CS 280
Spring 2024
Programming Assignment 2

March 25, 2024

Due Date: Sunday April 7, 2024, 23:59
Total Points: 20

In this programming assignment, you will be building a parser for a Simple Fortran 95-Like programming language (SFort95). 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 specified in EBNF notations.

```
1.  Prog ::= PROGRAM IDENT {Decl} {Stmt} END PROGRAM IDENT
2.  Decl ::= Type :: VarList
3.  Type ::= INTEGER | REAL | CHARARACTER [(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 programming language. Note that not all of these points will be addressed in this assignment. However, they are listed in order to give you an understanding of the language semantics and what to be considered for implementing an interpreter for the language in Programming Assignment 3. 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. A variable has to be declared in a declaration statement.

3. 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
```

4. `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 `grade` of one character and `name` 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'
```

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 (/), and CATenation (//) operators are left associative.

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

8. 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.

9. 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.

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

11. The ASSOP operator (=) in the AssignStmt assigns 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). 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.

12. 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).

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

14. 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.

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

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

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.
- If the parser fails, the program should stop after the parser function returns.
- The assignment does not specify the exact error messages that should be printed out by the parser; however, the format of the messages should be the line number, followed by a colon and a space, followed by some descriptive text. Suggested messages might include "Missing semicolon at end of Statement.", "Incorrect Declaration Statement.", "Missing Right Parenthesis", "Undefined Variable", "Missing END", etc.
- If the scanning of the input file is completed with no detected errors, the parser should display the message (DONE) on a new line before returning successfully to the caller program. In the case of clean programs, one or more output messages are printed out as well. Check the expected correct outputs of the test cases with clean programs (test1-test5). The following table shows the parser functions that would display specific messages under certain conditions:

Function	When	Message format	Test Case Example
`Decl()`	Character string is defined with a Length value constant.	"Definition of Strings with length of 20 in declaration statement."	test3
`VarList()`	A variable initialization is recognized.	"Initialization of the variable a in the declaration statement."	test2
`SimpleIfStmt()`	A Print or Assignment statements are parsed.	"Print statement in a Simple If statement."	test1
`BlockIfStmt()`	A closing END IF is parsed.	"End of Block If statement at nesting level 1"	test4 and test5

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 testing files 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 specific messages displayed based on the test case, and the two messages generated by the parser and the driver program, respectively:

```
(DONE)
Successful Parsing
```

  Check the test cases of clean programs for the displayed message(s). The specific messages are generated based on the specific language syntax being parsed. For example, the generated output messages for test1 are:

```
Print statement in a Simple If statement.
(DONE)
Successful Parsing
```

- In each of the other testing files, there are one or more syntactic errors. The parser would detect the first discovered syntactic error and returns back unsuccessfully. The automatic grading process will be based on the statement number at which this first syntactic error has been found, and the number of associated error messages with this syntactic error.

- You can use whatever error message you like. There is no check against the contents of the error messages.

- A check of the number of errors your parser has produced and the number of errors printed out by the program are made.

Submission Guidelines

- Submit your "parser.cpp" implementation through Vocareum. The "lex.h", "parser.h", "lex.cpp" and "prog2.cpp" files will be propagated to your Work Directory.
- **Submissions after the due date are accepted with a fixed penalty of 25%. No submission is accepted after Wednesday 11:59 pm, April 10, 2024.**

Grading Table

Item	Points
Compiles Successfully	1
test1: Program with simple if statement	1
test2: Program with initialization of variables in declaration	1
test3: Program with string declaration and length definition	1
test4: Program with Block If statement with Else-clause	1
test5: Program with nested Block If statements	1
test6: Variable declarations with missing comma	1
test7: Variable redefinition	1
test8: Syntax error in declaration statement	1
test9: Invalid identifier	1
test10: Print statement syntax error 1	1
test11: Using an uninitialized variable	1
test12: Missing operator between operands	1
test13: Block If statement missing THEN	1
test14: Block If statement missing END IF	1
test15: Print statement syntax error 2	1
test16: Missing program name	1
test17: Missing End of program	1
Test18: Invalid relational expression	1
test19: Invalid string length definition	1
Total	20