CS 280
Spring 2025
Programming Assignment 3

Building an Interpreter for Simple Ada-Like Language

April 14, 2025

Due Date: May 2nd, 2025
Total Points: 20

In this programming assignment, you will be building an Interpreter for a Simple ADA-Like programming language, called SADAL. The syntax definitions of the programming language are given below using EBNF notations. Your implementation of the interpreter to the language is based on the following grammar rules.

```
1.  Prog ::= PROCEDURE ProcName IS ProcBody
2.  ProcBody ::= DeclPart BEGIN StmtList END ProcName ;
3.  ProcName ::= IDENT
4.  DeclPart ::= DeclStmt { DeclStmt }
5.  DeclStmt ::= IDENT {, IDENT } : Type [:= Expr] ;
6.  Type ::= INTEGER | FLOAT | BOOLEAN | STRING | CHARACTER
7.  StmtList ::= Stmt  { Stmt }
8.  Stmt ::= AssignStmt | PrintStmts | GetStmt | IfStmt
9.  PrintStmts ::= (PutLine | Put) ( Expr) ;
10. GetStmt := Get (Var) ;
11. IfStmt ::= IF Expr THEN StmtList { ELSIF Expr THEN StmtList }   [ ELSE
    StmtList ] END IF ;
12. AssignStmt ::= Var := Expr ;
13. Expr ::= Relation {(AND | OR) Relation }
14. Relation ::= SimpleExpr [  ( = | /= | < | <= | > | >= )  SimpleExpr ]
15. SimpleExpr ::= STerm {  ( + | - | & ) STerm }
16. STerm ::= [ ( + | - ) ] Term
17. Term  ::= Factor { ( * | / | MOD ) Factor }
18. Factor ::= Primary [** [(+ | -)] Primary ] | NOT Primary
19. Primary ::= Name | ICONST | FCONST | SCONST | BCONST | CCONST | (Expr)
20. Name ::= IDENT [ ( Range ) ]
21. Range ::= SimpleExpr [. . SimpleExpr ]
```

The following modifications have been applied to the language rules in the implementation of the interpreter:

- Removal of the length specifier in the declaration of string variables.
- Removal of the definition of constant symbolic names.

- ICONST definition with an exponent part is removed.
- Nested if-statements are ignored.

The following points describe the SADAL programming language semantic rules. Many of the specified semantic definitions of the language are based on the original Ada programming language. These points are:

Table of Operators Precedence Levels

Precedence	Operator	Description	Associativity
1	not, **	Unary logical NOT, and exponentiation	(no cascading)
2	*, /, mod	Multiplication, Division, and Modulus	Left-to-Right
3	unary +, -	Unary signs plus and minus	(no cascading)
4	Binary +, -, &	Binary operations for addition, subtractions, and concatenation	Left-to-Right
5	=, /=, <, <=, >, >=	Relational operators	(no cascading)
6	and, or	Logical AND and OR	Left-to-Right

1. The language has five data types: Integer, Float, Character, Boolean, and String. In the SADAL language, the string type is treated as a built-in type. However, a *String* type is defined as a one-dimensional array of characters of length, *L*, whose elements are characters, similar to the Ada language definition of a simple string.

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

3. Declaring a variable does not automatically assign an initial value to the variable. Variables can be given initial values in the declaration statements using initialization expressions and literals. For example,
   ```
   x, y_1, w234: Integer:= 0;
   ```

4. A string variable size can be defined implicitly based on an initialization literal. Note that a length specifier for a string variable declaration has been removed from the SADAL grammar.

   ```
   str: String := "Welcome";--str is initialized with length=7
   ```

   In SADAL, a string variable of length *L* has its sequence of characters indexed by integer values in the range 0 to (*L*-1).

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

6. The PLUS (+), MINUS, MULT (*), DIV (/), CATENAT (&) and MOD (mod) operators are left associative.

7. The unary operators (+, -, not) and exponentiation (**) are not cascaded operators.

8. In this language, exponentiation operator is applied on Float types only. The form of the operation is: X ** N. where X is the base and N is the exponent. If N is positive the result is equivalent to the expression: X*X*...X (with N–1 multiplications). With N equal to zero, the result is one. If X is zero the result is 0. With the value of N negative, the result is the reciprocal of using the absolute value of N as the exponent. **In this project, the exponentiation operation is approximated by the C++ <cmath> `pow()` function in the interpreter implementation. However, you need to check for the conditions defined above before using the <cmath> pow function.**

9. The binary operations for addition, subtraction, multiplication, and division are performed upon two numeric operands of the same types. If the operands are of the same type, the type of the result is the same as the type of the operands. SADAL language does not allow mixed-type expressions. However, the concatenation operation is performed upon *String* or *Character* type operands, where the result is a string value with a length equal to the summation of the lengths of the two operands.

10. The remainder operation is performed upon two integer operands only.

11. Logic binary operators (and, or) are applied on two *Boolean* operands only.

12. The unary sign operators (+ or -) are applied upon unary numeric operands (i.e., *Integer*, or *Float*). While the unary *not* operator is applied upon a *Boolean* operand.

13. The concatenation operation (&) is applied upon two String or Character operands, or one String and one Character operand.

14. Similarly, all relational operators (=, /=, <, <=, > and >=) operate upon two operands of the same type. For all relational operators, no cascading is allowed.

15. It is an error to use a variable in an expression before it has been assigned. A reference to a character in a *String* variable can be done by using an index to the location of that character within the string. While, a reference to a substring of a string is specified using a range of indices within the string; where, the lower bound of the range must always be less than or equal to its upper bound.

16. The *AssignStmt* assigns a value to a variable on the left-hand side of the Assignment operator. It evaluates an Expr on the right-hand side of the Assignment operator and saves its value in a memory location associated with the left-hand side variable (Var). The type of the left-hand side variable must match with the type of the right-hand side expression. SADAL language is a strong typed language (similar to Ada) which does not allow mixed-mode assignments. The Assignment operation is also performed in the context of initializing variables in declaration statements. The target of the assignment operation in the declaration statement context is the list of all variables declared by that statement.

17. *IfStmt* conditions are logical expressions that evaluate to either a *true* or *false* value. If the condition value is true, then the list of statements of an *If-Then-clause* or an *Elsif-then-clause*

are executed, otherwise they are not. An *IfStmt* may include zero or more *Elsif-then-clauses*, and an optional *Else-clause*. Therefore, if an *Else-clause* is defined, the *Else-clause* part is executed when all the selection conditions are false. For the execution of an IfStmt, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to *True* or all conditions are evaluated and yield *False*. If a condition evaluates to *True,* then the corresponding list of statements is executed; otherwise none of them is executed.

18. The *Put* and *PutLine* statements evaluate an expression, and print out its value on the standard output device; where, only the *PutLine* statement follows the displayed value by a newline.

19. The *Get* statement reads a value from the standard input device into a variable.

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. 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 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". The "GivenparserInterpPart.cpp" file includes definitions of variables, functions and global container objects. 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 later on.

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 that are specified in the *Value* class definition.

2. **"parserInterp.h" includes the prototype definitions of the parser functions as in "parser.h" header file with the following applied modifications:**

```
extern bool Var(istream& in, int& line, LexItem & idtok);
extern bool Expr(istream& in, int& line, Value & retVal);
extern bool Relation(istream& in, int& line, Value & retVal);
extern bool SimpleExpr(istream& in, int& line, Value & retVal);
extern bool STerm(istream& in, int& line, Value & retVal);
extern bool Term(istream& in, int& line, int sign, Value & retVal);
extern bool Factor(istream& in, int& line, int sign, Value &
retVal);
extern bool Primary(istream& in, int& line, int sign, Value &
retVal);
extern bool Name(istream& in, int& line, int sign, Value & retVal);
extern bool Range(istream& in, int& line, Value & retVal1, Value
& retVal2);
```

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.

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 messages:

```
(DONE)

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>

- Please name your file as "firstinitial_lastname_parser.cpp". Where, "firstinitial" and "lastname" refer to your first name initial letter and last name, respectively. Uploading and submission of your implementations is via Vocareum. Follow the link on Canvas for PA 2 Submission page.
- The "lex.h", "parserInterp.h", "val.h", "lex.cpp" and "prog3.cpp" files will be propagated to your Work Directory.

- **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% deducted from the student's score. No submission is accepted after Sunday 11:59 pm, May 5, 2025.**

Grading Table

Item	Points
Compiles Successfully	1
testprog1: Using uninitialized variable.	1
testprog2: Illegal operand type for Sign operation.	1
testprog3: Procedure name mismatch with closing end identifier.	1
testprog4: Illegal Expression type for the assigned variable.	1
testprog5: Illegal mixed-mode operand types for arithmetic operator.	1
testprog6: Illegal operand type for an arithmetic operation.	1
testprog7: Illegal expression type for an If statement condition.	1
testprog8: Illegal expression type for an elsif-clause condition.	1
testprog9: Testing divide by zero operation	1
testprog10: Testing substring range bounds	1
testprog11: Testing out of bound index value	1
testprog12: Clean program testing exponentiation operation	1
testprog13: Clean program testing if statement then-clause	1
testprog14: Clean program testing if statement elsif-then-clause	1
testprog15: Clean program testing string operations	1
testprog16: Clean program testing if statement else-clause	1
testprog17: Clean program testing Get Statement input	1
testprog18: Clean program testing input of strings	1
testprog19: Clean program testing input of real numbers	1
Total	**20**