of parsing or interpreting the input program should cause the process of interpretation to stop and return back.**
- In addition to the error messages generated due to parsing, **the interpreter generates error messages due to its semantics checking**. The assignment does not specify the exact error messages that should be printed out by the interpreter. However, the format of the messages should be the line number, followed by a colon and a space, followed by some descriptive text, similar to the format used in Programming Assignment 2. Suggested messages of the interpreter's semantics errors might include messages such as "Run-Time Error-Illegal Mixed Type Operands", " Run-Time Error-Illegal Assignment Operation", "Run-Time Error-Illegal Division by Zero", etc.

Provided Files

You are given the following files for the process of building an **SPL** interpreter. These are "lex.h', "lex.cpp", "val.h", "parseInt.h", and "parseInt.cpp" with definitions and partial implementations of some functions. You need to complete the implementation of the interpreter in the provided copy of "parseInt.cpp. "parser.cpp" will be provided and posted later on.

1. **"val.h" includes the following:**

- A class definition, called Value, representing a value object in the interpreted source code for values of constants, variables or evaluated expressions.
- You are required to provide the implementation of the Value class in a separate file, called "val.cpp", which includes the implementations of the all the member functions and overloaded operator functions specified in the Value class definition.

2. **"parserInt.h" includes the prototype definitions of the parser functions as in "parse.h" header file with the following applied modifications:**
   - `extern bool Var(istream& in, int& line, LexItem & idtok);`
   - `extern bool ExprList(istream& in, int& line);`
   - `extern bool Expr(istream& in, int& line, Value & retVal);`
   - `extern bool RelExpr(istream& in, int& line, Value & retVal);`
   - `extern bool AddExpr(istream& in, int& line, Value & retVal);`
   - `extern bool MultExpr(istream& in, int& line, Value & retVal);`
   - `extern bool ExponExpr(istream& in, int& line, Value & retVal);`
   - `extern bool UnaryExpr(istream& in, int& line, Value & retVal);`
   - `extern bool PrimaryExpr(istream& in, int& line, int sign, Value & retVal);`

3. **"parserInt.cpp" includes the following:**
   - Map container definitions given in "parse.cpp" for Programming Assignment 2.
   - The declaration of a map container for temporaries' values, called `TempsResults`. Each entry of `TempsResults` is a pair of a string and a Value object, representing a variable name, and its corresponding Value object.
   - A map container SymTable that keeps a record of each declared variable in the parsed program and its corresponding type.
   - The declaration of a pointer variable to a queue container of Value objects.
   - Implementations of the interpreter actions in some example functions.

4. **"parser.cpp"**
   - Implementations of parser functions in "parser.cpp" from Programming Assignment 2.
     - **It will be provided after the deadline of PA 2 submission (including any extensions).**

5. **"prog3.cpp":**
   - You are given the testing program "prog3.cpp" that reads a file name from the command line. The file is opened for syntax analysis and interpretation, as a source code of the language.
   - A call to *Prog()* function is made. If the call fails, the program should display a message as "Unsuccessful Interpretation ", and display the number of errors detected, then the program stops. For example:
     `Unsuccessful Interpretation`

`Number of Syntax Errors: 3`

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

Vocareum Automatic Grading

- You are provided by a set of 19 testing files associated with Programming Assignment 3. 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 "PA 3 Test Cases.zip" on Canvas assignment. The testing case of each file is defined in the Grading table below.
- Automatic grading of testing files with no errors will be based on checking against the generated outputs by the executed source program and the output message:
  `Successful Execution`
- In each of the other testing files, there is one semantic or syntactic error at a specific line. The automatic grading process will be based on identifying the statement number at which this error has been found and associated with one or more error messages.
- You can use whatever error message you like. There is no check against the contents of the error messages. However, a check for the existence of a textual error messages is done.
- A check of the number of errors your parser has produced and the number of errors printed out by the program is also made.

Submission Guidelines

- **Submit your "parserInt.cpp" and "val.cpp" implementations through Vocareum.**
- **Submissions after the due date are accepted with a fixed penalty of 25%. No submission is accepted after Wednesday 11:59 pm, May 3rd, 2023.**

Grading Table

Item	Points
Compiles Successfully	1
testcase1: Testing string illegal repeat operation	1
testcase2: Testing illegal numeric operation	1
testcase3: Testing run-time error: division by zero	1
testcase4: Illegal exponentiation operation	1
testcase5: Illegal operand for the sign operation	1
testcase6: Testing mixed operand types for string relational operation	1
testcase7: Illegal mixed mode assignment operation to a numeric variable	1
testcase8: Illegal expression type for If statement condition	1
testcase9: Illegal assignment operatoration	1
testcase10: Illegal operation for If statement condition	1
testcase11: Illegal operand type for the operation	1
testcase12: Unrecognized input pattern	1
testcase13: Testing string repeat and catenation operators	1

testcase14: Testing if statement then-clause	1
testcase15: Testing if statement else-clause	1
testcase16: Testing exponentiation operation	1
testcase17: Clean Program: Solving a quadratic equation	1
testcase18: Testing mixed operand types for string relational operation	1
testcase19: Testing mixed mode assignment operation to a string variable	1
Total	20