SRS Documentation on:

Syntactic Analyzer

Session 2020-2020

CPSC 323

Submitted By: Zach Serna, Nicole Serna Submitted To: Professor Le

**Table of Contents:**

**I. Introduction**

1.1 Purpose

1.2 Document Conventions

1.3 Intended Audience and Suggestions

1.4 Project Scope

1.5 References

**II. Overall Description**

2.1 Product Perspective

2.2 Product Features

2.3 Product Utilization

2.4 Files and Functions

2.5 Product Production/Rules

2.6 Design and Implementation Constraints

2.7 Assumptions and Dependencies

**III. System Features**

3.1 Functional Requirements

**IV. External Interface Requirements**

4.1 User Interfaces

4.2 Software Interfaces

4.3 Communication Interfaces

**V. Nonfunctional Requirements**

5.1 Performance Requirements

5.2 Safety Requirements

5.3 Security Requirements

5.4 Software Quality Attributes

**I: Introduction**

**1.1 Purpose:**

The main purpose of the assignment was to design a syntactic analyzer. The syntactic analyzer should be capable of reading in a file and break down each statement into non terminals. From there each non terminal should reach a terminal state to confirm that it is a syntactically correct line of code. The assignment is divided into 3 main parts: Arithmetic Expressions, Assignment Statements and Declarative Statements. The syntactical analyzer can be implemented using a number of methods including RDP and or a table driven predictive parser. To receive maximum credit, the parser should print tokens, lexemes and the production rules used as well as the parse tree used for analyzing the tokens.

**1.2 Document Conventions:**

There are no notable document conventions in this assignment aside from acronyms such as RDP.

**1.3 Intended Audience and Suggestions**

The intended audience is those who have a general understanding of computer science. It is expected they will understand the general concept of what the syntactic analyzer as well as the lexical analyzer is designed to do. This is to make the overall understanding of the stack based approach of a syntactic analyzer easier to follow and understand when explained below.

**1.4 Project Scope**

The main design of the program is the recursive descent parser method (RDP) utilizing the stack based approach to store our lexemes and their respective tokens. Along with this we have also integrated our lexical analyzer into our program to work with the syntactic analyzer. The lexical analyzer will process the lexemes and tokens which will then be further processed until a semicolon is reached, implying a statement has been completed. Upon encountering an error the user will be notified that an improper statement has been encountered followed by the program ending.

**1.5 References**

The material provided by professor Anthony Le was referenced when constructing the syntactic analyzer as well as the lexical analyzer.

**II: Overall Description**

**2.1 Product Perspective**

The syntactic analyzer functions using the RDP method with utilization of a stack for storing the lexemes and tokens. Each nonterminal has a respective function tied to it and will call any other non-terminals associated with it. For example, the <Expression> -> <Term> <ExpressionPrime>; rule would call the term function, which would in turn call all the non terminals associated with the <Term> rule until a terminal is reached. It would then return true for the <Term> rule and proceed to call the <ExpressionPrime> function and follow a similar process. When the semicolon is reached, more lexemes would be loaded into the stack and this would repeat until all lexemes have been processed. The print function has been removed from the Lexical Analyzer and replaced with print statements that occur when a non-terminal function is reached. The DisplayandPop function has also been created to display the lexemes associated with each production rule. In the situation where an error is encountered, the DisplayError function will be called, passing the error type as a string parameter. The program will then end.

**2.2 Product Features**

The program features many functions, most of which function similarly in the case of the nonterminal functions and the terminal functions. The specific and more notable functions will be explained in the section below.

**2.3 Product Utilization**

The program utilizes the RDP method. Similar to the Lexical Analyzer, the only requirement from the user is to provide a test.txt file with text inside for the program to decipher. From there all they have to do is execute the program and the rest will be handled automatically. The main function will call the Lexical Analyzer to parse through each lexeme in test.txt file and assign them to their respective token. These tokens will be placed into a vector. From there a temporary vector will be created to read in the lexemes until a semicolon is reached. When a semicolon is reached it notifies the program that a full statement has been obtained and from there the lexemes in the temporary vector will be added into a stack. The statement function will be called as it is the starting state and from the Syntactical Analyzer will determine whether it is a Declarative statement or an Assignment statement. The program will continue to follow the respective rules (Production rules will be mentioned below) until a terminal is reached which will result in the program returning true. The Syntactical Analyzer will print out the lexemes and tokens lexeme by lexeme. The production rules used to reach their respective terminals will also be printed out below the lexemes. When a terminal is reached for a lexeme, the lexeme will be popped and a new lexeme will be analyzed. When a semicolon is reached the program will know that the end of the statement has been reached and will proceed to retrieve more lexemes from the original vector and add them to the stack. This will continue until the end of the vector has been reached resulting in the program ending. As opposed to the Lexical Analyzer, which printed out the lexemes and their respective tokens at the END of the program.

**2.4 Files and Functions**

**Files:**

There are several files attached to this program. They will be listed below.

* Test.txt:
* Users will input statements into this file which will then be converted into lexemes and their respective tokens. They will then be syntactically analyzed.
* Syntac.h:
* Location of all the functions tied to the syntactic analyzer
* Lexer.h:
* Location of all the functions tied to the syntactic analyzer
* Assignment\_2\_FINAL.cpp:
* Essentially the main function of the program. Connects all the other files together and runs them.

**Functions:**

There are many functions within the syntactic analyzer; however, many of the nonterminal functions function almost identically. Said functions will be stated as such while the more notable functions will go into more extensive detail below. The functions in Lexical Analyzer will not be mentioned. This is because they were already discussed in the Assignment 1 Documentation.

**Display:**

* Returns nothing. Displays the lexeme and token on top of the stack.

**DisplayandPop:**

* Returns nothing. Pops the token on the top of the stack then displays the next token if it is not the end of the file marker.

**Pop:**

* Pops the token on the top of the stack.

**Identifier:**

* Returns bool. Tests if the token is an identifier, if so, returns true.

**RealNumber:**

* Returns bool. Tests if the token is a number, if so, returns true.

**Factor:**

* Returns bool. Factor calls Identifier or RealNumber to see if Factor exists, if so DisplayandPop is called and the function returns true.
* Else, Factor checks if current lexeme is a “(” (left parenthesis), if it is, DisplayandPop is called followed by an Expression call. If Expression returns true, getLexeme will be called to check if lexeme is “)” (right parenthesis). If true, DisplayandPop is called and the function returns true.
* Otherwise, false is returned and ErrorHandler is called.

**Term:**

* Returns bool. Calls Factor and if it returns true, calls TermPrime to test. If any of these return false, the program returns false.

**TermPrime:**

* Returns bool. Calls getLexeme to check if EOF marker has been reached, if so return true.
* Calls getLexeme to check if it is equal to “\*” or “/”, if so DisplayandPop is called. Factor is then called and if it returns true, TermPrime is called once again.
* If TermPrime doesn’t meet any of these conditions, it has reached an epsilon case which is displayed to the console. True is then returned.

**Expression:**

* Returns bool. Calls Term and if it returns true, calls ExpressionPrime to test. If any of these return false, the program returns false.

**ExpressionPrime:**

* Returns bool. Calls getLexeme to check if EOF marker has been reached, if so return true.
* Calls getLexeme to check if it is equal to “+” or “-”, if so DisplayandPop is called. Term is then called and if it returns true, ExpressionPrime is called once again.
* If ExpressionPrime doesn’t meet any of these conditions, it has reached an epsilon case which is displayed to the console. True is then returned.

**Statement:**

* Returns bool. Calls Display.
* Calls Assign and if Assign is true, Statement will call getLexeme to check if it is equal to “;”. If this is the case, Pop is called and getLexeme will be called again to check if it is equal to “$$” (EOF marker). If not, Statement is recursively called, otherwise true is returned.
* If Assign fails, calls Declarative. If Declarative is true, Statement will call getLexeme to check if it is equal to “;”. If this is the case, Pop is called and getLexeme will be called again to check if it is equal to “$$” (EOF marker). If not, Statement is recursively called, otherwise true is returned.
* If both Assign and Declarative fail, we have reached an invalid statement and ErrorCall will be called.

**Assign:**

* Returns bool. Calls GetToken to check if it was an identifier, if it is true, DisplayandPop is called. Assign will then call getLexeme to check if lexeme is equal to “=” , if it is equal, DisplayandPop is called and then Expression will be called to check if it is true. If any of these checks fail it will return false.

**Declarative:**

* Returns bool. Calls Type. If Type returns true, Identifier is called. If Identifier returns true, DisplayandPop is called and true is returned.

**Type:**

* Returns bool. Calls getLexeme to check if the top lexeme is equal to int, bool, or float AND calls getToken to ensure it is a keyword. If both these conditions are met, DisplayandPop is called and true is returned.

**ErrorCall:**

* Returns nothing. Displays error message passed to the function.

**2.5 Product Production/Rules**

**Arithmetic Rules:**

<Expression> -> <Term> <ExpressionPrime>

<ExpressionPrime> -> + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | epsilon

<Term> -> <Factor> <TermPrime>

<TermPrime> -> \* <Factor> <TermPrime> | <Factor> <TermPrime> | epsilon

<Factor> -> (<Expression>) | <Identifier> | <RealNumber>

<Identifier> -> id

<RealNumber> -> integer

**Assignment Rules:**

<Statement> -> <Assign>

<Assign> -> <Identifier> = <Expression>;

**Declarative Rules:**

<Statement> -> <Declarative>

<Declarative> -> <Type> <Identifier>

<Type> -> bool | float | int

**2.6 Design and Implementation Constraints**

No constraints were met with coding this specific assignment. Due to the advent of COVID-19 however we were unable to progress to the machine code portion of the assignment 3 which was truly unfortunate.

**2.7 Assumptions and Dependencies**

When composing statements on test.txt, it is required that the user input a “;” at the end of each statement otherwise the program will not function correctly. Syntactically incorrect statements will also cause the program to notify the user which statement is incorrect, followed by the program ending.

**III. System Features**

**3.1 Functional Requirements**

The program requires minimal computing power to execute properly. While the program was developed on Windows,The Syntactic Analyzer does not modify or involve usage of any of the registers (down at the assembly level), as a result of this a Mac could likely run the program fine as well

**IV. External Interface Requirements**

**4.1 User Interfaces**

This program primarily focuses on user inputs and system outputs over ease of access and UI. Any interaction the user may have with the program is strictly done prior to the program being done. As a result User Interface is not applicable in this scenario.

**4.2 Software Interfaces**

The Syntactic Analyzer was created using Visual studio which features a plethora of options to manage code and run the program.

**4.3 Communication Interfaces**

There is no physical/ circuit connection; however, the Syntactic Analyzer has been coded to function with the Lexical Analyzer. The Lexical Analyzer will process the lexemes which will then be processed by the Syntactic Analyzer.

**V. Nonfunctional Requirements**

**5.1 Performance Requirements**

The program performs well when tested on a single statement or multiple statements. Performance speed does not decrease as the amount of statements increases.

**5.2 Safety Requirements**

There are no real safety requirements; however, to ensure the program properly functions it is important that all code statements end with a semicolon.

**5.3 Security Requirements**

Not applicable in this situation.

**5.4 Software Quality Attributes**

Performance: The Syntactic Analyzer performs the task it was designed to do. There is room for improvement likely as optimization is always possible.

Testability: The program is able to be tested easily. All the tester must do is modify the test.txt file to see how the Syntactic Analyzer handles statements.

Usability: The program can easily be used to decipher code statements to analyze whether they have correct syntax or not. Statements not included in the productions of the program will not be picked up.