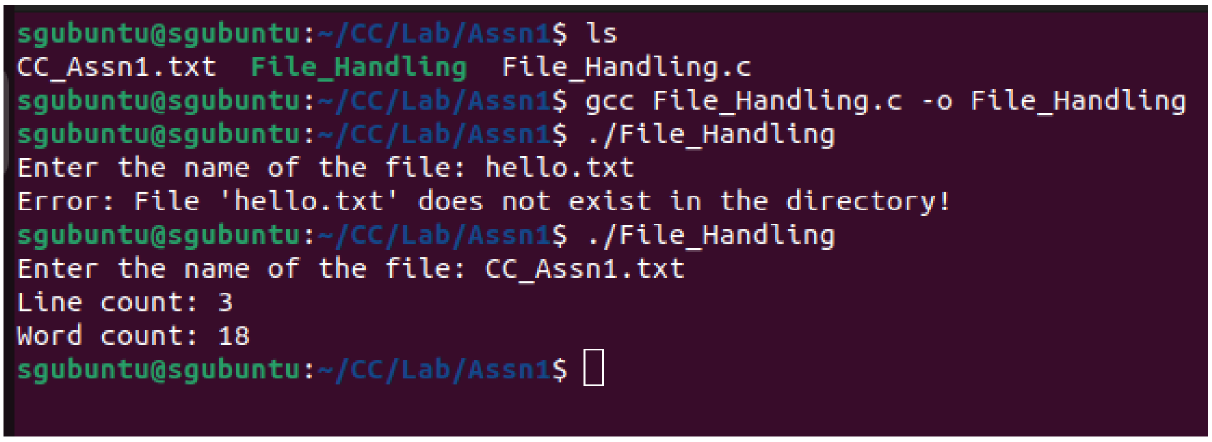
112103046 - CC Assignment

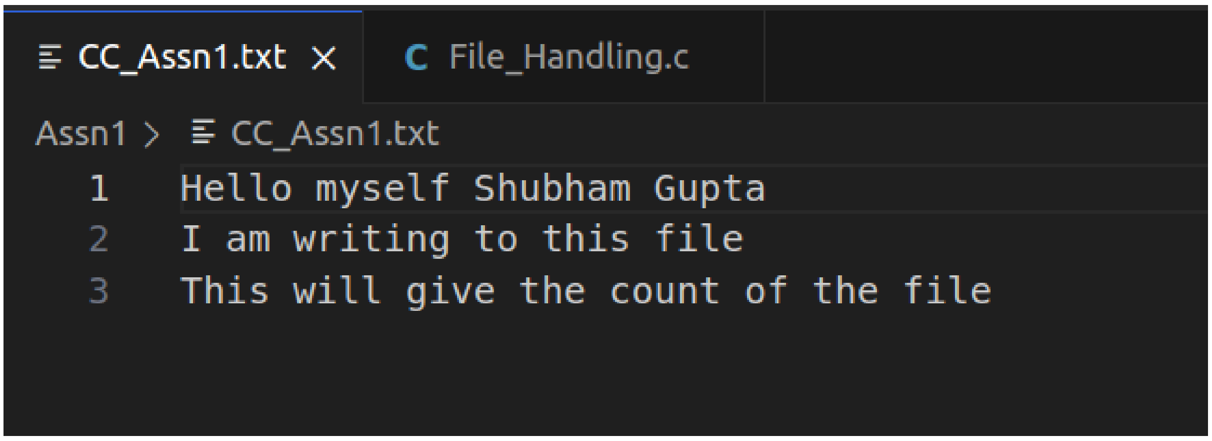
Name : Shubham Gupta
Batch : Div-1 CS2

1) Write a program to accept the text file and count the total number of lines , total words in a file.
->
Input and Output:



```
sgubuntu@sgubuntu:~/CC/Lab/Assn1$ ls
CC_Assn1.txt  File_Handling  File_Handling.c
sgubuntu@sgubuntu:~/CC/Lab/Assn1$ gcc File_Handling.c -o File_Handling
sgubuntu@sgubuntu:~/CC/Lab/Assn1$ ./File_Handling
Enter the name of the file: hello.txt
Error: File 'hello.txt' does not exist in the directory!
sgubuntu@sgubuntu:~/CC/Lab/Assn1$ ./File_Handling
Enter the name of the file: CC_Assn1.txt
Line count: 3
Word count: 18
sgubuntu@sgubuntu:~/CC/Lab/Assn1$
```

CC_Assn1.txt:



```
Hello myself Shubham Gupta
I am writing to this file
This will give the count of the file
```

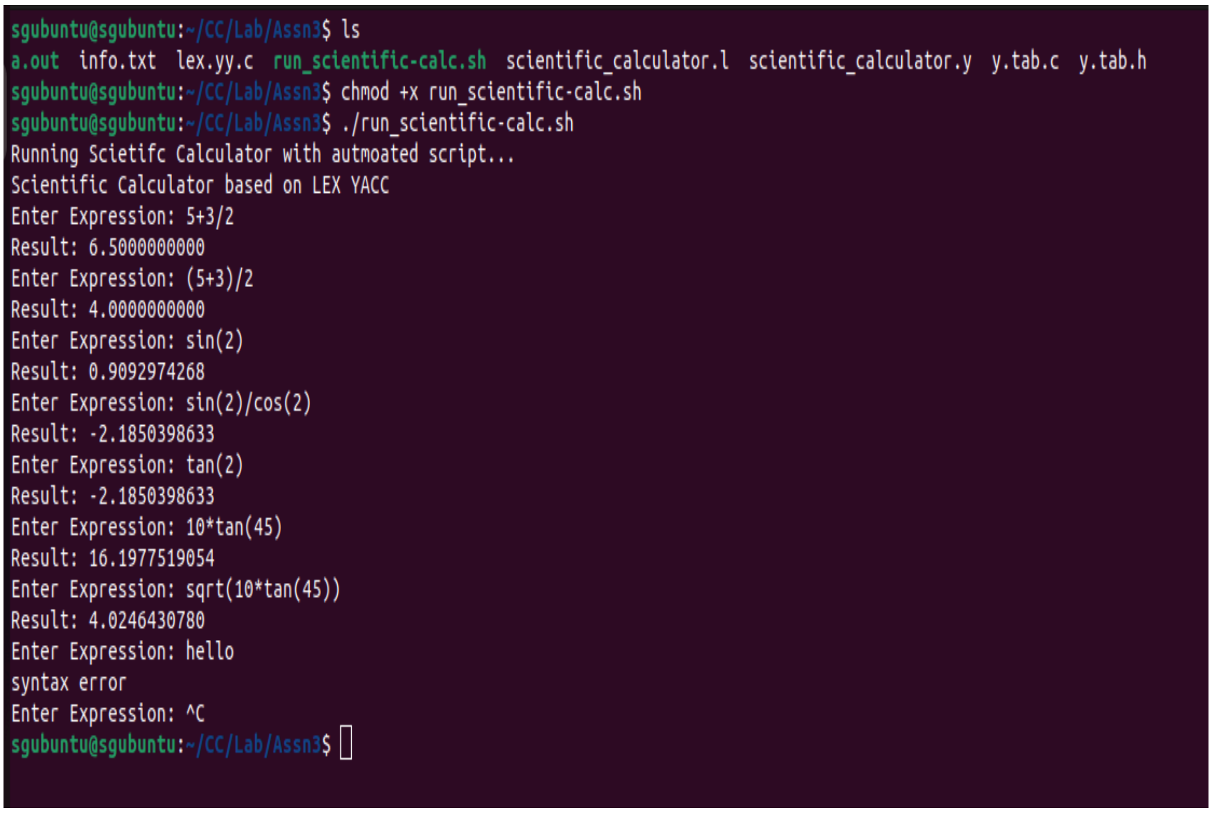
Explanation:

1. In this code , we are taking the input as the file name and returning the output as the number of lines and the number of words the file contains.
2. We have handled the error cases that may revolve in this code which is of the wrong file name given in shell , if the user inputs wrong file name then a prompt occurs indicating wrong file name and the code exits.
3. The code works correctly when given the correct file name that exists in the same directory and then it counts the words by looking for the whitespaces in between and it counts the line by taking the newline into consideration.
4. It counts the words and lines by making a buffer and then initialising the file pointer and pointing it to the buffer items and iterating through it.

2) Design a scientific calculator using Lex & Yacc or PLY or ANTLR tools.

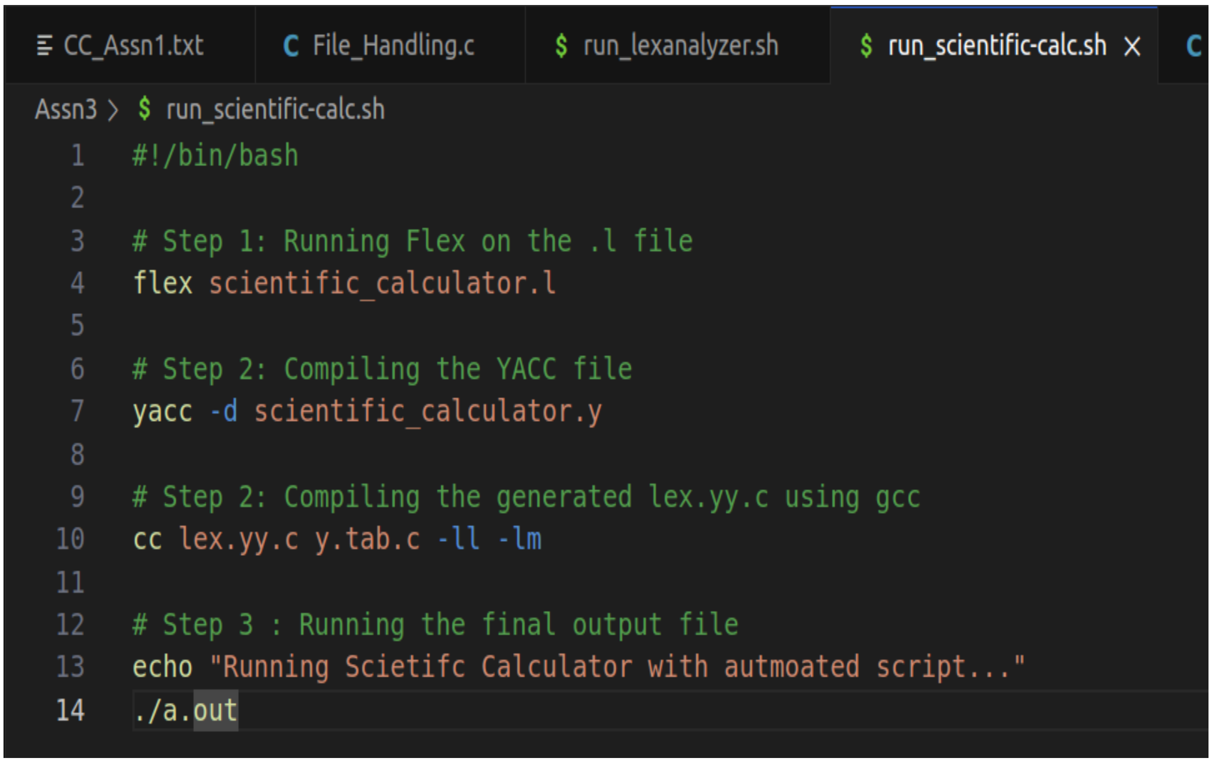
->

Input and Output:



```
sgubuntu@sgubuntu:~/CC/Lab/Assn3$ ls
a.out  info.txt  lex.yy.c  run_scientific-calc.sh  scientific_calculator.l  scientific_calculator.y  y.tab.c  y.tab.h
sgubuntu@sgubuntu:~/CC/Lab/Assn3$ chmod +x run_scientific-calc.sh
sgubuntu@sgubuntu:~/CC/Lab/Assn3$ ./run_scientific-calc.sh
Running Scietifc Calculator with autmoated script...
Scientific Calculator based on LEX YACC
Enter Expression: 5+3/2
Result: 6.5000000000
Enter Expression: (5+3)/2
Result: 4.0000000000
Enter Expression: sin(2)
Result: 0.9092974268
Enter Expression: sin(2)/cos(2)
Result: -2.1850398633
Enter Expression: tan(2)
Result: -2.1850398633
Enter Expression: 10*tan(45)
Result: 16.1977519054
Enter Expression: sqrt(10*tan(45))
Result: 4.0246430780
Enter Expression: hello
syntax error
Enter Expression: ^C
sgubuntu@sgubuntu:~/CC/Lab/Assn3$
```

run_scientific-calc.sh:



```bash
#!/bin/bash

# Step 1: Running Flex on the .l file
flex scientific_calculator.l

# Step 2: Compiling the YACC file
yacc -d scientific_calculator.y

# Step 2: Compiling the generated lex.yy.c using gcc
cc lex.yy.c y.tab.c -ll -lm

# Step 3 : Running the final output file
echo "Running Scietifc Calculator with autmoated script..."
./a.out
```

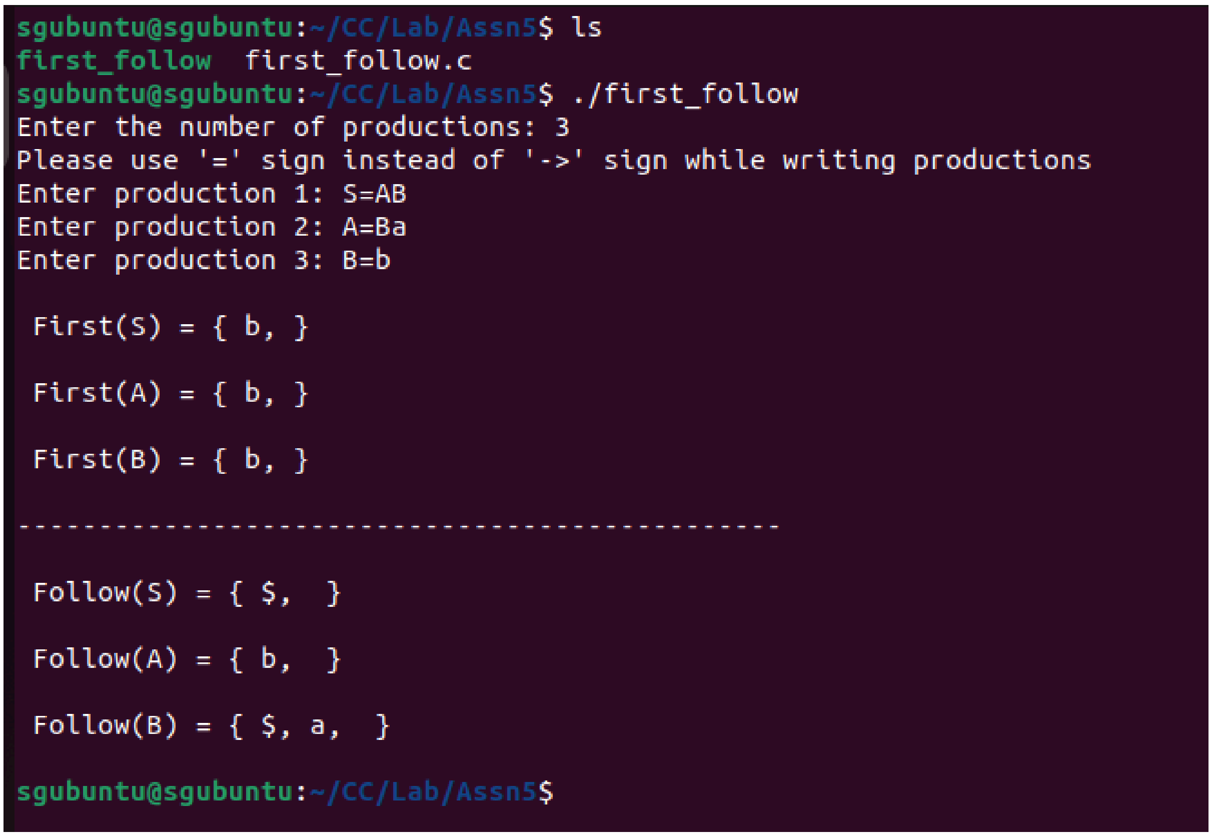
Explanation:

1. This is code for a scientific calculator program that accepts simple arithmetic expressions and scientific functions such as log, sin, cos, tan and square root. The calculator program reads input from the user, tokenizes the input using Lex, parses the tokens using Yacc, and evaluates the expression to produce the result.
2. It allows the user to enter expressions containing various mathematical functions and operators, including addition, subtraction, multiplication, division, sin, cos, tan, log, sqrt, and power.
3. The program evaluates the expression and prints the result.
4. The Lex portion of the program defines the lexical analyzer, which is responsible for recognizing tokens in the user's input.
5. Tokens represent the different types of values that can be recognized by the lexer. Tokens can be literals (like ADD), or they can be variables (like num).
6. It also takes care of precedence which determines the order in which the operations are performed.
7. Each rule in the Lex program matches a particular pattern in the user's input and returns the corresponding token to Yacc. For example, the first rule matches a sequence of one or more digits followed by an optional decimal point and one or more digits, and returns the num token with the corresponding value. The other rules match individual operators and special functions and return the corresponding token.
8. It also ignores whitespace and newlines.
9. Overall, the Lex program defines the tokens that will be used by the Yacc parser, and matches them to corresponding patterns in the user's input.
10. The Yacc takes the tokens as input from the lexer and then parses it and puts it together to evaluate the expression.
11. The program runs in a loop so that multiple expressions can be evaluated by the user and the program ends when Ctrl+C is pressed which is also the abort flag for the program.
12. The program also handles the case when the user inputs something wrong that is other than mathematical expressions.
13. For example when a user enters hello which is not a mathematical expression it will print syntax error and the loop continues until the abort flag is called on the terminal.

3)Write a code for finding the FIRST & FOLLOW of a grammar.

->

Input and Output:



```
sgubuntu@sgubuntu:~/CC/Lab/Assn5$ ls
first_follow  first_follow.c
sgubuntu@sgubuntu:~/CC/Lab/Assn5$ ./first_follow
Enter the number of productions: 3
Please use '=' sign instead of '->' sign while writing productions
Enter production 1: S=AB
Enter production 2: A=Ba
Enter production 3: B=b

 First(S) = { b, }

 First(A) = { b, }

 First(B) = { b, }

-----------------------------------------------

 Follow(S) = { $,  }

 Follow(A) = { b,  }

 Follow(B) = { $, a,  }

sgubuntu@sgubuntu:~/CC/Lab/Assn5$
```

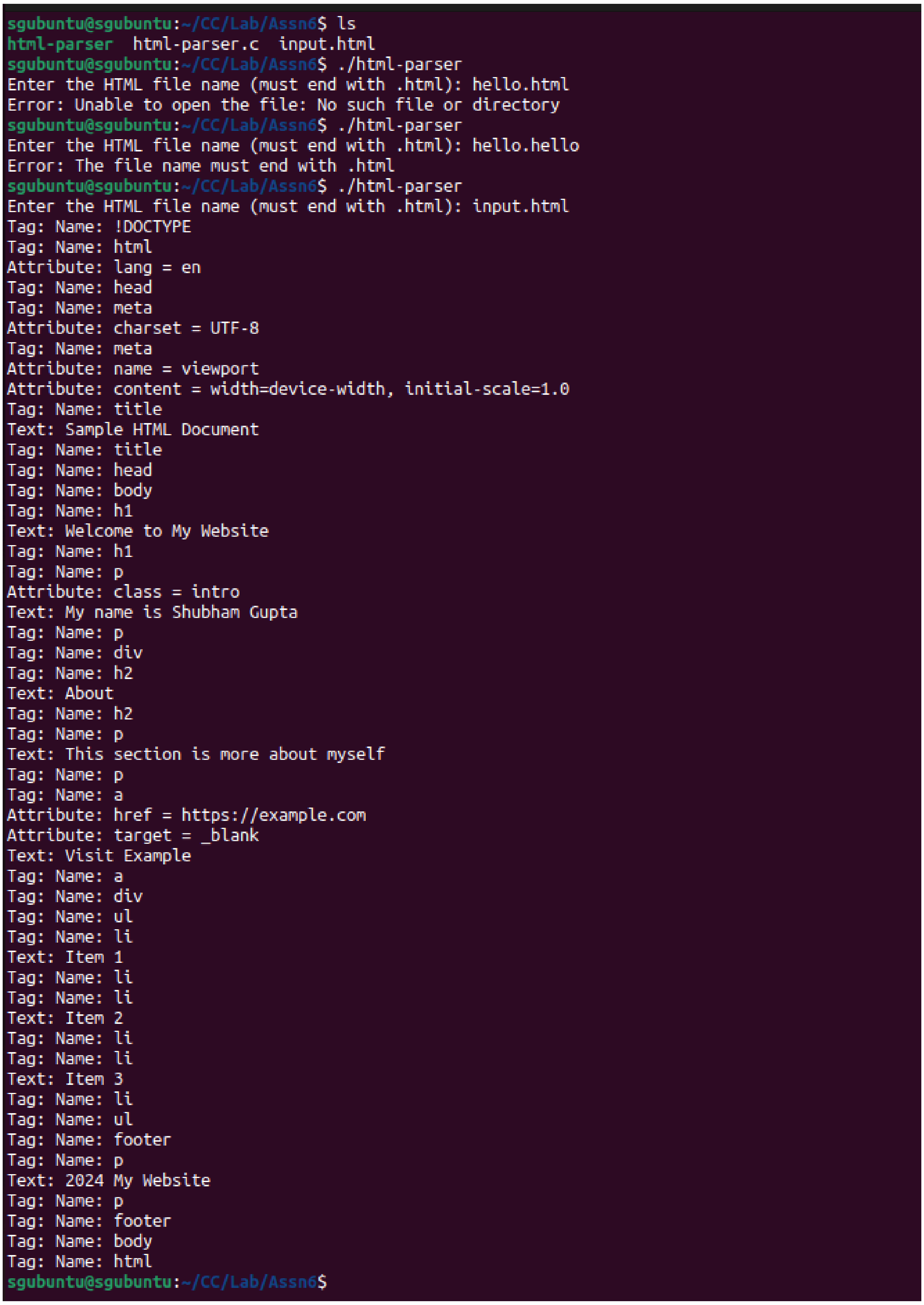
Explanation:

1. In the given code the user inputs the number of productions and then the user also enters the production rules and after the user enters correct rules the code calculates First and Follow of the variables.
2. First indicates which terminal can start production.
3. For each non-terminal, the first set is calculated by recursively checking the production rules, adding the first terminal or epsilon encountered.
4. Follow indicates what terminal can follow a non-terminal.
5. The follow set is determined by finding terminals or non-terminals that can appear immediately after a given non-terminal in the grammar and recursively calculating follow sets for non-terminals on the right-hand side of the production.
6. The given code only executes when the grammar is context free that is there is no left recursion and ambiguity in the grammar and by any chance if the grammar is not context free then it throws an error statement that the Segmentation is core dumped that is the code does not support this version of grammar.

4)Design a SQL parser / html parser.

->

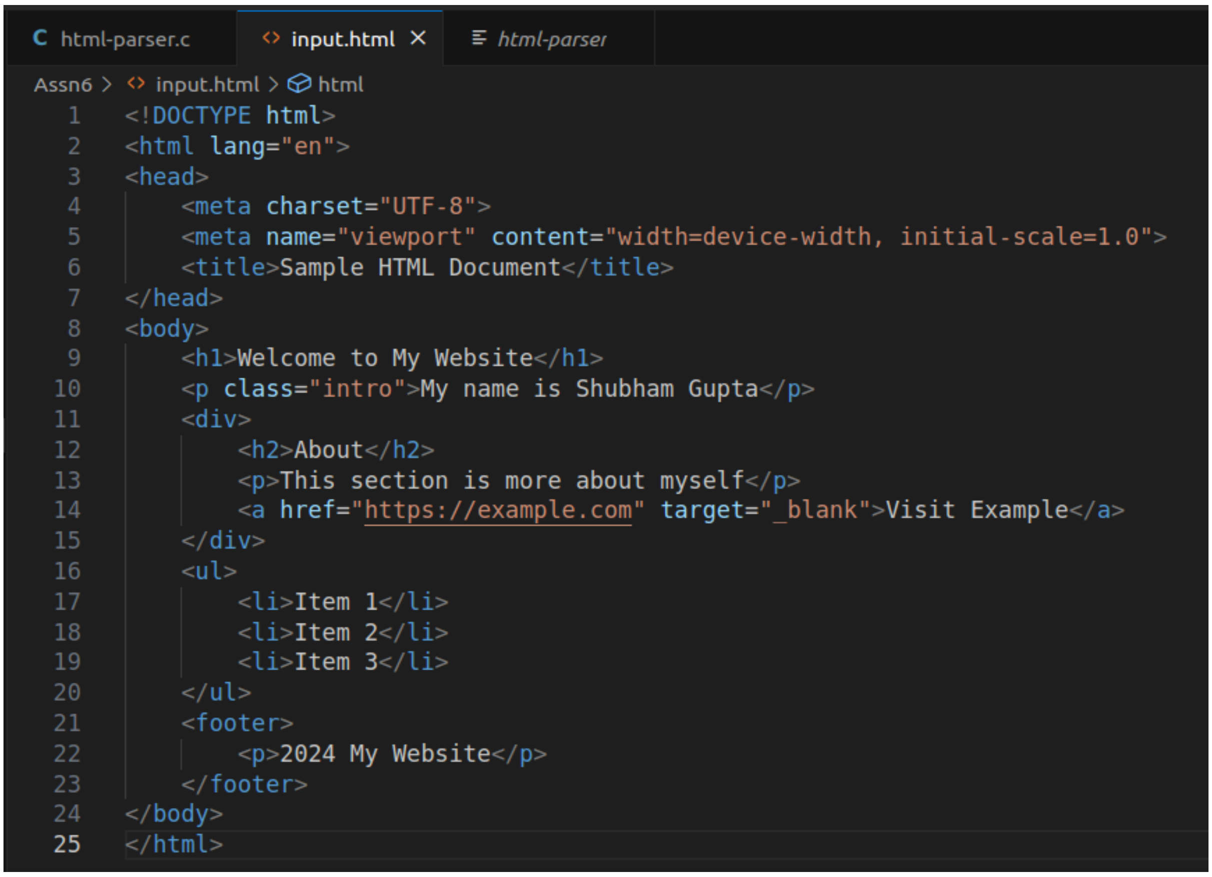
Input and Output:



```
sgubuntu@sgubuntu:~/CC/Lab/Assn6$ ls
html-parser  html-parser.c  input.html
sgubuntu@sgubuntu:~/CC/Lab/Assn6$ ./html-parser
Enter the HTML file name (must end with .html): hello.html
Error: Unable to open the file: No such file or directory
sgubuntu@sgubuntu:~/CC/Lab/Assn6$ ./html-parser
Enter the HTML file name (must end with .html): hello.hello
Error: The file name must end with .html
sgubuntu@sgubuntu:~/CC/Lab/Assn6$ ./html-parser
Enter the HTML file name (must end with .html): input.html
Tag: Name: !DOCTYPE
Tag: Name: html
Attribute: lang = en
Tag: Name: head
Tag: Name: meta
Attribute: charset = UTF-8
Tag: Name: meta
Attribute: name = viewport
Attribute: content = width=device-width, initial-scale=1.0
Tag: Name: title
Text: Sample HTML Document
Tag: Name: title
Tag: Name: head
Tag: Name: body
Tag: Name: h1
Text: Welcome to My Website
Tag: Name: h1
Tag: Name: p
Attribute: class = intro
Text: My name is Shubham Gupta
Tag: Name: p
Tag: Name: div
Tag: Name: h2
Text: About
Tag: Name: h2
Tag: Name: p
Text: This section is more about myself
Tag: Name: p
Tag: Name: a
Attribute: href = https://example.com
Attribute: target = _blank
Text: Visit Example
Tag: Name: a
Tag: Name: div
Tag: Name: ul
Tag: Name: li
Text: Item 1
Tag: Name: li
Tag: Name: li
Text: Item 2
Tag: Name: li
Tag: Name: li
Text: Item 3
Tag: Name: li
Tag: Name: ul
Tag: Name: footer
Tag: Name: p
Text: 2024 My Website
Tag: Name: p
Tag: Name: footer
Tag: Name: body
Tag: Name: html
sgubuntu@sgubuntu:~/CC/Lab/Assn6$
```

112103046 - CC Assignment

input.html:



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sample HTML Document</title>
</head>
<body>
    <h1>Welcome to My Website</h1>
    <p class="intro">My name is Shubham Gupta</p>
    <div>
        <h2>About</h2>
        <p>This section is more about myself</p>
        <a href="https://example.com" target="_blank">Visit Example</a>
    </div>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
    <footer>
        <p>2024 My Website</p>
    </footer>
</body>
</html>
```

Explanation:

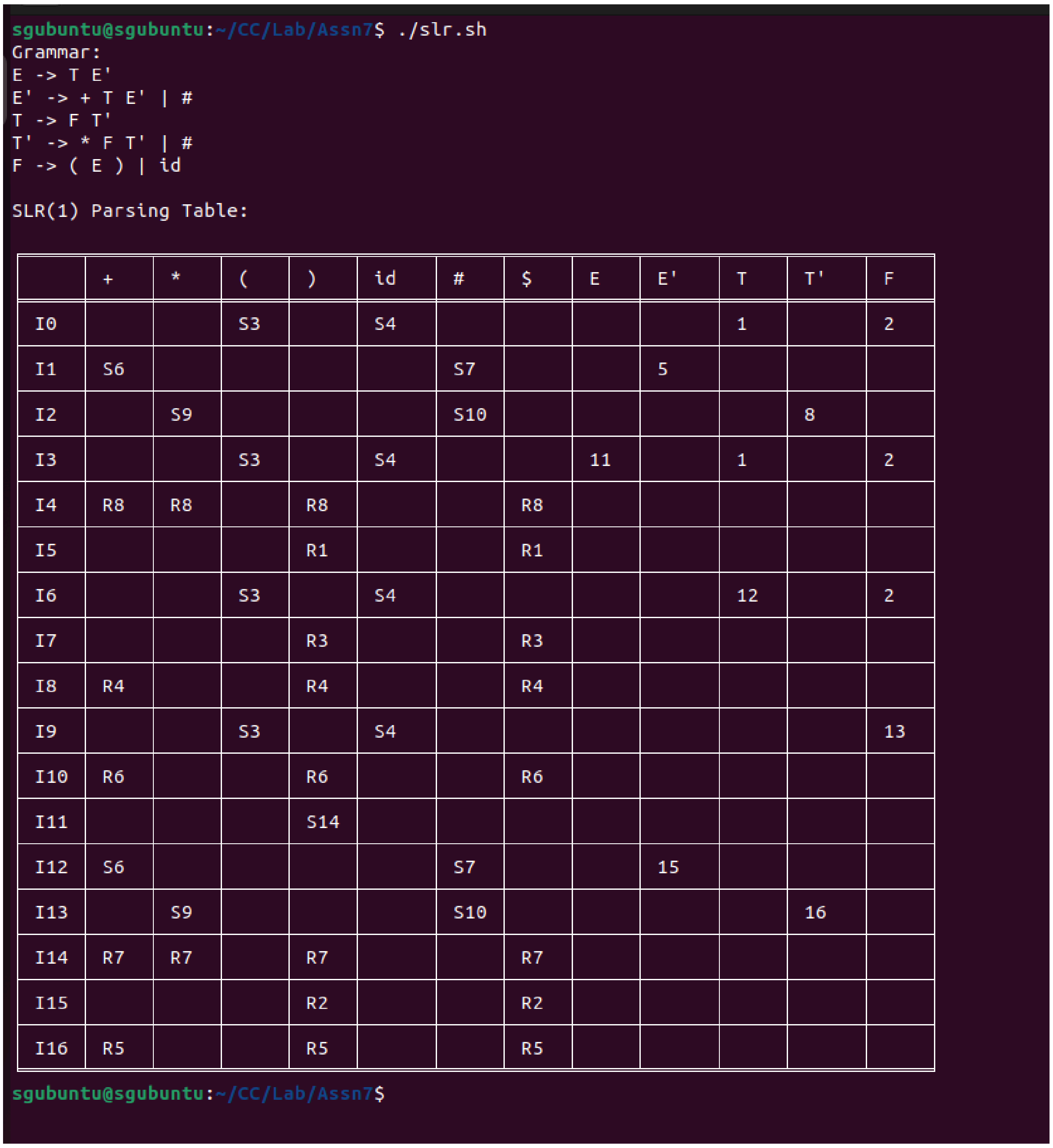
1. An HTML parser is a program designed to read HTML documents and extract structured data from them, enabling the analysis and manipulation of HTML content, such as tags, attributes, and text.
2. The provided code implements a simple HTML parser that reads an HTML file, identifies HTML tags, extracts their attributes, and captures the text content between the tags.
3. It goes through the flow of input handling,file reading,tag & text parsing,tag capture,text capture and memory management.
4. Input Handling: It prompts the user for an HTML filename, verifies the extension, and checks if the file exists.
5. In the input handling all the use cases are taken care like if the user inputs wrong filename then an error message pops that the file is not in the directory or if the filename contains some other extension rather than html then it prompts that the filename must end with html.
6. File Reading: Reads the content of the specified HTML file into a dynamically allocated buffer which is taken care of by allocating memory.
7. Tag and Text Parsing: Iterates through the HTML content, identifying when it is within tags and when it is capturing text outside of tags.

8. Tag Parsing: When a complete tag is detected, it calls parseTag to print the tag name and its attributes.
9. Text Capture: Prints any captured text only if it contains non-whitespace characters ensuring no extra text is printed on the screen.

5) Implement a SLR parser for a given grammar

->

Input and Output:



```
sgubuntu@sgubuntu:~/CC/Lab/Assn7$ ./slr.sh
Grammar:
E -> T E'
E' -> + T E' | #
T -> F T'
T' -> * F T' | #
F -> ( E ) | id

SLR(1) Parsing Table:

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|     | +   | *   | (   | )   | id  | #   | $   | E   | E'  | T   | T'  | F   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I0  |     |     | S3  |     | S4  |     |     |     |     | 1   |     | 2   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I1  | S6  |     |     |     |     | S7  |     |     | 5   |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I2  |     | S9  |     |     |     | S10 |     |     |     |     | 8   |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I3  |     |     | S3  |     | S4  |     |     | 11  |     | 1   |     | 2   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I4  | R8  | R8  |     | R8  |     |     | R8  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I5  |     |     |     | R1  |     |     | R1  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I6  |     |     | S3  |     | S4  |     |     |     |     | 12  |     | 2   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I7  |     |     |     | R3  |     |     | R3  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I8  | R4  |     |     | R4  |     |     | R4  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I9  |     |     | S3  |     | S4  |     |     |     |     |     |     | 13  |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I10 | R6  |     |     | R6  |     |     | R6  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I11 |     |     |     | S14 |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I12 | S6  |     |     |     |     | S7  |     |     | 15  |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I13 |     | S9  |     |     |     | S10 |     |     |     |     | 16  |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I14 | R7  | R7  |     | R7  |     |     | R7  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I15 |     |     |     | R2  |     |     | R2  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I16 | R5  |     |     | R5  |     |     | R5  |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
sgubuntu@sgubuntu:~/CC/Lab/Assn7$
```

Explanation:

112103046 - CC Assignment

1. SLR (Simple LR) is a type of bottom-up parsing technique that uses an LR(0) automaton and the Follow sets of non-terminals to decide reductions.
2. It's a simplified form of LR parsing, where decisions for shift or reduce actions are made based on the current state and lookahead symbol.
3. Now,in the code we have given an example grammar which can be seen from the grammar in the terminal and the output is the SLR Parsing Table.
4. This code follows this basic algorithm: first, it augments the grammar, generates states with closures and GOTO functions, and finally constructs the SLR table using shift and reduce actions based on the states and Follow sets.
5. Augment Grammar: We start the code by augmenting the grammar that is adding a new start symbol and a new rule to include it.
6. Create LR(0) Automaton:Generate the initial closure set of states, starting from the augmented grammar.
7. For this purpose we also use the GOTO function to generate new states by shifting the dot (.) over symbols.
8. We continue and iterate until all states are created.
9. Compute First and Follow Sets:Compute the First set for each non-terminal to determine what symbols can appear first in its production and also compute the Follow set for each non-terminal to identify the symbols that can follow it in a derivation.
10. SLR Table Construction:For shift actions,we find states where the dot (.) is not at the end, and compute transitions using GOTO for terminals.
11. For the reduce actions,we check if the dot (.) is at the end of a rule and we look at the Follow set of the rule's non-terminal.
12. We handle the start symbol with an Accept action when it is appropriate.
13. If it is not appropriate it throws an error of Segmentation Fault:core dumped and the process stops.
14. At the end when the action is appropriate it creates the final SLR parsing table.