
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2024

Programming Assignment 3

Building a Simple Fortran 95-Like Language Interpreter

Programming Assignment 3

- Objectives
  - In this programming assignment, you will be building an interpreter for our Simple Fortran 95-Like (SFort95) 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:

SFort95 Language Definition

1. Prog ::= PROGRAM IDENT {Decl} {Stmt} END PROGRAM IDENT
2. Decl ::= Type :: VarList
3. Type ::= INTEGER | REAL | CHARACTER [(LEN = ICONST)]
4. VarList ::= Var [= Expr] {, Var [= Expr]}
5. Stmt ::= AssigStmt | BlockIfStmt | PrintStmt | SimpleIfStmt
6. PrintStmt ::= PRINT *, ExprList
7. BlockIfStmt ::= IF (RelExpr) THEN {Stmt} [ELSE {Stmt}] END IF
8. SimpleIfStmt ::= IF (RelExpr) SimpleStmt
9. SimpleStmt ::= AssigStmt | PrintStmt
10. AssignStmt ::= Var = Expr
11. ExprList ::= Expr {, Expr}

SFort95 Language Definition

```
12. RelExpr ::= Expr [ ( == | < | > ) Expr ]
13. Expr ::= MultExpr { ( + | - | // ) MultExpr }
14. MultExpr ::= TermExpr { ( * | / ) TermExpr }
15. TermExpr ::= SFactor { ** SFactor }
16. SFactor ::= [+ | -] Factor
17. Var ::= IDENT
18. Factor ::= IDENT | ICONST | RCONST | SCONST | (Expr)
```

Example Program of SFort95 Language

```
PROGRAM circle
      !Variable declarations
      REAL :: r, a, p, b
      CHARACTER :: str1, str2
      a = (3.14) * r * r
      IF ( r == 5) THEN
        p = 2 * 3.14 * b
        print *, a, p, r
      else
            str1 = str1  // str2
            print *, str1, str2
      END IF
END PROGRAM circle
```

Description of the Language

- The language has three types: integer, real, and character.
- Declaring a numeric variable does not automatically assign a value, say zero, to this variable, until a value has been assigned to it. Variables can be given initial values in the declaration statements using initialization expressions and literals.

```
INTEGER :: i = 5, j = 100
REAL :: x, y = 1.0E5
```

- CHARACTER variables can refer to one character, or to a string of characters which is achieved by adding a length specifier to the object declaration. In this language, all character variables are assumed to be initialized with blank character(s) based on their lengths, if they are not initialized. The following are all valid declarations of the variables, where grade is of one character, and name is of 10 characters. Both are initialized with blank string.

Description of the Language

```
CHARACTER :: grade
CHARACTER(LEN=10):: name
```

- The following two declarations of the character variable are initialized, where light variable is initialized by the first character of the string ‘Amber’, and the boat variable is initialized by the string 'Wellie' padded, to the right, with 3 blanks.

```
CHARACTER :: light = 'Amber'
CHARACTER(LEN=9) :: boat = 'Wellie'
```

- 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 +, -	Unary plus, minus	Right-to-Left
2	**	Exponentiation	Right-to-Left
3	*, /	Multiplication, Division	Left-to-Right
4	+, -, //	Addition, Subtraction, Concatenation	Left-to-Right
5	<, >, ==	Less than, greater than, and equality	(no cascading)

Description of the Language

- The precedence rules of operators in the language are as shown in the table of operators' precedence levels.
- The PLUS (+), MINUS (-), MULT (*), DIV (/), and CATenation (//) operators are left associative.
- The unary operators (+, -) and exponentiation (**) are right-to-left associative.
- A SimpleIfStmt evaluates a relational expression (RelExpr) as a condition. The If-clause for a SimpleIfStmt is just one Simple statement (SimpleStmt). If the condition value is true, then the If-clause statement is executed, otherwise it is not. No Else-clause is associated with a SimpleIfStmt.

Description of the Language

- An BlockIfStmt evaluates a relational expression (RelExpr) as a condition. If the condition value is true, then the If-clause statements are executed, otherwise they are not. An else clause for a BlockIfSmt is optional. Therefore, if an Else-clause is defined, the Else-clause statements are executed when the condition value is false.
- A PrintStmt statement evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.
- The binary operations of numeric operators as addition, subtraction, multiplication, and division are performed upon two numeric operands.
  - 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 concatenation operator is performed upon two operands of character types (i.e. strings).
- The exponentiation operation is performed on REAL type operands only.

Description of the Language

- Similarly, relational operators (==, <, and >) operate upon two compatible type operands. The evaluation of a relational expression produces either a logical true or false value. For all relational operators, no cascading is allowed.
  - Numeric types (i.e., INTEGER and REAL) are compatible, and evaluation is based on converting an integer type operand to real if the two operands are of different numeric types. However, numeric types are not compatible with CHARACTER type.

Description of the Language

- The ASSOP operator (=) in the AssignStmt assigns a value to a variable. It evaluates the Expr on the right-hand side (RHS) and saves its value in a memory location associated with the left-hand side (LHS) variable (Var). A left-hand side variable of a Numeric type must be assigned a numeric value. Type conversion must be automatically applied if the right-hand side numeric value of the evaluated expression does not match the numeric type of the left-hand side variable.
- While a left-hand side variable of character type must be assigned a value of a string that has the same length as the left-hand variable length in the declaration statement. If the length of the RHS string is greater than the LHS variable the string would be truncated, otherwise the string would be padded by blank characters.

Interpreter Requirements

- Implement an 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 can use the provided implementations for the parser when it is posted. Rename the "parser.cpp" file as "parserInterp.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).
- “parserInterp.h”
  - Modified version of “parser.h”.
- Partial “GivenparserInterpPart.cpp”
- “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 and overloaded operator functions specified in the Value class definition. (Complete the implementations of the overloaded operator functions from RA 8)

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 VarList(istream& in, int& line, LexItem & idtok,
                    int strlen = 1 );
extern bool Var(istream& in, int& line, LexItem & idtok);

extern bool ExprList(istream& in, int& line);
extern bool RelExpr(istream& in, int& line, Value & retVal);
extern bool Expr(istream& in, int& line, Value & retVal);
extern bool MultExpr(istream& in, int& line, Value & retVal);
extern bool TermExpr(istream& in, int& line, Value & retVal);
extern bool SFactor(istream& in, int& line, Value & retVal);
extern bool Factor(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 {
    ValType T; bool Btemp; int Itemp; double Rtemp;
    string Stemp; int strLen;

public:
Value():T(VERR),  Btemp(false),  Itemp(0),  Rtemp(0.0),  Stemp(""),
       strLen(0) {}
Value(bool     vb):T(VBOOL),Btemp(vb),Itemp(0),Rtemp(0.0),Stemp(""),
       strLen(0) {}
Value(int  vi)  :  T(VINT),  Btemp(false),  Itemp(vi),  Rtemp(0.0),
       Stemp(""), strLen(0) {}
Value(double  vr)  :  T(VREAL),  Btemp(false),  Itemp(0),  Rtemp(vr),
       Stemp(""), strLen(0) {}
Value(string vs) : T(VSTRING), Btemp(false), Itemp(0), Rtemp(0.0),
       Stemp(vs), strLen(1) { }
```

“val.h” Description

```
//Getter member functions
int GetInt() const { if( IsInt() ) return Itemp; throw "RUNTIME
ERROR: Value not an integer"; }
. . . . .
. . . . .
//Setter member functions
void SetType(ValType type){ T = type;}
void SetInt(int val){ if( IsInt() ) {Itemp = 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

```
//string concatenation of this with op
  Value Catenate(const Value & op) const;

//compute the value of this raised to the exponent op
  Value Power(const Value & op) const;

. . . .


//Overloaded insertion operator for printing Value objects

  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 out << "ERROR";
          return out;
       }
```

“parserInterp.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, INTEGER, REAL, or CHARACTER).

“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.

“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 14 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. However, a check for the existence of a textual error messages 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.

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).
    - Each expression being parsed is evaluated and its value is 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}
```

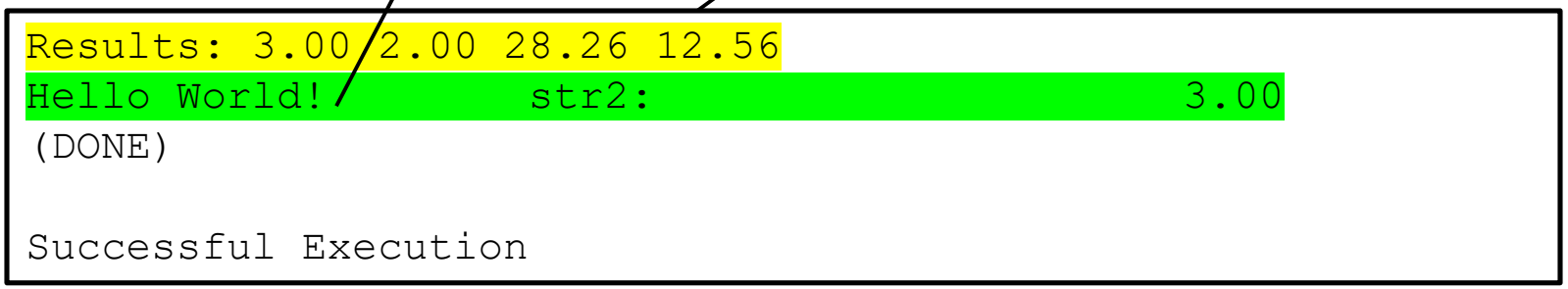
  - Calls `Expr()` function:

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

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

Example 1: Testing Block-If Then-clause and string initialization

```
1.  PROGRAM circle
2.       !Testing Block-If Then-clause and string initialization
3.       REAL :: r=3, a, p, b =2
4.       character (LEN = 20) :: str1 = "Hello World!", str2
5.       a = (3.14) * r * r
6.       IF ( r == 3) THEN
7.         p = 2 * 3.14 * b
8.       END IF
9.       PRINT *, "Results: ", r, " ", b, " ", a, " ", p
10.      print *, str1, "str2: ", str2, r
11. END PROGRAM circle.
```



Example 2: Testing String Catenation

```
1.  PROGRAM circle
2.       !Testing string catenation
3.       REAL :: r, a, p, b =2
4.       character (LEN = 15) :: str1 = "Hello ", str2 = "World!"
5.       character (LEN = 32):: str
6.       r = 3
7.       a = (3.14) * r * r
8.       IF ( r == 5) THEN
9.         IF ( r == 5) THEN
10.        p = 2 * 3.14 * b
11.        print *, a, p, r
12.      else
13.              str = str1  // str2// "?"
14.              print *, str, " ", b
15.      END IF
16.
17.
18. END PROGRAM circle
```

Output:

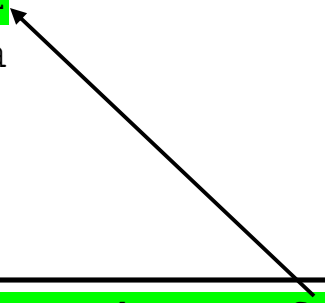
```
Hello           World!          ?  2.00
(DONE)

Successful Execution
```

Example 3: Illegal Operand Type for SIGN Operator

```
PROGRAM circle
       !Illegal operand type for SIGN operator
       real :: r=5, a, p = 2, b = -3
       character :: str1, str2
       a = r * p
       str1 = - str2
       print *, r, a

END PROGRAM circle
```



```
6: Illegal Operand Type for Sign Operator
6: Missing Expression in Assignment Statement
6: Incorrect Statement in program

Unsuccessful Interpretation
Number of Errors 3
```

Example 4: Illegal operand type for the operation

```
1.  PROGRAM circle
2.      !Illegal operand type for the operation
3.      real :: r = 10, a = 0
4.      character :: str
5.      str = "Goodbye"
6.
7.      r = r + str
8.      !Display the results
9.      PRINT *, "The output results are: " , r , ", ", str
10.
11. END PROGRAM circle
```

```
7: Illegal operand type for the operation.
7: Missing Expression in Assignment Statement
7: Incorrect Statement in program

Unsuccessful Interpretation
Number of Errors 3
```


NJIT
THE EDGE IN KNOWLEDGE