COVER PAGE

CS323 Programming Assignment #2 – Syntax Analyzer

Fill out all entries 1 - 7. If not, there will be deductions!

1. Names:

[Yesenia Huerta]	Section [Tues/Thurs: 2:30 - 3:45 pm]
[Victoria Guzman]	Section [Tues/Thurs: 2:30 - 3:45 pm]
[Luke Piña]	Section [Tues/Thurs: 1:00 - 2:15 pm]
[Ayman Sadek]	Section [Tues/Thurs: 2:30 - 3:45 pm]

2. Assignment Number: [2]

3. Due Date [Sunday, April 7th]

4. Submission Date [Sunday, April 7th]

5. Executable File name [Parse]
(A file that can be executed without compilation by the instructor, such as .exe, .jar, etc - **NOT** a source file such as .cpp)

6. Names of the test case files -

	input test file	output test file
test 1.	[T1.txt]	[T1_output.txt]
test 2.	[T2.txt]	[T2_output.txt]
test 3.	[T3.txt]	[T3_output.txt]

7. Operating System [Linux]
(Window – preferred or Unix/Linux)

To be filled out by the Instructor:

Comments and Grade:

Problem Statement

The purpose of this assignment was to write a syntax analyzer, built on-top of our existing lexical analyzer – with the goal of parsing a text file successfully, if it's syntactically correct. Our first step was to rewrite the Rat24S grammar to remove any left recursion. While using the lexer() function in Assignment 1 to generate tokens, the parser should print to an output file the following: tokens, lexemes, and production rules used.

How to use your program

For the following program, after downloading the zip file locally onto the machine – depending on your chosen IDE, the compilation and execution of the program may vary. However, let's assume (Professor Choi) is running this program on Visual Studio on a <insert OS here>. From here, he will access his terminal and the directory where the folder can be found. After navigating to the appropriate directory, <Insert directory path here> navigate to the chosen folder via cd <directory path> <filename>.

From here, the professor will compile the C++ program. After compilation, please enter "./Parse" into your terminal to begin. Enter the given .txt file into the program's user prompt. Please advise, the file the professor wants to run must be in the program's folder. It is also case-sensitive, so if it is misspelled or mistyped compared to the file name, it will not find it and will return an error message.

1. Enter into ComplierProject folder inside the terminal
2. Enter the compile command: g++ -std=c++20 -o program Parse.cpp
3. After it compiles completely, run executable: ./Parse
4. Then, enter the text file in the terminal. T1.txt , T3.txt
5. The program will then output a txt file starting with the original txt file name in the folder.
    a. If you wish to run your Txt file, it must be in the folder to avoid an error.

Design of your program

Our group's project is essentially designed to read source code from a text file, tokenize it (from Assignment 1), and then parse these tokens according to specified grammar rules. In the following analysis, let's focus on some data structures, algorithms, and key functions that achieve Assignment 2's end-goal.

Data Structures

Firstly, we used vectors (vector<Token>) to store a list of tokens generated by our lexer. These vectors resize themselves automatically when elements are added/removed – hence their dynamic nature. This proved to be an optimal choice for storing tokens, since the number of tokens is unknown before lexical analysis.

Unordered sets (unordered_set<string>) is used to store keywords for quick lookup to determine if a lexeme is a keyword or an identifier. An unordered set is chosen for its average constant time complexity for search options – proving efficient in checking if a lexeme matches any reserved keyword.

The Struct (Token) is a custom data structure to encapsulate details about each token – including its type like keyword, identifier, operator, the actual lexeme, and the line number on which it was found. This allow us to streamline the process of token management throughout the lexical/syntax analysis stages.

Algorithms

The following algorithms were used in order to ensure proper execution of the program: lexical analysis, syntax analysis, and error handling.

For lexical analysis, this process involved iterating over each character in the input file, grouping characters together into tokens based on certain rules (identifiers, keywords, operators). The lexical analyzer employs various checks to determine the type of each token and correctly handles edge cases like comments/string literals.

For syntax analysis, after successful tokenization, the parser examines the sequence of tokens to ensure they follow the defined grammar rules. This involves recursive descent parsing, which is a method where a set of mutually recursive procedures corresponds to the nonterminals of the grammar. For each nonterminal in a production rule, there's a corresponding function in the program that will match and consume tokens based on their expected pattern. The parser processes tokens to validate constructs like function definitions, variable declarations, and control flow statements. The program also includes mechanisms for handling unexpected tokens or syntax errors by throwing exceptions. This allows for error reporting when the source code doesn't comply with the expected grammar. Our parser function uses a match function to check if the current token matches the expected type. Additional functions like parseStatement, parseExpression serve to parse different grammatical constructs according to the Rat24S rules.

Any Limitation

print errors on the terminal and not on a text file

Any shortcomings

It only reads one line

Rules:

2) Syntax rules:
The following BNF describes the Rat24S.

R1. <Rat24S> ::= $ <Opt Function Definitions> $ <Opt Declaration List> $ <Statement List> $ Yesenia

R2. <Opt Function Definitions> ::= <Function Definitions> | <Empty> Yesenia

R3. <Function Definitions> ::= <Function> | <Function> <Function Definitions> Yesenia

R4. <Function> ::= function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>Yesenia

R5. <Opt Parameter List> ::= <Parameter List> | <Empty> Yesenia

R6. <Parameter List> ::= <Parameter> | <Parameter> , <Parameter List> Yesenia

R7. <Parameter> ::= <IDs > <Qualifier> Yesenia

R8. <Qualifier> ::= integer | boolean | real Yesenia

R9. <Body> ::= { < Statement List> }Yesenia

R10. <Opt Declaration List> ::= <Declaration List> | <Empty> Yesenia

R11. <Declaration List> := <Declaration> ; | <Declaration> ; <Declaration List> Victoria

R12. <Declaration> ::= <Qualifier > <IDs> luke

R13. <IDs> ::= <Identifier> | <Identifier>, <IDs> Victoria

R14. <Statement List> ::= <Statement> | <Statement> <Statement List> Victoria

R15. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While> luke

R16. <Compound> ::= { <Statement List> }luke

R17. <Assign> ::= <Identifier> = <Expression> ; luke

R18. <If> ::= if (<Condition>) <Statement> endif | if (<Condition>) <Statement> else <Statement> endif Victoria

R19. <Return> ::= return ; | return <Expression> ; Victoria

R21. <Print> ::= print (<Expression>); luke

R22. <Scan> ::= scan (<IDs>); luke

R23. <While> ::= while (<Condition>) <Statement> endwhile luke

R24. <Condition> ::= <Expression> <Relop> <Expression> Victoria

R25. <Relop> ::= == | != | > | < | <= | => Victoria

R26. <Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term> Victoria

R27. <Term> ::= <Term> * <Factor> | <Term> / <Factor> | <Factor> Victoria

R28. <Factor> ::= - <Primary> | <Primary> Victoria

R29. <Primary> ::=<Identifier> | <Integer> |<Identifier> (<IDs>) | (<Expression>) | <Real> | true | false Victoria

R30. <Empty>::=

Note: <Identifier>, <Integer>, <Real> are token types as defined in section (1) above

left recursion

Backtracking

Step 1: Write all the rules 28, but there will be more after left & backtracking fixes(Should be around 35)

1. rewritten R26:
   <Expression> ::= <Term> <ExpressionPrime>
   <ExpressionPrime> ::= + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | ε

2. rewritten R27:
   <Term> ::= <Factor> <TermPrime>
   <TermPrime> ::= * <Factor> <TermPrime> | / <Factor> <TermPrime> | ε

3. rewritten R3: <FD> → <function> <FD’>
   i. <FD’> → <FD> or Empty(epsilon)

4. rewritten R6: <PL> → <Parameter> <PL’>
   i. <PL’> → <PL> or Empty(epsilon)

5. rewritten R11: <DL> → <Declaration><DL’>
   i. <DL’> → <DL> or empty (epsilon)

6. rewritten R13: <IDS> → <Identifier> <IDS’>
   i. <IDS’> → <IDS> or empty(epsilon)

7. rewritten R14: <SL> → <Statement><SL’>
   i. <SL’> → <Statement> or Empty(epsilon)

8. rewritten R18:
   <if> ::= if (<Condition>) <Statement> <Prime if>
   <Prime if> ::= endif | else <Statement> endif

9. rewritten R19:
   <Return>::= return; | <return’>
   <return’::=<Expression> ; | empty(epsilon)