
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2023

Programming Assignment 3
Building an SPL Language Interpreter

Programming Assignment 3

- Objectives
  - In this programming assignment, you will be building an interpreter for our Simple Perl-Like (SPL) language 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 grammar rules are described in EBNF notations as follows:

SPL Language Definition

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)

Example Program of SPL Language

```
#Clean Program: Solving a quadratic equation
        $a = 2;
        $b = 7;
        $c = 3;
        $disc = $b * $b - 4 * $a * $c;
        writeln('Discriminant= ', $disc);
        if ($disc < 0){
                writeln('There are no real roots.');
        }
        else {
                if($disc == 0){
                        $x = -$b/(2 * $c);
                        writeln ('One real root: ', $x);
                }
                else{
                        $x1 = -$b + $disc ^ 0.5;
                        $x2 = -$b - $disc ^ 0.5;
                        writeln('Two real roots:');
                        writeln('x1 = ', $x1, '; ', 'x2 = ', $x2);
                };
        };
        writeln('Goodby!!');
```

Description of the SPL Language

- The language has two types: Numeric, and String.
- 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 "@".
- All SPL variables must first be initialized by an assignment statement before being used.
- The precedence rules of operators in the language are as shown in the table of operators' precedence levels.
- The PLUS, MINUS, MULT, DIV, CAT, and SREPEAT operators are left associative.

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)

Description of the SPL Language

- 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.
- 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 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.

Description of the SPL Language

- 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.
- 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.
- The unary sign operators (+ or -) are applied upon unary numeric type operands only.

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

Description of the SPL Language

- 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.
- 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 (from RA 8). 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.

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).
- “parserInt.h”
  - Modified version of “parser.h”.
- Partial “parserInt.cpp”

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 and overloaded operator functions specified in the Value class definition. (Complete the implementations of the overloaded operator functions from RA 8)

Given Files

- **“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 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);

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.

“val.h” Description

```
enum ValType { VINT, VREAL, VSTRING, VBOOL, VERR };
class Value {
private:
  ValType T; bool Btemp; int Itemp; double Rtemp; string Stemp;

public:
  Value() : T(VERR), Itemp(0), Rtemp(0.0){}
  Value(bool vb) : T(VBOOL), Btemp(vb), Itemp(0), Rtemp(0.0) {}
  Value(int vi) : T(VINT), Itemp(vi) {}
  Value(double vr) : T(VREAL), Itemp(0), Rtemp(vr) {}
  Value(string vs) : T(VCHAR), Itemp(0), Rtemp(0.0), Stemp(vs) {}

  ValType GetType() const { return T; }
  bool IsErr() const { return T == VERR; }
  bool IsInt() const { return T == VINT; }
  bool IsString() const { return T == VSTRING; }
  bool IsReal() const {return T == VREAL;}
  bool IsBool() const {return T == VBOOL;}
```

“val.h” Description

```
int GetInt() const { if( IsInt() ) return Itemp; throw "RUNTIME
ERROR: Value not an integer"; }
string GetString() const { if( IsString() ) return Stemp; throw
"RUNTIME ERROR: Value not a string"; }
float GetReal() const { if( IsReal() ) return Rtemp; throw "RUNTIME
ERROR: Value not a real"; }
bool GetBool() const {if(IsBool()) return Btemp; throw "RUNTIME
ERROR: Value not a boolean";}
void SetType(ValType type){ T = type;}
void SetInt(int val){ if( IsInt() ) {Itemp = val; else
                      throw "RUNTIME ERROR: Type not an integer";}
void SetReal(float val){ if( IsReal() ) Rtemp = val; else
                      throw "RUNTIME ERROR: Type not a real“;}
void SetString(string val){ if( IsString() ) Stemp = val; else
                      throw "RUNTIME ERROR: Type not a string"; }
void SetBool(bool val){ if(IsBool()) Btemp = val; else
                      throw "RUNTIME ERROR: Type not a boolean";}
}
```

“val.h” Description

```
// numeric overloaded add op to this
Value operator+(const Value& op) const;
// numeric overloaded subtract op from this
Value operator-(const Value& op) const;
// numeric overloaded multiply this by op
Value operator*(const Value& op) const;
// numeric overloaded divide this by op
Value operator/(const Value& op) const;
//numeric overloaded equality operator of this with op
Value operator==(const Value& op) const;
//numeric overloaded greater than operator of this with op
Value operator>(const Value& op) const;
//numeric overloaded less than operator of this with op
Value operator<(const Value& op) const;
//Numeric exponentiation operation this raised to the power of op
Value operator^(const Value& oper) const;
//string concatenation operation of this with op
Value Catenate(const Value& oper) const;
//string repetition operation of this with op
Value Repeat(const Value& oper) const;
//string equality (-eq) operator of this with op
Value SEqual(const Value& oper) const;
```

“val.h” Description

```
//string greater than (-gt) operator of this with op
Value SGthan(const Value& oper) const;
//string less than operator of this with op
Value SLthan(const Value& oper) const;

friend ostream& operator<<(ostream& out, const Value& op) {
  if( op.IsInt() ) out << op.Itemp;
     else if( op.IsString() ) out << op.Stemp ;
     else if( op.IsReal()) out << fixed << showpoint <<
     setprecision(1) << op.Rtemp;
     else if(op.IsBool()) out << (op.GetBool()? "true" : "false");
     else out << "ERROR";
        return out;
    }
```

“parserInt.cpp” Description

- All definitions from “parser.cpp”.
- A map container that keeps a record of each declared variable in the parsed program and its corresponding type, defined as:

```
map<string, Token> SymTable;
```

  - The key of the `SymTable` is a variable name, and the value is a Token that is set to the token of the variable type(i.e, SIDENT, or NIDENT).

“parserInt.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.

“parserInt.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 WriteInStmt which makes `ValQue` to point to it.
  - Utilized to queue the evaluated list of expressions parsed by `ExprList`. In WriteInStmt 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:
    `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.
  - 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.

Implementation Examples: PrintStmt

- WritelnStmt function
  - Grammar rule

    WritelnStmt := **writeln** (ExprList)
  - function calls ExprList()
    - Checks the returned value, if it returns false an error message is printed, such as

      Missing expression after print
    - Then WritelnStmt function returns a false value
  - Evaluation: the function prints out the list of expressions' values, and returns successfully.
    - The values to be printed out in order as they are inserted in the queue of Value objects, (*ValQue).
    - Insertion of values parsed in the ExprList are queued into *ValQue by the ExprList function.

Implementation Examples: WritelnStmt

```
bool WritelnStmt(istream& in, int& line) {
   LexItem t;
   /*create an empty queue of Value objects.*/
   ValQue = new queue<Value>;
   t = Parser::GetNextToken(in, line);
   if( t != LPAREN ) {. . .}
   bool ex = ExprList(in, line);
   if( !ex ) {//empty the ValQue and delete it. . .);
   t = Parser::GetNextToken(in, line);
   if(t != RPAREN ) { . . . }
   //Evaluate: print out the list of expressions' values
   while (!(*ValQue).empty()){
        cout << (*ValQue).front();
        ValQue->pop();}
   cout << endl;
   return ex;
}
```

Implementation Examples: ExprList

- ExprList Function Definition

```
bool ExprList(istream& in, int& line);
```

  - Grammar rule:

```
ExprList ::= Expr {, Expr}
```

  - Calls Expr() function:

```
status = Expr(in, line, retVal);
```

    - Stores the retVal in the (*ValQue)
    - Continues parsing the remaining expressions

Example 1: Illegal string repeat operation

```
1.  # Testing illegal string repeat operation
2.
3.  $x1 = 0.5; #numeric variable
4.  @_y1 = 'Welcome!' ** 'Hello';       #string variable
5.  @str = 'Hello ' . @_y1;
6.  $z = 0.0 ;
7.
8.  @r_25 = 50.;
9.  writeln ($x1, ' ', @r_25, ' ', @_y1, ' ', @str);
```

From the Value member function Repeat

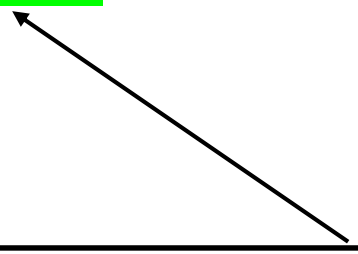
Output:

```
Invalid conversion from string to double.
1. Line # 4: Illegal operand type for the operation.
2. Line # 4: Missing Expression in Assignment Statement
3. Line # 4: Incorrect Assignment Statement.
4. Line # 4: Syntactic error in Program Body.
5. Line # 4: Missing Program

Unsuccessful Interpretation
Number of Errors 5
```

Example 2: Illegal operand type for the operation

```
1.  #Illegal operand type for the operation
2.       $r = 2;
3.       $x = 3.0;
4.
5.       $y = $x / $r * $x + ($x > $r);
6.
7.       writeln ( $x, ' ', $y);
```



```
1. Line # 5: Illegal operand type for the operation.
2. Line # 5: Missing Expression in Assignment Statement
3. Line # 5: Incorrect Assignment Statement.
4. Line # 5: Syntactic error in Program Body.
5. Line # 5: Missing Program

Unsuccessful Interpretation
Number of Errors 5
```

Example 3: Clean Program

```
#Clean Program: Solving a quadratic equation
$a = 2;
$b = 7;
$c = 3;
$disc = $b * $b - 4 * $a * $c;
writeln('Discriminant= ', $disc);
if ($disc < 0){
   writeln('There are no real roots.');
}
else {
   if($disc == 0){
      $x = -$b/(2 * $a);
      writeln ('One real root: ', $x);
   }
   else{
      $x1 = (-$b + $disc ^ 0.5)/(2 * $a);
      $x2 = (-$b - $disc ^ 0.5)/(2 * $a);
      writeln('Two real roots:');
      writeln('x1 = ', $x1, '; ', 'x2 = ', $x2);
   };
};
writeln('Goodby!!');
```

```
Discriminant= 25.0
Two real roots:
x1 = -0.5; x2 = -3.0
Goodby!!

(DONE)

Successful Execution
```

Example 4: Illegal expression type for If statement condition

```
1.  #Illegal expression type for If statement condition
2.        $r = 50;
3.        @flag = 'true';
4.        if (@flag . 'case') {
5.                $y_1 = $r * 5;
6.                @flag = 'hello';
7.        }
8.        else {
9.                $y_1 = (7.5);
10.               @flag = 'goodbye';
11.       };
12.       writeln ( $y_1, ' ', @flag);
13.       $r = $r + $y_1;
```

```
1. Line # 4: Illegal Type for If statement condition.
2. Line # 4: Incorrect If-Statement.
3. Line # 4: Syntactic error in Program Body.
4. Line # 4: Missing Program

Unsuccessful Interpretation
Number of Errors 4
```


NJIT
THE EDGE IN KNOWLEDGE