CS 280
Programming Language Concepts

Fall 2025

Recitation Assignment 7

Handling Mixed Mode Expressions in BPL Language Interpreter

Expression Evaluations

- Programming Assignment 3 (PA3) objective is to construct an interpreter for the language.
  - The interpreter needs to evaluate expressions and execute statements.
  - The evaluation process requires to determine the type of the operands for the execution of an operator for type checking. The evaluation process needs to determine whether the two types of operands of an operator are compatible or not.

Definitions of Expressions in the BPL Language

1. `Expr ::= LogANDExpr { || LogANDRxpr }`
2. `LogANDExpr ::= EqualExpr { && EqualExpr }`
3. `EqualExpr ::= RelExpr [ (== | != ) RelExpr ]`
4. `RelExpr ::= AddExpr [ ( < | > ) AddExpr ]`
5. `AddExpr :: MultExpr { ( + | - ) MultExpr }`
6. `MultExpr ::= UnaryExpr { ( * | / | % ) UnaryExpr }`
7. `UnaryExpr ::= [( - | + | ! )] PrimaryExpr`
8. `PrimaryExpr ::= IDENT | ICONST | RCONST | SCONST |( Expr )`

Expression Evaluations: Semantic Rules for BPL Language Expressions

- 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.
  - Example:

```
$x1 = 3.5; #numeric variable
y_1 = 'Welcome!';  #string variable
num = "25.7";
res_1 = $x1 * 4 + 8 / 5 ;
res_2 = num / $x1 ;
res_3 = y_1 - num;#invalid operations
y_1 = num ** $x ** 2 ** 2;
```

Expression Evaluations: Semantic Rules for BPL Language Expressions

- 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.
  - Example:

```
$x = 6;
$y = -8 * $x;
flag = 'hello' . $y * 4;#valid concatenation
```

Expression Evaluations: Semantic Rules for BPL Language Expressions

- 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.
  - Example:

```
str = "Hello";
$x1 = 2.5; #numeric variable
#valid string repeat operations
y_1 = 'Welcome!' .x. $x1;
str_1 = $x1 .x. 3;

#Invalid string repeat operation
str_2 = $x1 .x. str ;
```

Expression Evaluations: Semantic Rules for BPL Language Expressions

- 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.

Expression Evaluations: Semantic Rules for BPL Language Expressions

- Example:

```
size = 50;
$z = 6;
$w = -8 * size;
word1 = 'good';
word2 = "pass";
if (word1 @le word2 || $w >= 0) {
  if($z >= 8)
  {
    ind = 'hello';
    $z = $z + 1;
  }
  else
  {
    ind = 'goodbye';
    $z = $z - 1;
  };
}
```

Expression Evaluations: Semantic Rules for BPL Language Expressions

- 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.
- Example

```
$x = 6;
$y = -8 * $x;
flag = 'true';
if (flag && $x == 6) {
    $y_1 = -5;
    flag = 'hello' . $y_1 * 4;
}
```

Expression Evaluations: Semantic Rules for BPL Language Expressions

- 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. Examples:

```
#invalid operations
num = -"25.7";
$y_1 = -!(7.5);

y_1 = 'Welcome!';     #string variable
println("!y_1 = ", !y_1);
```

Expression Evaluations

- *Value* Class Definition
  - A class representing values of the different BPL language types using C++ types as (double, bool, and string) through three data members.
  - An enumerated type data member that records the type of a *Value* object.
  - Getter and Setter member functions
  - Functions for testing the type of a *Value* object
  - Overloaded operators for numeric arithmetic and logic operators in the language.
  - Overloaded operators for relational operators in the language (==, <, >=).
  - Member function for performing string relational operations (@le, @gt, @eq).
  - Member function for performing string catenation and repetition operation.

“val.h” Description

```
enum ValType { VNUM, VSTRING, VBOOL, VERR };
class Value {
    ValType    T;
    bool    Btemp;
    double   Ntemp;
    string     Stemp;
    //string ErrMsg;
public:
    Value() : T(VERR), Btemp(false), Ntemp(0.0), Stemp("") {}
    Value(bool vb) : T(VBOOL), Btemp(vb), Ntemp(0.0), Stemp("") {}
    Value(double vr) : T(VNUM), Btemp(false), Ntemp(vr), Stemp("")
{}
    Value(string   vs)   :   T(VSTRING),   Btemp(false),   Ntemp(0.0),
Stemp(vs) {}
ValType GetType() const { return T; }
    bool IsErr() const { return T == VERR; }
    bool IsString() const { return T == VSTRING; }
    bool IsNum() const {return T == VNUM;}
    bool IsBool() const {return T == VBOOL;}
```

“val.h” Description

```
//Getter member functions
int GetNum() const { if( IsNum() ) return Ntemp; throw "RUNTIME
ERROR: Value not an integer"; }
. . . . .
//Setter member functions
void SetType(ValType type){ T = type;}
void SetNum(double val){ if( IsNum() ) {Ntemp = val; else
                        throw "RUNTIME ERROR: Type not an integer";}
. . . . .
// numeric overloaded add this to op
Value operator+(const Value& op) const;
// numeric overloaded subtract op from this
Value operator-(const Value& op) const;
```

“val.h” Description

```
//overloaded numeric equality operator of this with op
Value operator==(const Value& op) const;
. . . .
//overloaded logical Anding operator of this with op
Value operator&&(const Value& op) const;
. . . . .

//Overloaded insertion operator for printing Value objects
friend ostream& operator<<(ostream& out, const Value& op) {
   if( op.IsNum()) out << fixed << setprecision(1) << op.Ntemp;
   else if( op.IsString() ) out <<  op.Stemp ;
   else if(op.IsBool()) out << (op.GetBool()? "true" : "false");
   else out << "ERROR";
  return out;
}
```

Given Files: "parserInterp.h" Description

- **"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);`

    - Var function returns a LexItem ref, `idtok`, to an object that carries a variable's lexeme and token. It is used by AssignStmt function to search for the variable name in the SymTable (or TempsResults) container for creating an entry for it with its type, or setting the entry with an updated type in BPL.
    - The Value attribute of the Var nonterminal is inherited from the Expr Value attribute in the AssignStmt nonterminal.

Given Files: "parserInterp.h" Description

```
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);
```

- The *retVal* reference parameter returns a reference to an object as the result of evaluating an expression.
- All the *retVal* parameters are representing synthesized attributes for the expression nonterminals.

Implementation Issues for Expressions

- Implementation issues for an interpreter
  - All language non-terminal rules that are representing expressions are associated with a *Value* object as a synthesized attribute. (See section 3.4 on attribute grammars in Ch. 3)
  - See the definition of the *Value* class. Each *Value* object represents a value of one of the possible operand types and its type.
  - The interpreter is based on the recursive descent parser. All the functions representing expressions have an added reference parameter to a *Value* object that returns the result of evaluating an expression and its type.
  - All semantic functions associated with expression rules are implemented by the *Value* class member functions defining the BPL language operations.

Implementation Issues for Expressions

- Evaluation of an expression based on a binary operator OP, and using two operands, *val1* and *val2*, of *Value* objects produces a *Value* object, *retVal*, where
  - `retVal = val1 OP val2`
    - Assuming the *Value* class has a member function as `operatorOP()`:

      ```
      Value operatorOP(const Value & val)const;
      ```

    - For example: A multiplication operation in the `MultExpr` function would evaluate the value of the operation as:

      ```
      retVal = val1 * val2
      ```

      Where retVal is the MultExpr reference parameter, and val1 and val2 are the references to the returned objects from the UnaryExpr() function.
  - `retVal = val1.Mfun(val2)`
    - Assuming the *Value* class has a member function as `Mfun()`:

      ```
      Value MFun(const Value & val)const;
      ```

    - For example as in **Repeat(), and SEQ() member** functions.

Implementation Issues for Expressions

- Evaluation of an expression based on a unary operator OP, and using a single operand, *val1* of *Value* object produces a *Value* object, *retVal*, where
  - `retVal = OP val1`
    - Note: Only the NOT (!) operator has been overloaded.
    - Evaluation of NOT operator is done in the UnaryExpr() function.
    - `val1` is the reference to the `Value` object returned by `ExponExpr`.
  - Assuming the *Value* class has a member function as `operatorOP()`:

    ```
    Value operatorOP()const;
    ```

- The sign parameter in **`ExponExpr and PrimaryExpr`** functions is used as an inherited attribute by the **`ExponExpr and PrimaryExpr`** nonterminals from the **`UnaryExpr.`**

Recitation Assignment 7

- In this assignment, you are given the definition of a class, called *Value*, which represents values of operands in the Basic Perl-Like (BPL) language interpreter (PA 3). The class represents the types of operands in BPL language (i.e., numeric, string and Boolean) using data members of C++ types as double, string, and Boolean, respectively. The objective of defining the *Value* class is to facilitate constructing an interpreter for the language which evaluates expressions and executes statements using C++.
- The *Value* class includes as member functions overloaded operator functions for the evaluation of BPL arithmetic operators (+, -, *, /, and %), equality and relational operators (==, <, and >=), and logical operators (&&, ||, and !). It includes also member functions for the BPL operators: Exponentiation (Exponent), Concatenation (Catenate), string repetition (Repeat), and String relational operators (@eq, @le, @gt).

Recitation Assignment 7

- In RA 7, you are required to implement some of the member functions of the *Value* class that are used to represent the evaluation of operations by those operators. The objective of the assignment is to enable testing the class separately as a unit before using it in the construction of the interpreter in PA 3. You are required to implement the following member functions:
  - **Operator*(), operator<(), Catenate(), Repeat(), and SEQ()**
- **Note**: It is recommended to implement the other overloaded operators in the *Value* class and testing them before using the class implementation in the PA 3 interpreter project.

Recitation Assignment 7

- **Vocareum Automatic Grading**
  - A driver program, called "RA7prog.cpp", is provided for testing the implementation on Vocareum. The "RA8prog.cpp" will be propagated to your Work directory, along with the definition of the *Value* class in the "val.h" file.
  - You are graded based on 5 test cases that are generated by the driver program. The expected output results of those 5 test cases are provided in the "RA 7 Test Cases.zip" archive and associated with the Recitation Assignment 7 on Canvas. Each test case checks the implementation of one of the required member functions to be implemented. Vocareum automatic grading will be based on the produced output of your implementations for the required member function compared with the test case output file. You may use them to check and test your implementation.
  - "RA7prog.cpp" is available with the other assignment material on Canvas. Review the driver program for implementation details.

Recitation Assignment 7: Driver Program Functions

```
void ExecuteMult(const Value& val1, const Value& val2) {
  cout << val1 << " * " << val2 << " is " <<  val1 * val2 << endl;
}

void ExecuteLThan(const Value& val1, const Value& val2) {
  cout << val1 << " < " << val2 << " is " <<  (val1 < val2) << endl;
}

void ExecuteSEQ(const Value& val1, const Value& val2) {
  cout << val1 << " @eq " << val2 << " is " <<   (val1.SEQ(val2))<<
  endl;
}

void ExecuteCatenate(const Value& val1, const Value& val2) {
  cout << val1 << " . " << val2 << " is " <<  (val1.Catenate(val2)) <<
  endl;
}

void ExecuteRepeat(const Value& val1, const Value& val2) {
  cout << val1 << " .x. " << val2 << " is " <<  (val1.Repeat(val2)) <<
  endl;
}
```

Partial Output of Multiplication: See case1.correct

```
. . . . . .
9.5 * ERROR is ERROR
9.5 * 9.5 is 90.2
9.5 * 4.0 is 38.0
Coercion Error: Invalid conversion from string to double.
9.5 * CS280 is ERROR
Coercion Error: Invalid conversion from string to double.
9.5 * Fall 2025 is ERROR
9.5 * true is ERROR
9.5 * false is ERROR
9.5 * 25 is 237.5
4.0 * ERROR is ERROR
4.0 * 9.5 is 38.0
4.0 * 4.0 is 16.0
Coercion Error: Invalid conversion from string to double.
4.0 * CS280 is ERROR
Coercion Error: Invalid conversion from string to double.
4.0 * Fall 2025 is ERROR
4.0 * true is ERROR
4.0 * false is ERROR
4.0 * 25 is 100.0R
. . . . . .
```

Partial Output of String Repeat: See case5.correct

```
. . . . . .
true .x. ERROR is ERROR
true .x. 9.5 is ERROR
true .x. 4.0 is ERROR
true .x. CS280 is ERROR
true .x. Fall 2025 is ERROR
true .x. true is ERROR
true .x. false is ERROR
true .x. 25 is ERROR
false .x. ERROR is ERROR
false .x. 9.5 is ERROR
false .x. 4.0 is ERROR
false .x. CS280 is ERROR
false .x. Fall 2025 is ERROR
false .x. true is ERROR
false .x. false is ERROR
false .x. 25 is ERROR
25 .x. ERROR is ERROR
25 .x. 9.5 is 252525252525252525
25 .x. 4.0 is 25252525
Coercion Error: Invalid conversion from string to double.
25 .x. CS280 is ERROR
Coercion Error: Invalid conversion from string to double.
25 .x. Fall 2025 is ERROR
25 .x. true is ERROR
25 .x. false is ERROR
25 .x. 25 is 252525252525252525252525252525252525252525252525252525
```