CS 280
Fall 2025
Programming Assignment 3

Building an Interpreter for Basic Peril-Like Language

November 20, 2025

Due Date: December 8, 2025
Total Points: 20

In this programming assignment, you will be building an Interpreter for a Basic Perl-Like programming language, called BPL. 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 BPL 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 ::= IfStmt | AssignStmt | PrintLnStmt`
4. `PrintLnStmt ::= PRINTLN (ExprList)`
5. `IfStmt ::= IF (Expr) '{' StmtList '}' [ ELSE '{' StmtList '}' ]`
6. `Var ::= IDENT`
7. `ExprList ::= Expr { , Expr }`
8. `AssignStmt ::= Var AssigOp Expr`
9. `Expr ::= OrExpr`
10. `AssigOp ::= ( = | += | -= | .= )`
11. `OrExpr ::= AndExpr { || AndExpr }`
12. `AndExpr ::= RelExpr { && RelExpr }`
13. `RelExpr ::= AddExpr [ ( @le | @gt | @eq | < | >= | == ) AddExpr ]`
14. `AddExpr :: MultExpr { ( + | - | . ) MultExpr }`
15. `MultExpr ::= UnaryExpr { ( * | / | % | .x. ) UnaryExpr }`
16. `UnaryExpr ::= [( - | + | ! )] ExponExpr`
17. `ExponExpr ::= PrimaryExpr { ** PrimaryExpr }`
18. `PrimaryExpr ::= IDENT | ICONST | FCONST | SCONST | (Expr)`

The following points describe the BPL programming language. These points that are related to the language syntactic rules were implemented in Programming Assigning 2. However, the points related to the BPL language semantics are required to be implemented in the BPL language interpreter. These points are:

Table of Associativity and Precedence Levels of Operators

Precedence	Operator	Description	Associativity
1	**	Exponentiation	Right-to-Left
2	Unary +, -, !	Unary plus and minus, and NOT	Right-to-Left
3	*, /, %, .x.	Multiplication, Division, Remainder, and string repetition	Left-to-Right
4	+, -, .	Addition, Subtraction, and Concatenation	Left-to-Right
5	<, >=, ==, @le, @gt, @eq	Relational operators for numeric and string types	Left-to-Right
6	&&	Logical AND	Left-to-Right
7	||	Logical OR	Left-to-Right
8	=, +=, -=, .=	Assignment operators	Right-to-Left

1. The BPL language has two data types: Numeric and string. The data type is dynamically bound to a variable name in an assignment statement.

2. A variable has to be defined in an assignment statement before it is been used.

3. The associativity and precedence rules of operators in the language are as shown in the table of operators' precedence levels.

4. 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. Howevef, the BPL language does not include a Boolean type, instead it uses the following simple rules:

   - If the value is a number, 0 means false; all other numbers mean true.
   - Otherwise, if the value is a string, the empty string (") and the string '0' mean false; all other strings mean true.
   - If the variable doesn't have a value yet, it's an undefined variable error.

5. A PrinteLnStmt evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.

6. The ASSOP operator (=) and all the combined assignment operators (+=, -=, .=) assign a value to a variable according to the right-hand side expression. 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). The type of the variable is bound dynamically to the type of the right-hand side expression. However, the assignment of a Boolean value to a variable is illegal.

7. The binary operations for numeric operators are the addition, subtraction, multiplication, division, remainder (modulus), and exponentiation. These are performed upon two numeric operands. Except of the exponentiation operator, if any of the operands is a string, it will be automatically converted to a numeric value. For the remainder operator, a numeric operand is converted to an integer value in order to perform the operation.

8. The string concatenation operator is performed upon two string operands. If one of the operands is not a string, that operand is automatically converted to a string.

9. The string repetition operator must have the first operand as a string, while the second operand must be of a numeric type of an integer value.

10. Similarly, numeric relational and equality operators (==, <, and >=) operate upon two numeric type operands. While, string relational and equality operators (@eq, @le, @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. Logic binary operators (&&, ||) are applied on two operands, which their Boolean values are determined based on the following rules:
    - If the value is a number, 0 means false; all other numbers mean true.
    - Otherwise, if the value is a string, the empty string ('') and the string '0' mean false; all other strings mean true.
    - If the variable doesn't have a value yet, it's an undefined variable error.

12. The unary sign operators (+ or -) are applied upon unary numeric operand only. While the unary *not* operator (!) is applied upon a *Boolean* operand, according to the rules given previously.

Interpreter Requirements:

Implement an 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. 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 "parserInterp.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 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 interpreter. These are "lex.h', "lex.cpp", "val.h", "parserInt.h", and "GivenparserIntPart.cpp". The "GivenparserIntPart.cpp" file includes definitions of variables, functions and global container objects. You need to complete the implementation of the interpreter in the provided copy of "GivenparserInterpPart.cpp" and rename it as "parserInterp.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 all the member functions that are specified in the *Value* class definition.

2. **"parserInt.h" includes the prototype definitions of the parser functions as in "parser.h" header file with the following applied modifications:**

```
extern bool Var(istream& in, int& line, LexItem & idtok);
extern bool Expr(istream& in, int& line, Value & retVal);
extern bool OrExpr(istream& in, int& line, Value & retVal);
extern bool AndExpr(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 UnaryExpr(istream& in, int& line, Value & retVal);
extern bool ExponExpr(istream& in, int& line, int sign, Value &
retVal);
```

```
extern bool PrimaryExpr(istream& in, int& line, int sign, Value &
retVal);
```

3. "GivenparserIntPart.cpp" includes the following:

- Map container definitions given in "parser.cpp" for Programming Assignment 2.
- A map container *SymTable* that keeps a record of each declared variable in the parsed program and its corresponding type.
- 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.

4. "parser.cpp"

- Implementations of parser functions in "parser.cpp" from Programming Assignment 2.

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 messages:
  ```
  (DONE)

  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

- Please name your file as "firstinitial_lastname_parser.cpp". Where, "firstinitial" and "lastname" refer to your first name initial letter and last name, respectively. Uploading and submission of your implementations is via Vocareum. Follow the link on Canvas for PA 3 Submission page.
- The "lex.h", "parserInterp.h", "val.h", "lex.cpp" and "prog3.cpp" files will be propagated to your Work Directory.
- **Extended submission period of PA 3 will be allowed after the announced due date for 3 days with a fixed penalty of 25% deducted from the student's score. No submission is accepted after Thursday 11:59 pm, December 11, 2025.**

Grading Table

Item	Points
Compiles Successfully	1
testcase1: Illegal string repeat operation (testprog1)	1
testcase2: Illegal operand for an arithmetic operation (testprog2)	1
testcase3: Testing run-time error-Division by zero (testprog3)	1
testcase4: Illegal operand type for the sign operation	1
testcase5: Illegal exponentiation operation	1
testcase6: Illegal assignment of a Boolean value to a variable	1
testcase7: Invalid remainder (%) operation	1
testcase8: Invalid numeric relational operator with Boolean Operand	1
testcase9: Invalid string relational operator with Boolean operand	1
testcase10: Testing if statement string condition and combined assignments	1
testcase11: Testing if statement else-clause	1
testcase12: Testing if statement if-then-clause	1
testcase13: Testing string repeat operations	1
testcase14: Testing NOT operation	1
testcase15: Evaluation of exponentiation operation	1
testcase16: Testing mixed operand types expressions	1
testcase17: Testing nested if statements I	1
testcase18: Testing nested if statements II	1
testcase19: Testing nested if statements III	1
Total	**20**