CS 280
Spring 2024
Programming Assignment 3

April 15, 2024

Due Date: Sunday, April 28, 2024, 23:59
Total Points: 20

In this programming assignment, you will be building an interpreter for our Simple Fortran 95-Like (SFort95) programming language. The grammar rules of the language and its tokens were given in Programming Assignments 1 and 2. You are required to modify the parser you have implemented for the language to implement an interpreter for it. The specifications of the grammar rules are described in EBNF notations as follows.

```
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}
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)
```

The following points describe the SFort95 programming language. These points that are related to the language syntactic rules were implemented in Programming Assigning 2. However, the points related to the language semantics are required to be implemented in the language interpreter. These points are:

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)

1. The language has three types: integer, real, and character.

2. Declaring a 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
```

3. `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.

```
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'
```

4. The precedence rules of operators in the language are as shown in the table of operators' precedence levels.

5. The PLUS (+), MINUS (-), MULT (*), DIV (/), and CATenation (//) operators are left associative.

6. The unary operators (+, -) and exponentiation (**) are right-to-left associative.

7. A 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.

8. 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.

9. A PrintStmt statement evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.

10. 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 string variable, it would be truncated, otherwise the RHS string would be padded by blank characters.

11. The binary operations for addition, subtraction, multiplication, and division are performed upon two numeric operands (i.e., INTEGER, REAL) 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. While the binary concatenation operator is performed upon two operands of CHARACTER types (i.e. strings).

12. The exponentiation operation is performed on REAL type operands only.

13. Similarly, relational operators (==, <, and >) operate upon two compatible type operands. 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. The evaluation of a

relational expression produces either a logical true or false value. For all relational operators, no cascading is allowed.

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

15. It is an error to use a variable in an expression before it has been assigned.

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. 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 "parserInterp.cpp" to reflect the applied changes on the current parser implementation for building an interpreter. 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.
- **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.

Provided Files

You are given the following files for the process of building an interpreter. These are "lex.h', "lex.cpp", "val.h", "parserInterp.h", and "GivenparserInterpPart.cpp" with definitions and partial implementations of some functions. You need to complete the implementation of the interpreter in the provided copy of “GivenparserInterpPart.cpp” and rename it as “parserInterp.cpp”. “parser.cpp” will be provided and posted.

1. **“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 all the member functions and overloaded operator functions specified in the *Value* class definition. Note: RA 8 included the implementations of some of the member functions of the *Value* class.

2. **“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 Expr(istream& in, int& line, Value & retVal);
extern bool LogANDExpr(istream& in, int& line, Value & retVal);
extern bool RelExpr(istream& in, int& line, Value & retVal);
extern bool SimpleExpr(istream& in, int& line, Value & retVal);
extern bool Term(istream& in, int& line, Value & retVal);
extern bool SFactor(istream& in, int& line, Value & retVal);
extern bool Factor(istream& in, int& line, Value & retVal, int
sign);
```

3. **“GivenparserInterpPart.cpp” includes the following:**

   - Map container definitions given in “parser.cpp” for Programming Assignment 2.
   - A map container SymTable that keeps a record of each declared variable in the parsed program and its corresponding type.
   - 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.
   - The declaration of a pointer variable to a queue container of Value objects.
   - Implementations of the interpreter actions in some example functions.

4. **“parser.cpp”**

   - Implementations of parser functions in “parser.cpp” from Programming Assignment 2.

5. **“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 display a message as "Unsuccessful Interpretation", and display the number of errors detected, then the program stops. 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.

Vocareum Automatic Grading

- You are provided by a set of 14 testing files associated with Programming Assignment 3. Vocareum automatic grading will be based on these testing files. You may use them to check and test your implementation. These are available in compressed archive "PA 3 Test Cases.zip" on Canvas assignment. The testing case of each file is defined in the Grading table below.
- Automatic grading of testing files with no errors will be based on checking against the generated outputs by the executed source program and the output message:
  ```
  Successful Execution
  ```
- In each of the other testing files, there is one semantic or syntactic error at a specific line. The automatic grading process will be based on identifying the statement number at which this error has been found and associated with one or more error messages.
- You can use whatever error message 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.
- A check of the number of errors your parser has produced and the number of errors printed out by the program is also made.

Submission Guidelines

- **Submit your "parserInterp.cpp" and "val.cpp" implementations through Vocareum.**
- **Submissions after the due date are accepted with a fixed penalty of 25%. No submission is accepted after Wednesday 11:59 pm, May 1st, 2024.**

Grading Table

Item	Points
Compiles Successfully	1
test1: Testing initialization of variables	2
test2: Testing evaluation of expressions	2
test3: Testing Block-If Then-clause and string initialization	2
test4: Testing Block-If Else-clause and string assignment	2
test5: Testing string catenation	2
test6: Testing exponentiation evaluation	1
test7: Testing uninitialized variables	1
test8: Illegal operand type for SIGN operator	1
test9: Illegal type for an If condition	1
test10: Testing division by zero	1
test11: Illegal mixed-mode assignment operation	1
test12: Illegal operand type for an arithmetic operator	1
test13: Illegal operand type for a relational operator	1
test14: Incorrect Initialization of a String Length	1
Total	20