CS 280
Spring 2025
Programming Assignment 2

Building a Parser for Simple Ada-Like Language

March 24, 2025

Due Date: April 7, 2025
Total Points: 20

In this programming assignment, you will be building a parser 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 a parser 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 } : [CONSTANT] Type [(Range)] [:= 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. Var ::= IDENT
14. Expr ::= Relation {(AND | OR) Relation }
15. Relation ::= SimpleExpr [  ( = | /= | < | <= | > | >= )  SimpleExpr ]
16. SimpleExpr ::= STerm {  ( + | - | & ) STerm }
17. STerm ::= [ ( + | - ) ] Term
18. Term  ::= Factor { ( * | / | MOD ) Factor }
19. Factor ::= Primary [** [(+ | -)] Primary ] | NOT Primary
20. Primary ::= Name | ICONST | FCONST | SCONST | BCONST | CCONST | (Expr)
21. Name ::= IDENT [ ( Range ) ]
22. Range ::= SimpleExpr [. . SimpleExpr ]
```

The following points describe the SADAL programming language. Most of these points will not be addressed in this assignment. However, they are specified in order to describe the language semantics and what need to be considered in the implementation of an interpreter for the language in Programming Assignment 3. 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, or using a specified length in the declaration statement. Two attributes are associated with a string variable. The first is its maximum length as being defined explicitly or implicitly. The second is the string's current length, which reflects the current length of the string's value. For example,
   ```
   str: String := "Welcome";//str is initialized with max. length=7
   name: String (20);
   //name string is initialized as a null string with current length=0,
   //and max. length = 20
   ```

   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. A named symbolic constant is defined using the *constant* keyword. For example,
   ```
   str : constant string := "Welcome";
   ```

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

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

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

9. 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 the 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, check for the conditions defined above should be applied before using the <cmath> pow function.**

10. 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.

11. 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.

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

13. 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.
    - Assignment of a string variable a string value should take into consideration the maximum target string variable length according to its declaration, which may include the truncation of the string value to fit to the length of the string variable.

14. 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.

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

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

17. 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.

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

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

20. 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.

Parser Requirements:

Implement a recursive-descent parser for the given language. You may use the lexical analyzer you wrote for Programming Assignment 1, OR you may use the provided implementation when it is posted. The parser should provide the following:

- The results of an unsuccessful parsing are a set of error messages printed by the parser functions, as well as the error messages that might be detected by the lexical analyzer. The parser should detect the first discovered syntactic error and return back unsuccessfully.
- If the parser fails, the program should stop after the *Prog()* parser function returns.
- Typically, the first displayed error message is directly associated with the designated test case defined in the Grading Table below. All other following error messages are generated as a consequence of the detected syntactic error, which its message is displayed first.
- The assignment specifies a list of exact error messages that should be displayed by the parser for certain detected syntactic errors, which they are associated with designated test cases, as shown in the given table below for the error messages associated with specified test cases.
- The format of the error messages should be the line number, followed by a colon and a space, followed by some descriptive text. In general, suggested error messages for the none specified test cases in the given table or other syntactic errors might include messages as "Missing semicolon at end of Statement.", "Incorrect Declaration Statement.", “Missing Right Parenthesis”, “Undefined Variable”, “Missing END”, etc.

- In case of successful parsing, the parser should display the list of all the declared variables in the scanned input file in order. Then, it should display the message "(DONE)" on a new line before returning back successfully to the driver program. Check the format of the displayed list of variables associated with the clean programs in testprog17-testprog19.

Provided Files

You are given the header file for the parser, "parser.h" and **an incomplete file for the "parser.cpp", called "GivenParserPart.cpp". You should use "GivenParserPart.cpp" to complete the implementation of the parser.** In addition, "lex.h", "lex.cpp", and "prog2.cpp" files are also provided. The descriptions of the files are as follows:

"parser.h"

"parser.h" includes the following:

- Prototype definitions of the parser functions (e.g., Prog, Stmt, etc.)

"GivenParserPart.cpp"

- A map container that keeps a record of the defined variables in the parsed program, defined as: `map<string, bool> defVar;`
  - The key of the `defVar` is a variable name, and the value is a Boolean that is set to true when the first time the variable has been initialized, otherwise it is false.
- A function definition for handling the display of error messages, called `ParserError`.
- Functions to handle the process of token lookahead, GetNextToken and PushBackToken, defined in a namespace domain called *Parser*.
- Static int variable for counting errors, called `error_count`, and a function to return its value, called `ErrCount()`.
- Implementations of some functions of the recursive-descent parser.

"prog2.cpp"

- You are given the testing program "prog2.cpp" that reads a file name as an argument from the command line. The file is opened for syntax analysis, as a source code for your parser.
- A call to *Prog()* function is made. If the call fails, the program should stop and display a message as "Unsuccessful Parsing ", and display the number of errors detected. For example:

```
Unsuccessful Parsing
Number of Syntax Errors: 3
```

- If the call to *Prog()* function succeeds, the program should stop and display the message "Successful Parsing ", and the program stops.

Vocareum Automatic Grading

- You are provided by a set of 19 test cases associated with Programming Assignment 2. 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 as "PA 2

Test Cases.zip" on Canvas assignment. The testing case of each file is defined in the Grading table below.

- The automatic grading of a clean source code file will be based on checking against the displayed list of declared variables, and the two messages generated by the parser and the driver program, respectively. Check the test cases of clean programs for the format of displayed outputs. For example, testprog17 output displays the following list of declared variables:

```
Declared Variables:
bing, ch, flag, prog17, str1, str2, w123, x, y_1, z

(DONE)
Successful Parsing
```

- The automatic grading process for test cases with unsuccessful parsing will be based on:
  - The statement number at which this first syntactic error has been detected.
  - The first error message being displayed (for specified test cases in the given table).
  - The number of associated error messages being generated with this syntactic error. A check of the number of errors your parser has produced and the number of errors printed out by the driver program is checked.
  - Note, for all the other test cases that are not listed in the table, you can use whatever error message you like for those test cases. There is no check against the contents of the first error message being displayed, but there is a check that a certain textual message being displayed.

Submission Guidelines

- 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", "parser.h", "lex.cpp" and "prog2.cpp" files will be propagated to your Work Directory.
- **Extended submission period of PA 2 will be allowed after the announced due date for 3 days with a fixed penalty of 25% deducted from the student's score. No submission is accepted after Thursday 11:59 pm, April 10, 2025.**

Grading Table

Item	Points
Compiles Successfully	1
testprog1: Incorrect type name	1
testprog2: Variable Redefinition	1
testprog3: Missing Procedure Name	1
testprog4: Missing comma in Declaration Statement	1
testprog5: Missing Semicolon	1
testprog6: Missing PROCEDURE Keyword	1
Testprog7: Missing closing parenthesis	1
Testprog8: Missing opening parenthesis	1
Testprog9: Undeclared variable	1
testprog10: Missing assignment operator	1
Testprog11: Missing Begin keyword for procedure body	1
testprog12: Missing colon in declaration statement	1
testprog13: Missing operator in expression	1
testprog14: Illegal relational expression	1
testprog15: Missing closing end for If statement	1
testprog16: Invalid Else selection	1
testprog17: Clean program testing substring operation syntax	1
testprog18: Clean program testing Elsif selection	1
testprog19: Clean program testing nested If-Then-Else	1
Total	20

Table of Error Messages Associated with Specified Test Cases

Test Case	Error Message
testprog1: Incorrect type name	“Incorrect Declaration Type.”
testprog2: Variable Redefinition	“Variable Redefinition”
testprog3: Missing Procedure Name	“Missing Procedure Name.”
testprog4: Missing comma in Declaration Statement	“Missing comma in declaration statement.”
testprog5: Missing Semicolon	“Missing semicolon at end of statement”
testprog6: Missing PROCEDURE Keyword	“Incorrect compilation file.”
Testprog7: Missing closing parenthesis	“Missing Right Parenthesis”
Testprog8: Missing opening parenthesis	“Missing Left Parenthesis”
Testprog9: Undeclared variable	“Using Undefined Variable”
testprog10: Missing assignment operator	“Missing Assignment Operator”