# CS 280
# Programming Language Concepts

## Fall 2025

## Project Assignment 3 Description

## Building an Interpreter for Basic Peril-Like Language

# Programming Assignment 3

- Objectives
  - In this programming assignment, you will be building an interpreter for our Basic Perl-Like programming language, called BPL, based on the recursive-descent parser developed in Programming Assignment 2.

- Notes:
  - Read the assignment carefully to understand it.
  - Understand the functionality of the interpreter, and the required actions to be performed to execute the source code.

# Programming Assignment 3

- You are required to modify the parser you have implemented for the language to implement an interpreter for it.
- The grammar rules of the language and its tokens were given in Programming Assignments 1 and 2.
- The specifications of the language rules are described in EBNF notations.

# BPL Language Definition

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 ::= ( = | += | -= | .= )`

# BPL Language Definition

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)`

# Example Program of BPL Language

```
# Clean Program
# Testing if statement string condition
# and combined assignments
$r = 50;
flag = "true";
if (flag)
{
    $y_1 = 5;
    $r += 20;
};

$y_1 += (7.5);
$r -= 20;
flag .= $y_1;

println ('$r = ', $r, ", ", '$y_1 = ', $y_1, ", ", 'flag =
"', flag, '"');
```

# 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 | No cascading |
| 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 |

# Description of the BPL Language

- The BPL language has two data types: Numeric and string. The data type is dynamically bound to a variable name in an assignment statement.
- A variable has to be defined in an assignment statement before it is been used.
- 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. For the exponentiation operator, the operators must be of numeric types only.

# Description of the BPL Language

- The string concatenation operator 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.
- 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.
- 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.

# Description of the BPL Language

- 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.
- 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. Note that unary operators are not cascaded in BPL language.

# Description of the BPL Language

- 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. However, 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.

# Description of the BPL Language

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

- 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.

# Interpreter Requirements

- 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.

# Interpreter Requirements

- **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.

# Given Files

- “lex.h”
- “lex.cpp”
  - You can use your implementation, or use my lexical analyzer when I publish it.
- “parser.cpp”
  - It is provided after deadline of PA2 submissions (including any extensions).
- “parserInterp.h”
  - Modified version of “parser.h”.
- “GivenparserIntPart.cpp”
  - Definitions of variables, functions and global container objects.
- “val.h”

# Given Files

- **"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 member functions specified in the *Value* class definition.**
    - **Note:** All semantic rules for type checking of the compatibility of operands related to operators are defined by the *Value* member functions for those operators.
    - *Value* class member functions address the handling of Mixed Mode Expressions in BPL Language Interpreter. See RA 7 statement.

# Given Files

- **“parserInterp.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);
```

# Given Files

- **“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 stop and display a message as "Unsuccessful Interpretation", and display the number of errors detected. 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.

# Implementation of an Interpreter for the Language

- The interpreter parses the input source code statement by statement. For each parsed statement:
  - The parser/interpreter stops if there is a lexical/syntactic error.
  - If parsing is successful for the statement, it interprets the statement:
    - Checks for semantic errors (i.e., run-time) in the statement.
    - Stops the process of interpretation if there is a run-time error.
    - Executes the statement if no errors found.
  - 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.

# “parserInterp.cpp” Description

- Repository of temporaries values using a map container
  - `map<string, Value> TempsResults;`
  - Each entry of `TempsResults` is a pair of a string and a *Value* object. Each key element represents a variable name, and its corresponding *Value* object.
  - `TempsResults` holds all variables that have been defined by assignment statements.
    - Any variable that is to be accessed as an operand must have been defined before being used in the evaluation of any expression.
    - It is an execution/interpretation error to use a variable before being defined.
    - With dynamic binding of variables to a type, the *Value* object associated with a variable name determines the type of that variable.

# “parserInterp.cpp” Description

- Queue container for *Value* objects
  - `queue <Value> * ValQue;`
  - Declaration of a pointer variable to a queue of *`Value`* objects.
  - A queue structure to be created by the `PrintStmt` which makes `ValQue` to point to it.
  - Utilized to queue the evaluated list of expressions parsed by `ExprList`. In PrintStmt function, the values of evaluated expressions stored in the queue are removed in order, to be printed out.

- Implementations of the interpreter actions in some functions.

# Testing Program Requirements

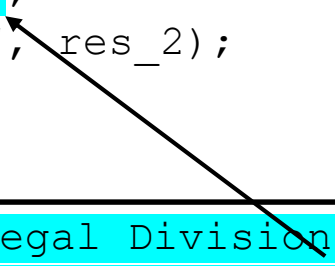
- **Vocareum Automatic Grading**
  - You are provided by a set of **19** testing files associated with Programming Assignment 3. These are available in compressed archive as "PA 3 Test Cases.zip" on Canvas assignment.
  - Automatic grading is performed based on the testing files. Test cases without errors are based on checking against the generated output by the interpreted source code execution, and the message:

    ```
    (DONE)

    Successful Execution
    ```
  - In the case of a testing file with a semantic error, there is one semantic 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 associated other error messages.
  - You can use whatever error messages you like. There is no check against the contents of the error messages. However, a check for the existence of a textual error message is done.
  - There is also a check of the number of errors your parser/interpreter has produced and the number of errors printed out by the program.

# **Example 1:** Testing division by zero run-time error (testprog3)

```
1. # Testing run-time error: division by zero
2.
3. $x1 = 0; #numeric variable
4. y_1 = 'Welcome!';   #string variable
5. num = "25.7";
6.
7. res_2 = num / $x1 ;
8. println("res_2 = ", res_2);
```



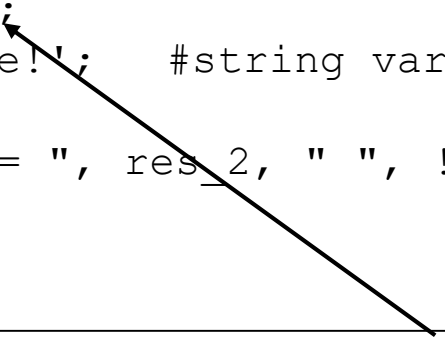
```
Run-Time Error: Illegal Division by Zero
1. Line 7: Illegal operand type or value for the division
operation.
2. Line 7: Missing Expression in Assignment Statement
3. Line 7: Incorrect Assignment Statement.
4. Line 7: Syntactic error in Program Body.
5. Line 7: Missing Program

Unsuccessful Interpretation
Number of Errors 5
```

# **Example 2:** Illegal operand type for sign operation (testprog4)

```
1. # Illegal operand type for sign operation
2.
3. $x1 = 0; #numeric variable
4.
5. num = -"25.7";
6. y_1 = 'Welcome!';   #string variable

7. println("num = ", res_2, " ", !y_1);
```



```
1. Line 5: Illegal Operand Type for Sign Operator
2. Line 5: Missing operand for an operator
3. Line 5: Missing Expression in Assignment Statement
4. Line 5: Incorrect Assignment Statement.
5. Line 5: Syntactic error in Program Body.
6. Line 5: Missing Program

Unsuccessful Interpretation
Number of Errors 6
```

# **Example 3:** Testing If statement string condition and combined assignments (testprog10)

```
1.      $r = 50;
2.      flag = "true";
3.      if (flag) {
4.              $y_1 = 5;
5.              $r += 20;
6.      };
7.      $y_1 += (7.5);
8.      $r -= 20;
9.      flag .= $y_1;
10.     println ('$r = ', $r, ", ", '$y_1 = ', $y_1, ", ", 'flag
= "', flag, '"');
```

```
$r = 50.0, $y_1 = 12.5, flag = "true12.5"


DONE

Successful Execution
```

# **Example 4:** Testing string repeat operations (testprog13)

```
1.  # Clean Program
2.  #Testing string repeat operations

3.  str = "Hello";
4.  $x1 = 2.5; #numeric variable
5.  y_1 = 'Welcome!' .x. $x1;  #valid string repeat operation
6.  println ("String Repeat Factor: ", $x1);
7.  println ("String Repeat Result: ", y_1);
8.  str_1 = $x1 .x. 3;
9.  println ("String Repeat Factor: ", 3);
10. println ("String Repeat Result: ", str_1);
```

```
String Repeat Factor: 2.5
String Repeat Result: Welcome!Welcome!
String Repeat Factor: 3.0
String Repeat Result: 2.52.52.5


DONE

Successful Execution
```