
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2023

Recitation Assignment 8
Expression Evaluation

Expression Evaluations

- Programming Assignment 3 (PA3) objective is to construct an interpreter for the language.
  - The interpreter needs to evaluate expressions and execute statements.
  - In SPL (similar to Perl), the operator determines what operation is performed, independent of the type of the operands. For example $x + @y is always a numeric addition. The operand @y must be converted to a numeric value.
    - In SPL language, there are two versions of some operators (e.g., relational operators)
  - The evaluation process requires to determine whether the type of the operand is compatible with the type of the operator. A conversion process is applied automatically to the non-compatible operand type. If the conversion process is not successful, the evaluation process is non-successful and a semantic error message is generated.

Definitions of Expressions in the SPL Language

1. `Expr ::= RelExpr [(-EQ|==) RelExpr ]`
2. `RelExpr ::= AddExpr [ ( -LT | -GT | < | > ) AddExpr ]`
3. `AddExpr :: MultExpr { ( + | - | .) MultExpr }`
4. `MultExpr ::= ExponExpr { ( * | / | **) ExponExpr }`
5. `ExponExpr ::= UnaryExpr { ^ UnaryExpr }`
6. `UnaryExpr ::= [( - | + )] PrimaryExpr`
7. `PrimaryExpr ::= IDENT | SIDENT | NIDENT | ICONST | RCONST | SCONST | (Expr)`

Expression Evaluations: Semantic Rules for Expressions

- Numeric Operators
  - The binary operations for addition, subtraction, multiplication, division, and exponentiation (+, -, *, and /) are performed upon two numeric operands. If one or both operands are not numeric, an attempt is made to convert the non-numeric operand to a numeric one. Otherwise, there is a semantic error.
  - The exponentiation operator is applied upon numeric operands only. No type conversion from a string to numeric is possible.
  - Similarly, the numeric relational operators (==, <, >) operate upon two numeric operands. The evaluation of a numeric relational expression, produces either a true or false value.
  - The unary sign operators (+ or -) are applied upon one numeric operand.

Expression Evaluations: Semantic Rules for Expressions

- String Operators
  - The binary operation for string concatenation (. Dot) is performed upon two string operands. If one or both operands are numeric, an attempt is made to convert the operand to a string one. Otherwise, there is a semantic error.
  - Similarly, the string relational operators (-eq, -lt, -gt) operate upon two string operands. The evaluation of a string relational expression, produces either a true or false value.
  - 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.

Expression Evaluations

- Value Class Definition
  - A class representing values of the different types using C++ types as (int, double, bool, and string) through four 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 operators in the language: addition, subtraction, multiplication, division, and exponentiation.
  - Overloaded operators for numeric relational operators in the language (==, <, >).
  - Member functions for string operators in the language: concatenation, and repetition.
  - Member functions for string relational operators in the language (-eq, -lt, -gt).

“val.h” Description

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

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

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

“val.h” Description (Cont’d)

```
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"; }

double 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){Itemp = val;}

void SetReal(double val){Rtemp = val;}

void SetString(string val){Stemp = val;}

void SetBool(bool val){Btemp = val;}
```

“val.h” Description (Cont’d)

```
// 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;
```

Highlighted definitions in Yellow are to be implemented in RA8

"val.h" Description (Cont'd)

```
  //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;
  //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 << "\"" <<
  op.Stemp;
    else if( op.IsReal()) out << fixed << showpoint <<
       setprecision(2) << op.Rtemp;
    else if(op.IsBool()) out << (op.GetBool()? "true" : "false");
    else out << "ERROR";

    return out;
  }
};
```

Highlighted definitions in Yellow are to be implemented in RA8

“parserInt.h” Description

- **“parserInt.h”:** includes the prototype definitions of the parser functions as in “parser.h” header file with the following applied modifications:
  - 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);

Interpreter Implementation for Expressions Evaluations

- Implementation issues for an interpreter
  - All language non-terminals 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 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 evaluation of the expression and its type.

Problems of converting a string to double

- To satisfy the semantics of the SPL language, a string operand for a numeric operator must be converted to a numeric value. Since all numeric values are double, we make the conversion to be always to a double type.
  - Use the function stod* from <string>
    - std::stod
    - double stod (const string& str, size_t* idx = 0);
    - Exceptions
      - If no conversion could be performed, an invalid_argument exception is thrown.
      - If the value read is out of the range of representable values by a double (in some library implementations, this includes underflows), an out_of_range exception is thrown.

* https://cplusplus.com/reference/string/stod/?kw=stod

An Overview of Exception Handling in C++

- In order to avoid the termination of the interpreter on non-successful conversions from string to a double, we need to apply exception handling.
  - All overloaded numeric operator functions should apply exception handling clauses when using the “stod” function.

- In a language without exception handling
  - When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated.
- In a language with exception handling
  - Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing.

Basic Concepts

- Many languages allow programs to trap input/output errors (including EOF)
- An *exception* is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called *exception handling*
- The exception handling code unit is called an *exception handler*

Exception Handling Alternatives

- An exception is raised when its associated event occurs
- A language that does not have exception handling capabilities can still define, detect, raise, and handle exceptions (user defined, software detected)
- Alternatives:
  - Send an auxiliary parameter or use the return value to indicate the return status of a subprogram
  - Pass a label parameter to all subprograms (error return is to the passed label)
  - Pass an exception handling subprogram to all subprograms

Advantages of Built-in Exception Handling

- Error detection code is tedious to write and it clutters the program
- Exception handling encourages programmers to consider many different possible errors
- Exception propagation allows a high level of reuse of exception handling code

C++ Exception Handlers

- Exception Handlers Form:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
...
catch (formal parameter) {
-- handler code
}
```

Example of a `try-catch` Statement

```
 try
{
        // Statements that process personnel data and may throw
        // exceptions of type int, string, and SalaryError
}
catch ( int )
{
        // Statements to handle an int exception
}
catch ( string s )
{
       cout << s << endl;  // Prints "Invalid customer age"
        // More statements to handle an age error
}
catch ( SalaryError )
{
       // Statements to handle a salary error
}
```

The `catch` Function

- **`catch`** is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter need not have a variable
  - It can be simply a type name to distinguish the handler it is in from others
- The formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis (. . .), in which case it handles all exceptions not yet handled

Throwing Exceptions

- Exceptions are all raised explicitly by the statement:

  `throw` [*expression*]**`;`**

  - The brackets are metasymbols
  - A **`throw`** without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere.
  - The type of the expression disambiguates the intended handler.

Unhandled Exceptions

- An unhandled exception is propagated to the caller of the function in which it is raised.
- This propagation continues to the main function.
- If no handler is found, the default handler is called.

Continuation

- After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element
- Other design choices
  - All exceptions are user-defined
  - Exceptions are neither specified nor declared
  - The default handler, `unexpected`, simply terminates the program; `unexpected` can be redefined by the user
  - Functions can list the exceptions they may raise
  - Without a specification, a function can raise any exception (the **`throw`** clause)

class
std::invalid_argument*

- **class invalid_argument;**
- **Invalid argument exception**

This class defines the type of objects thrown as exceptions to report an invalid argument.

It is a standard exception that can be thrown by programs. Some components of the standard library also throw exceptions of this type to signal invalid arguments.

* https://cplusplus.com/reference/stdexcept/invalid_argument/

Partial Implementation of Member Function: Repeat()

```
//repeat string of this object op number of times
   Value Value::Repeat(const Value& op) const {
   //First operands is a string and the second operand is an integer
   double oper;
   if( IsString() && op.IsString()){
     //second operand must be converted to a numeric
     try {
       oper = stod(op.GetString());
     }
     catch( invalid_argument & arg){
       cout << "Invalid conversion from string to double." << endl;
       return Value();
     }
     int times = oper;
     string str = GetString(), newstr = "";
     for(int i = 1; i <= times; i++){
       newstr += str;
     }
     return Value( newstr );
   }
```

Partial Implementation of Member Function: Repeat()

```
else if(IsReal() && op.IsString()){
//Both operands to be converted
//first operand must be converted to a string
  ostringstream outStr1;
  outStr1 << GetReal();
  string stroper1 = outStr1.str(), newstr = "";
  //second operand must be converted to a numeric
  try {
       oper = stod(op.GetString());
  }
  catch( invalid_argument & arg){
       cout << "Invalid conversion from string to double." << endl;
       return Value();
  }
  int times = oper;
  for(int i = 1; i <= times; i++){
       newstr += stroper1;
  }
  return Value( newstr );
}
else if(IsString() && op.IsReal()){ . . .
//Other cases to follow
```

Recitation Assignment 8

- In this recitation assignment, you are given the definition of a class, called *Value*, which represents values of operands in the SPL language to implement its interpreter in Programming Assignment 3. These include values for the two defined types in the language: Numeric (double), and String (string). The objective of defining the *Value* class is to facilitate constructing an interpreter for the SPL language which evaluates expressions and executes statements using C++.
- In RA 8, you are required to implement some of the member functions of the *Value* class that apply the numeric or string operators in order to enable testing the class separately as a unit before using it in the construction of the interpreter in PA3. You are required to implement the following member functions:
  - `operator*(),operator==(),operator^(),Catenate(),SLthan()`

Recitation Assignment 8

- **Vocareum Automatic Grading**
  - A driver program, called "RA8prog.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 provided with 5 test case files associated with Recitation Assignment 8. Each test case checks the implementation of one of the overloaded operator functions. Automatic grading by Vocareum will be based on the produced output of your implementations for the required overloaded operators compared with the test case file. You may use them to check and test your implementation. These are available in a compressed archive "RA 8 Test Cases.zip" on Canvas assignment.
  - "RA8prog.cpp" is available with the other assignment material on Canvas. Review the driver program for implementation details.

Partial Output of Multiplication: See case1.correct

```
ErrorVal=ERROR
doubleVal1=7.25
doubleVal2=3.00
StrVal1="CS280"
StrVal2="Spring 2023"
StrVal3="25.7"
StrVal4="3.0"
ERROR * 7.25 is ERROR
ERROR * 3.00 is ERROR
. . . .
7.25 * "Spring 2023" is ERROR
7.25 * "25.7" is 186.32
7.25 * "3.0" is 21.75
3.00 * ERROR is ERROR
3.00 * 7.25 is 21.75
Invalid conversion from string to double.
3.00 * "CS280" is ERROR
Invalid conversion from string to double.
3.00 * "Spring 2023" is ERROR
3.00 * "25.7" is 77.10
3.00 * "3.0" is 9.00
. . . .
"25.7" * ERROR is ERROR
"25.7" * 7.25 is 186.32
"25.7" * 3.00 is 77.10
Invalid conversion from string to double.
"25.7" * "CS280" is ERROR
. . . . .
```

Partial Output of Concatenation: See case4.correct

```
. . . . . .
7.25 . ERROR is ERROR
7.25 . 3.00 is "7.253"
7.25 . "CS280" is "7.25CS280"
7.25 . "Spring 2023" is "7.25Spring 2023"
7.25 . "25.7" is "7.2525.7"
7.25 . "3.0" is "7.253.0"
3.00 . ERROR is ERROR
3.00 . 7.25 is "37.25"
3.00 . "CS280" is "3CS280"
3.00 . "Spring 2023" is "3Spring 2023"
3.00 . "25.7" is "325.7"
3.00 . "3.0" is "33.0"
. . . . . .
"25.7" . "CS280" is "25.7CS280"
"25.7" . "Spring 2023" is "25.7Spring 2023"
"25.7" . "3.0" is "25.73.0"
"3.0" . ERROR is ERROR
. . . . . .
```


NJIT
THE EDGE IN KNOWLEDGE