
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Fall 2022

Programming Assignment 3
Building an Interpreter

Programming Assignment 3

- Objectives
  - In this programming assignment, you will be building an interpreter for our simple 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:

Programming Language Definition

1. Prog ::= PROGRAM IDENT StmtList END PROGRAM
2. StmtList ::= Stmt; { Stmt; }
3. Stmt ::= DeclStmt | ControlStmt
4. DeclStmt ::= (INT | FLOAT | BOOL) VarList
5. VarList ::= Var { ,Var }
6. ControlStmt ::= AssigStmt | IfStmt | PrintStmt
7. PrintStmt ::= PRINT (ExprList)
8. IfStmt ::= IF (Expr) THEN StmtList [ELSE StmtList] END IF
9. AssignStmt ::= Var = Expr
10. Var ::= IDENT
11. ExprList ::= Expr { , Expr }
12. Expr ::= LogORExpr ::= LogANDExpr { || LogANDRxpr }
13. LogANDExpr ::= EqualExpr { && EqualExpr }
14. EqualExpr ::= RelExpr [== RelExpr]
15. RelExpr ::= AddExpr [(< | >) AddExpr]
16. AddExpr :: MultExpr { (+ | -) MultExpr }
17. MultExpr ::= UnaryExpr { (* | /) UnaryExpr }
18. UnaryExpr ::= (- | + | !) PrimaryExpr | PrimaryExpr
19. PrimaryExpr ::= IDENT | ICONST | RCONST | SCONST | BCONST | (Expr)

Description of the Language

- The language has three types: INT, FLOAT and BOOL.
- A used variable in an expression must have been defined previously by an assignment statement.
- It is illegal to use the name of the program as a variable anywhere in the program.
- The scope of a declared variable extends from the point of its declaration statement to the End of the Program.
- The precedence rules of operators in the language are as shown in the table of operators precedence levels.
- The PLUS, MINUS, MULT, DIV, AND, OR operators are left associative.
- A variable has to be declared in a declaration statement.

Description of the Language

- An IfStmt evaluates a logical expression (Expr) as a condition. If the logical condition value is true, then the StmtList in the Then-part are executed, otherwise they are not. An else part for an IfSmt is optional. Therefore, If an Else-part is defined, the StmtList in the Else-part are executed when the logical condition value is false.
- A PrintStmt evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.
- 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 can be assigned a value of either one of the numeric types (i.e., INT, FLOAT) of the language. For example, an integer variable can be assigned a real value, and a real variable can be assigned an integer value. In either case, conversion of the value to the type of the variable must be applied. A BOOL var in the left-hand side of an assignment statement must be assigned a Boolean value.

Description of the Language

- The binary operations for addition, subtraction, multiplication, and division are performed upon two numeric operands (i.e., INT, FLOAT) of the same or different types. If the operands are of the same type, the type of the result is the same type as the operator's operands. Otherwise, the type of the result is REAL.
- The binary logic operations for the AND and OR operators are applied on two Boolean operands only.
- The LTHAN and GTHAN relational operators and the EQUAL operator operate upon two operands of compatible types. The evaluation of a relational expression, based on LTHAN or GTHAN operators, or an Equality expression, based on the Equal operator, produce either a true or false value.
- The unary sign operators (+ or -) are applied upon one numeric operand (i.e., INT, FLOAT). While the unary NOT operator is applied upon a one Boolean operand (i.e., BOOL).
- It is an error to use a variable in an expression before it has been assigned.

Table of Operators Precedence Levels

Precedence	Operator	Description	Associativity
1	Unary +, -, and !	Unary plus, minus, and logical NOT	Right-to-Left (ignored)
2	*, /	Multiplication and Division	Left-to-Right
3	+, -	Addition and Subtraction	Left-to-Right
4	<, >	Relational operators < and >	(no cascading)
5	==	Equality operator	(no cascading)
6	&&	Logical AND	Left-to-Right
7	||	Logical OR	Left-to-Right

Example of a Program

```
PROGRAM Cylinder
        /*Clean Program testing If-stmt Then-part*/
        INT r, a, h, b;
        r = 8;
        h = 10;
        a = 0;
        FLOAT surface, volume;
        IF ( r > 5 && h > r) THEN
                volume = 3.14 * r * r * h;
                surface = 2 * 3.14 * r * h + 2 * 3.14 * r * r;
                PRINT  ("The output results are " , volume,
                " ", surface, " " , h, " ", r);
        ELSE
                PRINT  ("No computations should be done!" );
        END IF;
END  PROGRAM
```

Interpreter Requirements

- 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.
  - Rename the parse.cpp file as parserInt.cpp to reflect the applied changes on the current parser implementation for building an interpreter.
- The interpreter performs the following:
  - Builds information of variables types in a map container for all the defined variables.
  - Evaluates expressions and determines their values and types.
  - Uses a map container that keeps a record of the defined variables in the parsed program.
  - 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

- Failures due to the process of parsing or interpreting the input program should cause the process of interpretation to stop and return back.
- Generates error messages due to its semantics checking.
  - No specified error messages.
  - 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 PA2.
  - Suggested messages of the interpreter's semantics check 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 copy and use my lexical analyzer when I publish it.
- “parse.cpp”
  - It is provided after deadline of PA2 submissions (including any extensions).
- “parserInt.h”
  - Modified version of “parse.h”.
- Partial “parseInt.cpp”
  - Map containers definitions given in “parse.cpp” for Programming Assignment 2.
  - 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.

Implementation of an Interpreter for the Language

- A map container SymTable that keeps a record of each declared variable in the parsed program and its corresponding type.
- The declaration of a pointer variable to a queue container of Value objects.
- Implementations of the interpreter actions in some functions.

- "val.h"
  - Specification of the Value class.

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;
  Int Itemp;
  Float Rtemp;
  String Stemp;
  Bool Btemp;
public:
  Value() : T(VERR), Itemp(0), Rtemp(0.0){}
  Value(int vi) : T(VINT), Itemp(vi) {}
  Value(float vr) : T(VREAL), Itemp(0), Rtemp(vr) {}
  Value(string vs) : T(VCHAR), Itemp(0), Rtemp(0.0), Stemp(vs) {}
  Value(bool vb) : T(VBOOL), Btemp(vb), Itemp(0), Rtemp(0.0) {}
  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

```
// Overloaded operations
// add op to this
Value operator+(const Value& op) const;
// subtract op from this
Value operator-(const Value& op) const;
// multiply this by op
Value operator*(const Value& op) const;
// divide this by op
Value operator/(const Value& op) const;
Value operator==(const Value& op) const;
Value operator>(const Value& op) const;
Value operator<(const Value& op) const;
Value operator&&(const Value& op) const;
Value operator||(const Value& op) const;
Value operator!() const;//NOT Unary operator
```

```
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(2)
<< op.Rtemp;
    else if(op.IsBool()) out << (op.GetBool()? "true" : "false");
else out << "ERROR"; return out;
 };
```

“parserInt.h” Description

- **“parserInt.h”:** includes the prototype definitions of the parser functions as in “parse.h” header file with the following applied modifications:
  - extern bool Var(istream& in, int& line, LexItem & tok);
  - extern bool LogANDExpr(istream& in, int& line, Value & retVal);
  - extern bool Expr(istream& in, int& line, Value & retVal);//or LogORExpr
  - extern bool EqualExpr(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 PrimaryExpr(istream& in, int& line, int sign, Value & retVal);

“parserInt.cpp” Description

- All definitions from “parse.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 type token (e.g., INT, or FLOAT) when the variable is declared in a declaration statement.

“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 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 by the PrintStmt function.

- Implementations of the interpreter actions in some functions.

Testing Program Requirements

- **“prog3.cpp”:**
  - You are given 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 stop and display the message "Successful Execution", and the program stops.

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

- `PrintStmt` function
  - Grammar rule

    ```
    PrintStmt := PRINT (ExprList)
    ```

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

      ```
      Missing expression after print
      ```

    - Then `PrintStmt` 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: PrintStmt

```
bool PrintStmt(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

Implementation Examples: `LogANDExpr`

- `LogANDExpr` Function Definition

```
bool LogANDExpr (istream & in, int & line, Value & retVal);
```

  - `retVal` object returns the value of ANDing Boolean operands.
  - retValue is a synthesized attribute of the `LogANDExpr non-terminal`.
  - The implementation of the overloaded AND operator method of Value class defines the behavior of the language semantics for the AND operator.
    - What is acceptable and what is not acceptable as operands for the AND operator, including any possible mixed type operands for automatic type conversions.

Implementation Examples: LogANDExpr

```
// LogANDExpr::= EqualExpr { && EqualExpr }
bool LogANDExpr(istream& in, int& line, Value & retVal) {
  Value val1, val2;
  bool t1 = EqualExpr(in, line, val1);
  LexItem tok;
  if( !t1 ) { . . . }
  retVal = val1;
  tok = Parser::GetNextToken(in, line);
  if(tok.GetToken() == ERR){ . . . }
  while ( tok == AND) {
     t1 = EqualExpr(in, line, val2);
     if( !t1 ) { . . . }
     retVal = retVal && val2;
     if(retVal.IsErr())
     { . . . }
     tok = Parser::GetNextToken(in, line);
     if(tok.GetToken() == ERR){ . . .}

  }
  Parser::PushBackToken(tok);
  return true;}
```

Test Cases: Example 1

```
1.  PROGRAM circle
2.       /*Undefined variable*/
3.       FLOAT r, a, p, b;
4.
5.       r = 8;
6.       a = 0;
7.
8.       IF ( r > 5) THEN a = (3.14) * r * r; END IF;
9.       IF (a < 100) THEN p = 2 * 3.14 * b; END IF;
10.      /*Display the results*/
11.      PRINT ( "The output results are " , r, p, b);
12. END PROGRAM
```

- Use print statements to trace execution.
- Use a calculator to check the correction of the evaluated expressions.

Output:

```
11: Undefined Variable
11: Missing Expression
11: Missing expression list after Print
11: Incorrect control Statement.
11: Syntactic error in Program Body.
11: Incorrect Program Body.

Unsuccessful Interpretation
Number of Errors 6
```

Test Cases: Example 2

```
1.  PROGRAM  rectangle
2.       /*Illegal operand type for the NOT operator*/
3.       INT width, length;
4.       FLOAT area, perimeter ;
5.       Width = 8.25;
6.       length = 12.5;
7.       area = !width * length;
8.       perimeter = 2*width + 2 * length;
9.       /*Display the results*/
10.      PRINT ( "The rectangle width and length are " , width, "
    ", length);
11.      PRINT ( "The rectangle area and perimeter are " , area,"
    ", perimeter);
12. END PROGRAM
```

```
7: Illegal Operand Type for NOT Operator
7: Missing Expression in Assignment Statement
7: Incorrect control Statement.
7: Syntactic error in Program Body.
7: Incorrect Program Body.

Unsuccessful Interpretation
Number of Errors 5
```

Test Cases: Example 3

```
1.  PROGRAM Cylinder
2.       /*Clean Program testing If-stmt Then-part*/
3.       INT r, a, h, b;
4.       r = 8;
5.       h = 10;
6.       a = 0;
7.       FLOAT surface, volume;
8.       IF ( r > 5 && h > r) THEN
9.                volume = 3.14 * r * r * h;
10.               surface = 2 * 3.14 * r * h + 2 * 3.14 * r * r;
11.               PRINT  ("The output results are " , volume, " ",
    surface, " " , h, " ", r);
12.      ELSE
13.               PRINT  ("No computations should be done!" );
14.      END IF;
15. END  PROGRAM
```

```
The output results are 2009.60 904.32 10 8

Successful Execution
```

Test Cases: Example 4

```
1.  PROGRAM Cylinder
2.      /*Illegal operands type for Add operator*/
3.      INT r, a, h, b;
4.      BOOL bvar1, bvar2;
5.      r = 8;
6.      h = 10;
7.      a = 0;
8.      FLOAT surface, volume;
9.      bvar1 = (r == 8);
10.     bvar2 = bvar1 ;
11.     PRINT  ("The output results " + "for Boolean variables ",
    bvar1, " ", bvar2);
12. END  PROGRAMD
```

```
11: Illegal Addition operation.
11: Missing Expression
11: Missing expression list after Print
11: Incorrect control Statement.
11: Syntactic error in Program Body.
11: Incorrect Program Body.

Unsuccessful Interpretation
Number of Errors 6
```


NJIT
THE EDGE IN KNOWLEDGE