CS 280
Fall 2024

Programming Assignment 3
Building an Interpreter for Mini C-Like Language

November 19-25, 2024
Total Points: 20

In this programming assignment, you will be building an interpreter for our Mini C-Like (MCL) 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 CompStmt
2.  StmtList ::= Stmt { Stmt }
3.  Stmt ::= DeclStmt | ControlStmt | CompStmt
4.  DeclStmt ::= ( INT | FLOAT | BOOL | CHAR | STRING ) VarList ;
5.  VarList ::= Var [= Expr] { ,Var [= Expr]}
6.  ControlStmt ::= AssgnStmt ; | IfStmt | PrintStmt ;
7.  PrintStmt ::= PRINT (ExprList)
8.  CompStmt ::= '{' StmtList '}'
9.  IfStmt ::= IF (Expr) Stmt [ ELSE Stmt ]
10. AssgnStmt ::= Var ( = | += | -= | *= | /= | %= ) Expr
11. Var ::= IDENT
12. ExprList ::= Expr { , Expr }
13. Expr ::= LogANDExpr { || LogANDRxpr }
14. LogANDExpr ::= EqualExpr { && EqualExpr }
15. EqualExpr ::= RelExpr  [ (== | != ) RelExpr ]
16. RelExpr ::= AddExpr  [ ( < | > ) AddExpr ]
17. AddExpr :: MultExpr { ( + | - ) MultExpr }
18. MultExpr ::= UnaryExpr { ( * | / | % ) UnaryExpr }
19. UnaryExpr ::= [( - | + | ! )] PrimaryExpr
20. PrimaryExpr ::= IDENT | ICONST | RCONST | SCONST | BCONST |
    CCONST | ( Expr )
```

The following points describe the MCL 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 +, -, and !	Unary plus, minus, and logical NOT	Right-to-Left (ignored)
2	*, /, %	Multiplication, Division, and Remainder	Left-to-Right
3	+, -	Addition and Subtraction	Left-to-Right
4	<, >	Relational operators < and >	(no cascading)
5	==, !=	Equality and Nonequality operators	(no cascading)
6	&&	Logical AND	Left-to-Right
7	||	Logical OR	Left-to-Right

1. The language has five data types: integer (int), float (float), character (char), Boolean (bool), and string (string). In the MCL language, the string type is treated as a built-in type.

2. A variable has to be declared in a declaration statement.

3. Declaring a variable does not automatically assign an initial 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. For example,
```
int i = 5, j = 100;
float x, y = 1.0E5 * i;
```

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 REM (%) operators are left associative.

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

7. An IfStmt evaluates a logical expression as a condition. If the condition value is true, then the If-clause part is executed, otherwise they are not. An Else-clause for an IfSmt is optional. Therefore, if an Else-clause is defined, the Else-clause part is executed when the condition value is false.

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

9. The ASSOP operator (=) and all combined assignment operators in the AssignStmt assign 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). The type of the left-hand side must be compatible with the type of the right-hand side expression. For compatible types, a type conversion must be automatically applied to the right-hand side expression value, if it does not match the type of the left-hand side variable. In MCL language, any numeric expression value type can be assigned to any compatible left-hand numeric variable type. CHAR and INT are compatible for assignment. CHAR type expressions can be assigned to STRING variable. While, BOOL expressions can be assigned to BOOL variables only.

10. Assignment statements based on any of the combined assignment operators (e.g., +=, *=, etc.) have the left-hand side variable in the right-hand side expression as an operand of the combined operator. For example, a statement such as x += y*z -w is equivalent to the statement x = x + y*z -w.

11. The binary operations for addition, subtraction, multiplication, and division are performed upon two numeric operands (i.e., INT, FLOAT) of the same or compatible 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 FLOAT. Note that the CHAR type is compatible with the INT type, but the STRING type is not compatible with numeric types in the MCL language.

12. The remainder operation is performed upon two integer operands, or two operand types that are compatible with INT (i.e., CHAR).

13. Logic binary operators (&& and ||) are applied on two Boolean operands only.

14. In the MCL language, the PLUS operator is treated as an overloaded operator for concatenation when it is applied on two string operands, or applied on one string operand and a char operand.

15. The unary sign operators (+ or -) are applied upon unary numeric operands (i.e., INT, FLOAT). While the unary NOT operator is applied upon a Boolean operand (i.e., BOOL).

16. Similarly, relational operators (==, !=, <, and >) operate upon two operands of the same type or of compatible types. For all relational operators, no cascading is allowed.

17. 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 overloaded operator member functions. 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 overloaded operator member 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.

<u>Provided Files</u>

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". The "GivenparserInterpPart.cpp" file includes 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 overloaded operator member functions that are 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);
extern bool Var(istream& in, int& line, LexItem & idtok);
extern bool ExprList(istream& in, int& line);
extern bool Expr(istream& in, int& line, Value & retVal);
extern bool LogANDExpr(istream& in, int& line, Value & retVal);
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);
```

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 **19** 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.

<u>Submission Guidelines</u>

- **Submit your "parserInterp.cpp" and "val.cpp" implementations through Vocareum.** The "lex.h", "parserInterp.h", "val.h", "lex.cpp" and "prog3.cpp" files will be propagated to your Work directory on Vocareum.
- **Due date of the submission of your section is shown on the Canvas Submission Page of the assignment. Extended submission period of PA 3 will be allowed after the announced due date <u>for 3 days</u> with a fixed penalty of 25% deduction from the student's score.**

Grading Table

Item	Points
Compiles Successfully	1
testprog1: Using uninitialized variable.	1
testprog2: Illegal operand type for Sign operation.	1
testprog3: Illegal operand type for NOT operation.	1
testprog4: Illegal mixed-mode assignment operation.	1
testprog5: Illegal type for if-statement condition.	1
testprog6: Illegal operand type for an arithmetic operation.	1
testprog7: Illegal operand type for a logical operation.	1
testprog8: Illegal operand type for remainder operation.	1
testprog9: Testing division by zero.	1
testprog10: Using unassigned LHS variable in a combined assignment statement.	1
testprog11: Illegal operand for relational operation.	1
testprog12: Testing combined assignment operations += and /=.	1
testprog13: Testing combined assignment operations *=, %=, and -=	1
testprog14: Testing string catenation.	1
testprog15: Testing if-statement then-clause.	1
testprog16: Testing if-statement else-clause.	1
testprog17: Testing if-statement compound statement then/else clause.	1
testprog18: Testing nested if-statement in else-clause.	1
testprog19: Testing nested if-statement in then-clause.	1
Total	20