CS323 Documentation

Assignment 2 - Syntax Analyzer

Group Members:
Rosa Cho
Preet Desai
Cuong Pham
Christin Takeyama
Kien Truong

1. Problem Statement

We were assigned to create a Syntactical Analyzer using C++ as our programming language. The Syntax Analyzer must analyze the structure of the code statements based on a set of production rules and check for errors. Utilizing grammatical rules based on Chomsky's works, we established a foundation defining what is terminal, non-terminal, unique starting symbols, and a finite set of productions. Furthermore, we were able to utilize our Lexical Analyzer to help break down inputs into identifiable tokens. From there, the code is parsed and the results printed into a text file.

1.1 Purpose

This program is used to analyze code and then break it down into its semantic and syntactic components.

1.2 Document Conventions

C++ guidelines utilizing GCC conventions and rules

1.3 Intended Audience

Intermediate to advanced programmers, C++ programmers

1.4 Product Scope

A syntactic analyzer capable of taking the tokens formed by a Lexical Analyzer and organizing them syntactically.

1.5 References

Introduction to Compilers by Thomas W. Parsons
Lectures by Anthony Le

2. How to use your program

1. Provide text file as input.
2. LexicalAnalyzer.cpp will read the file line-by-line, separate it into tokens.
3. Have each token converted to a certain state via the state table (aka an array).
4. Each token will then be pushed to a token list in the SyntaxAnalyzer object.

5. Upon encountering a newline character, GrammarCheck() function will be called to analyze the structure of the statement.
6. Repeat with each line until the end of the file.
7. Each line will then be printed onto an output.txt file with tokens, lexemes and production rules.
8. End program

- Steps to execute the program on the terminal running Ubuntu 20.04 or Tuffix
    - $ sh run.sh
    - $ ./proj2.out input1.txt (Format ./proj2.out <input file>)
    - $ cat output.txt
        - Notice: The output will be inside the output.txt file

3. Design of your program

For Project 2, we are building on top of our Lexical Analyzer program. We created a new SyntaxAnalyzer class which is located in the SyntaxAnalysis.h and SyntaxAnalysis.cpp file. We use a *recursive descent parser* algorithm (RDP) to implement our Syntax Analyzer. The class contains public functions: Push(), Pop(), IsEmpty(), PrintAll(), and GrammarCheck(). The push function takes two arguments, string token and string lex. These values are used to push a struct instance (Token) to a list of struct instances called tokenLists. The pop() function pops a Token of type struct from the front of tokenLists. IsEmpty() returns a boolean value depending on if the list is empty or not. PrintAll() is mostly for testing purposes, prints out all tokens from tokenLists. The GrammarCheck() function is a critical component of our program. Every time the lexical analyzer encounters a newline character, the GrammarCheck() function is called to analyze the structure of a statement. The GrammarCheck() function references private boolean functions responsible for analyzing the statement based on given production rules.

3.1 Product Perspective

This product is a culmination of what we have learned so far: lexical analyzers, syntax analyzers, and the parse tables involved.

3.2 Production Rules

The functions D, DPrime, and Type represent the following productions:

* <D> -> <Type> id <DPrime> ; | epsilon
* <DPrime> -> , id <DPrime> | epsilon

The function A represents the following production:

* <A> -> id = <E> ;

The functions E, EPrime, T, TPrime, F, and id represent the following productions:

* <E> -> <T> <EPrime>
* <EPrime> -> + <T> <EPrime> | - <T> <EPrime> |epsilon

* <T> -> <F> <TPrime>
* <TPrime> -> * <F> <TPrime> | / <F> <TPrime> | epsilon
* <F> -> (<E>) | i | num
* i -> id

The function S (Statement) represents the following production:
* <S> -> <A> | <D> | <W> ; W - while loop

The functions W, C, and R represent the following productions:
* <W> -> 'while' <C> 'do' <S> 'whileend' ;
* <C> -> <E> <R> <E> | <E>
* <R> -> < | > | <= | >= | == | <>

The function I, C, CPrime, and A represent the following productions:
* <I> -> 'if' <C> 'then' <S> 'else' <S> 'endif';
* <C> -> <E> <CPrime>
* <CPrime> -> <R> <E> | epsilon
* <A> -> id = <E> ;

The GrammarCheck() function will call each of these functions and print out a message depending on the boolean value returned. If any syntax errors are encountered, a message will be printed describing the error and the program will be terminated.

3.2 Product Output

Input: *if a >=2 then b = c+2 else b = c-2 endif;*

Token: Keyword Lexeme: if
Token: Identifier Lexeme: a
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> <Identifier>
<TPrime> -> epsilon.
<EPrime> -> epsilon.
Token: Compound Operator Lexeme: >=
<R> -> < | > | >= | <= | <> | ==
Token: Integer Lexeme: 2
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> num
<TPrime> -> epsilon.
<EPrime> -> epsilon.
Token: Keyword Lexeme: then
Token: Identifier Lexeme: b

<Statement> -> <Assign>
<Assign> -> <Identifier> = <Expression>
Token: Operator Lexeme: =
Token: Identifier Lexeme: c
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> <Identifier>
<TPrime> -> epsilon.
Token: Operator Lexeme: +
<EPrime> -> + <Term> <EPrime>
Token: Integer Lexeme: 2
<Term> -> <Factor> <Term Prime>
<Factor> -> num
<TPrime> -> epsilon.
<EPrime> -> epsilon.
Token: Keyword Lexeme: else
Token: Identifier Lexeme: b
<Statement> -> <Assign>
<Assign> -> <Identifier> = <Expression>
Token: Operator Lexeme: =
Token: Identifier Lexeme: c
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> <Identifier>
<TPrime> -> epsilon.
Token: Operator Lexeme: -
<EPrime> -> - <Term> <EPrime>
Token: Integer Lexeme: 2
<Term> -> <Factor> <Term Prime>
<Factor> -> num
<TPrime> -> epsilon.
<EPrime> -> epsilon.
Token: Keyword Lexeme: endif
Token: Separator Lexeme: ;
It's an if statement

Since the design of our program is built on the previous assignment, the Lexical Analyzer is being run with the new additional features. A recap of the program consists of a majority of the data structures used to create our FSM is held within the LexicalAnalyzer.cpp file. The first three functions following the constructor and destructor are booleans which, along with their helper functions FindChar and FindString, determine whether the first inputted value at hand is a separator, operator, or a keyword. The function ConvertCharToCol is what follows up to take the value at hand and then determine what to categorize it as through a series of if/else statements. The if/else statements can be divided into three different ways: the ones that determine if they are

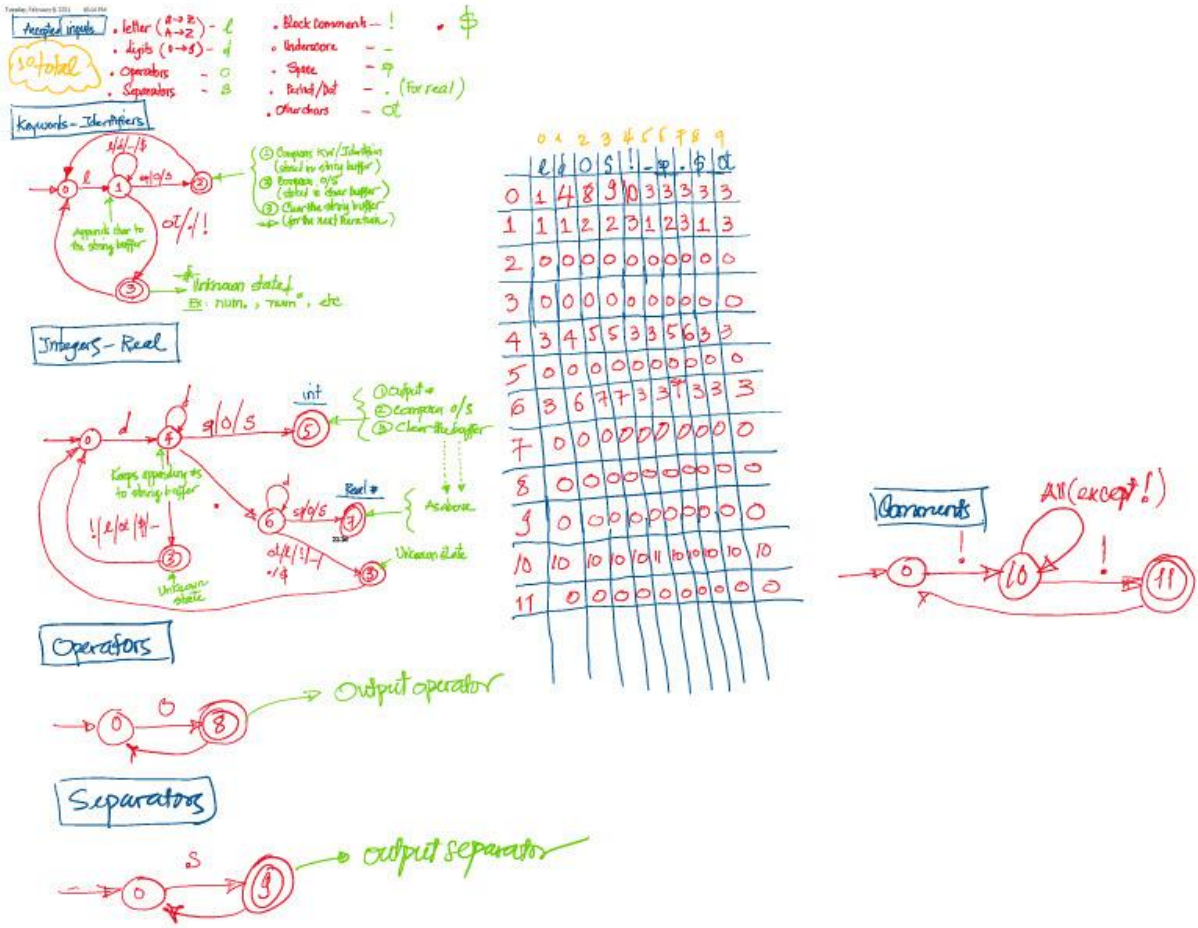
letters/operators/separators i.e. the ones that require their own functions; the ones that determine if they are certain characters (!, _, a white space, . (for floats), or %); and finally the ones that determine that a value is its own special case. A state table (as represented as a matrix) is then used to act as the vehicle for our FSM. Finally, the data is printed onto an outputted text file. The file should look like this:

TOKENS		*LEXEMES*
KEYWORD	=	*while*
SEPARATOR	=	*(*
IDENTIFIER	=	*fahr*
OPERATOR	=	*<*
IDENTIFIER	=	*upper*
SEPARATOR	=	*)*
IDENTIFIER	=	*a*
OPERATOR	=	*=*
REAL	=	*23.00*
KEYWORD	=	*whileend*
KEYWORD	=	*int*
IDENTIFIER	=	*num1*
SEPARATOR	=	*,*
IDENTIFIER	=	*num2$*
SEPARATOR	=	*,*
IDENTIFIER	=	*large_num*
SEPARATOR	=	*;*
KEYWORD	=	*if*
SEPARATOR	=	*(*
IDENTIFIER	=	*num1*
OPERATOR	=	*>*
IDENTIFIER	=	*num2$*
SEPARATOR	=	*)*
SEPARATOR	=	*{*
IDENTIFIER	=	*large_num*
OPERATOR	=	*=*
IDENTIFIER	=	*num1*
SEPARATOR	=	*;*
SEPARATOR	=	*}*
KEYWORD	=	*else*
SEPARATOR	=	*{*
IDENTIFIER	=	*large_num*
OPERATOR	=	*=*
IDENTIFIER	=	*num2$*
SEPARATOR	=	*;*
SEPARATOR	=	*}*

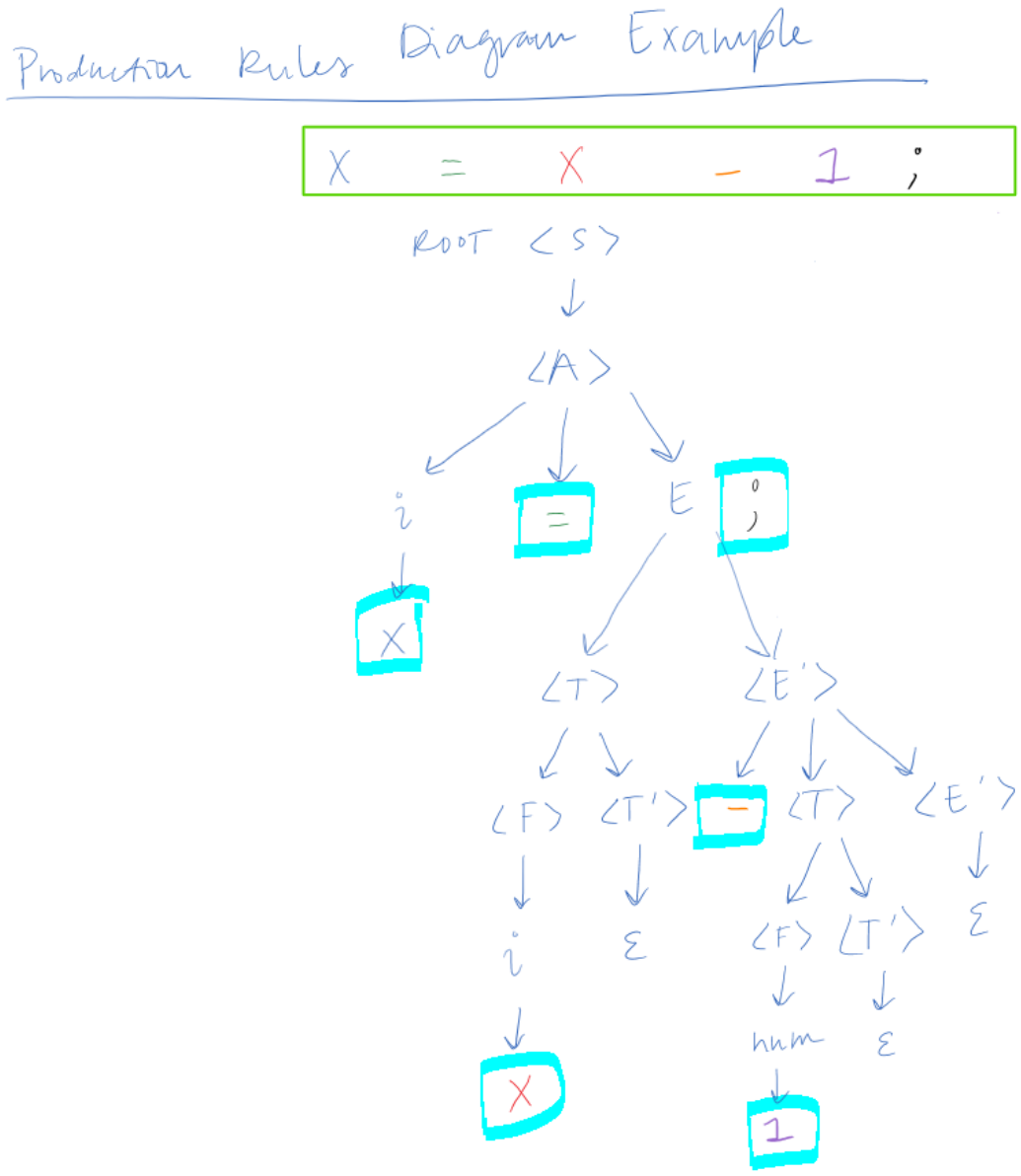
As you can see, the data is organized into three columns: token type, =, the value. The main.cpp ultimately writes the output file that will be printed.

FSM - Diagram (Credit: Kien Truong)

The FSM diagram below displays how different inputs are tokenized.



Production Rules - Diagram (Credit: Christin Takeyama)



4. **Any Limitation**
   None

5. **Any shortcomings**
   None, we essentially accomplished what the instructions wanted.