**FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)**

****

**COMP 451 (Compiler Construction)**

**2022 FALL**

**Project Report**

**Gulraiz Noor Bari (231-525536)**

**TASK – 2**

**LL(1) PARSER**

LL(1) is a Top-Down Parser without backtracking/Non-recursive descent parser. The problem statement is that we have to design and code a LL(1) Parser for the following Grammar:

**S 🡪 A a**

**A 🡪 B D**

**B 🡪 b | ε**

**D 🡪 d | ε**

For that we will follow the following steps:

* Find the FIRST and FOLLOW of the grammar.
* List down the Parse Table using FIRST and FOLLOW.
* Generate the Stack Implementation Table using Parse Table, to check whether the input string is valid for this grammar or not.

**FIRST & FOLLOW:**

|  |  |  |
| --- | --- | --- |
| NT | FIRST | FOLLOW |
| S | a, b, d | $ |
| A | b, d, **ε** | A |
| B | b, **ε** | d, $ |
| D | d, **ε** | a |

**PARSE TABLE:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | d | $ |
| S | S 🡪 A a | S 🡪 A a | S 🡪 A a |  |
| A | A 🡪 B D | A 🡪 BD | A 🡪 BD |  |
| B |  | B 🡪 b | B 🡪  ε | B 🡪  ε |
| D | D 🡪  ε |  | D 🡪  d |  |

**PROGRAM FOR STACK IMPLEMENTATION TABLE:**

Stack implementation table is used to check validity of a string for a grammar. The Algorithm along with the code has been explained in this section for generating the Stack Implementation Table to validate strings for our grammar.

**INTRODUCTION:**

* **stdio.h** (standard input/output)is a header file that contains declarations for functions like, printf, scanf, etc.
* **stdlib.h** (standard library) is a header file that contains declarations of functions that involves memory allocation and process control. For example, in our program we used **exit(0)** at the end; which means to successfully terminate the program.
* **string.h** is a header file that contains declarations of functions that are used for working with strings. In our program we have used **strlen** function to get the length of string.
* **ctype.h** is a header file that contains functions that are used to handle characters. In our program we have used **isalpha** function to check whether the character is alphabet or not.
* **main():** The main function serves as the starting point of the program execution in C language. User can pass any number of parameters depending upon the requirements of the program logic or structure.

**LOGIC/ALGORITHM:**

The code is a program that implements Stack Implementation Table for the following grammar:

**S 🡪 A a**

**A 🡪 B D**

**B 🡪 b | ε**

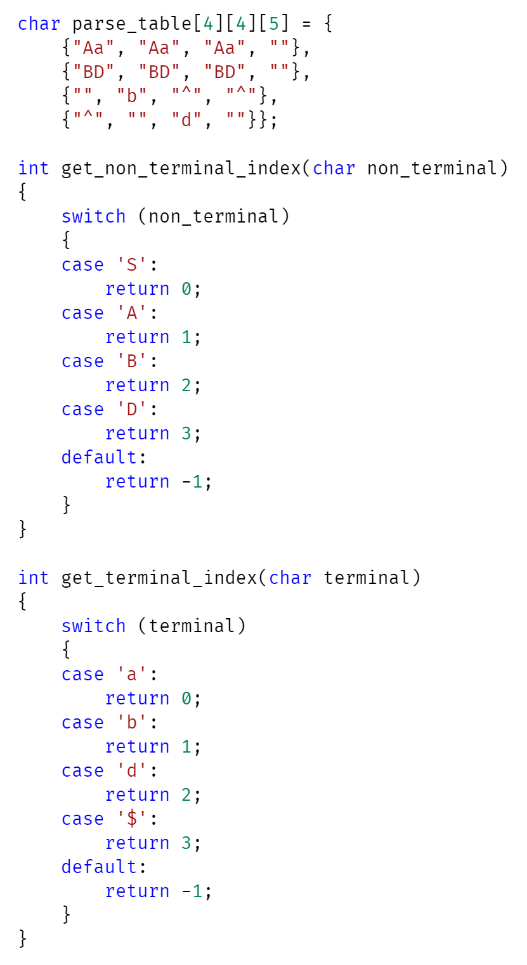
**D 🡪 d | ε**

This Stack Implementation Table is generated using the Parse Table.

We initialize three lists that will be implemented to behave as stacks throughout the program:

* + Stack1: processing stack
  + Stack2: input stack

and initialize two pointers and each of them will point to the top of each stack.

* Then we write functions for the implementation of stack features:
  + Push: pushing elements on the top of stack
  + Pop: removing elements from the top of stack
  + Peak: check the element on the top of stack
* Two functions are also initialized:
  + **Get\_Terminal\_Index()**: takes a terminal symbol as parameter and returns its specified index; using switch statements.
  + **Get\_NonTerminal\_Index()**: takes a non-terminal symbol as parameter and returns its specified index; using switch statements.
* The program starts by checking whether two arguments were passed by the user in command line, which is done by argc (which has the count of arguments entered in the command-line). If the number of arguments is not two the function is not proceeded and displays a message “Invalid argument”, otherwise if user has entered two arguments; in our case the executable and the string to be parsed, the program proceeds.
* It stores the strring entered by the user in a character array “input” and checks if the last character of the input string is a **“$”** (sentinel symbol); if not then it terminates the program and throws error.
* If previous condition satisfies then it runs a reverse for-loop on the input string which is stored as a array of characters in C. At every iteration it pushes the current character in stack-2, at the end of the loop the last element of the array (in our case “b”) will be at the top of stack.
* Graphical user interface, text, application, email

  Description automatically generatedNow, we run another loop on the input string but this time it runs till before last element (i < array.length – 1) and we check if the input string characters entered by the user are all alphabets.
* Till now we did some user input preprocessing part, but now we will start the actual Stack Table Generation and for that we will first push **“$”** and **“S”** on stack1.
* As the code will run in a constant loop till the terminating condition has been met, so for that purpose we will initiate a while loop and it will execute till both the stacks (stack1 & stack2) are left with sentinel/terminating symbol, which is **“$”.**
* Inside this while loop we will perform a if-else check for the following conditions:
  + First it checks if the current element at the top of processing stack (stack1) is a terminal or non-terminal, by using **isUpper()**, in case of True it means that it is a non-terminal symbol. Now it opens the identified non-terminal symbol by performing the following checks:
    - First we identify the non-terminal index using the non-terminal symbol which is always at the top of processing stack (stack1) and we pass it to the user-defined function get\_nonterminal\_index(). If the function returns a value of -1 then it means that the symbol is invalid, and it means that the string cannot be processed by our grammar; hence the program breaks out of the loop.
    - First we identify the terminal index using the terminal symbol which is always at the top of input stack (stack2) and we pass it to the user-defined function get\_terminal\_index(). If the function returns a value of -1 then it means that the symbol is invalid, and it means that the string cannot be processed by our grammar; hence the program breaks out of the loop
    - After successful execution of the above conditions we have the terminal and non-terminal indices so we can search for the production in our parse table.
    - If there is no production for our current terminal and non-terminal symbol then it means that the string cannot be parsed by our grammar/parser.
    - Otherwise, we print the current status of both our stacks and pop the topmost non-terminal element from stack1 (processing stack) and check if the production is not equal to epsilon, if not then we print the production picked from the parse table to processing stack (stack1) of the Stack Implementation Table.

Text

Description automatically generated with medium confidence

* + If the **isUpper()** is False, then it means that we have a terminal symbol (represented by lower case) and now we check if the terminal symbol currently being read (top of stack1) is equal to the top of input stack (stack2); if True then we pop elements from both the stacks and print the current status of both the stacks.
  + If the both the terminals are not equal, then we can say that the string is not a valid string for the grammar; and we break out of the while loop.
* After the while-loop is over we apply the final check to display the result that whether the string is accepted or rejected by our grammar. We do this by checking if the stack top pointers point to the **“$”**; as “$” will always be at the bottom of the stack, if True than String is Accepted otherwise it is Rejected.

Graphical user interface, application

Description automatically generated with medium confidence

Table

Description automatically generated with low confidence

Text

Description automatically generated with low confidence

**OUTPUT:**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**TASK – 1**

**LR(0) PARSER**

LR(0) is a Bottom-Up Parser. The problem statement is that we have to design and code a LR(0) Parser for the following Grammar:

**E 🡪 B B**

**B 🡪 c B**

**B 🡪 d**

For that we will follow the following steps:

* Find the Augmented grammar.
* List down the Canonical Item sets.
* Assign Numbers to the production rules.
* List down the goto[list of non-terminals] and action[list of terminals] in the parsing table.
* Generate the Stack Implementation Table using Parse Table, to check whether the input string is valid for this grammar or not.

**AUGMENT THE GRAMMAR:**

E**’** 🡪 E

E 🡪 B B

B 🡪 c B

B 🡪 d

**NUMBER THE PRODUCTIONS:**

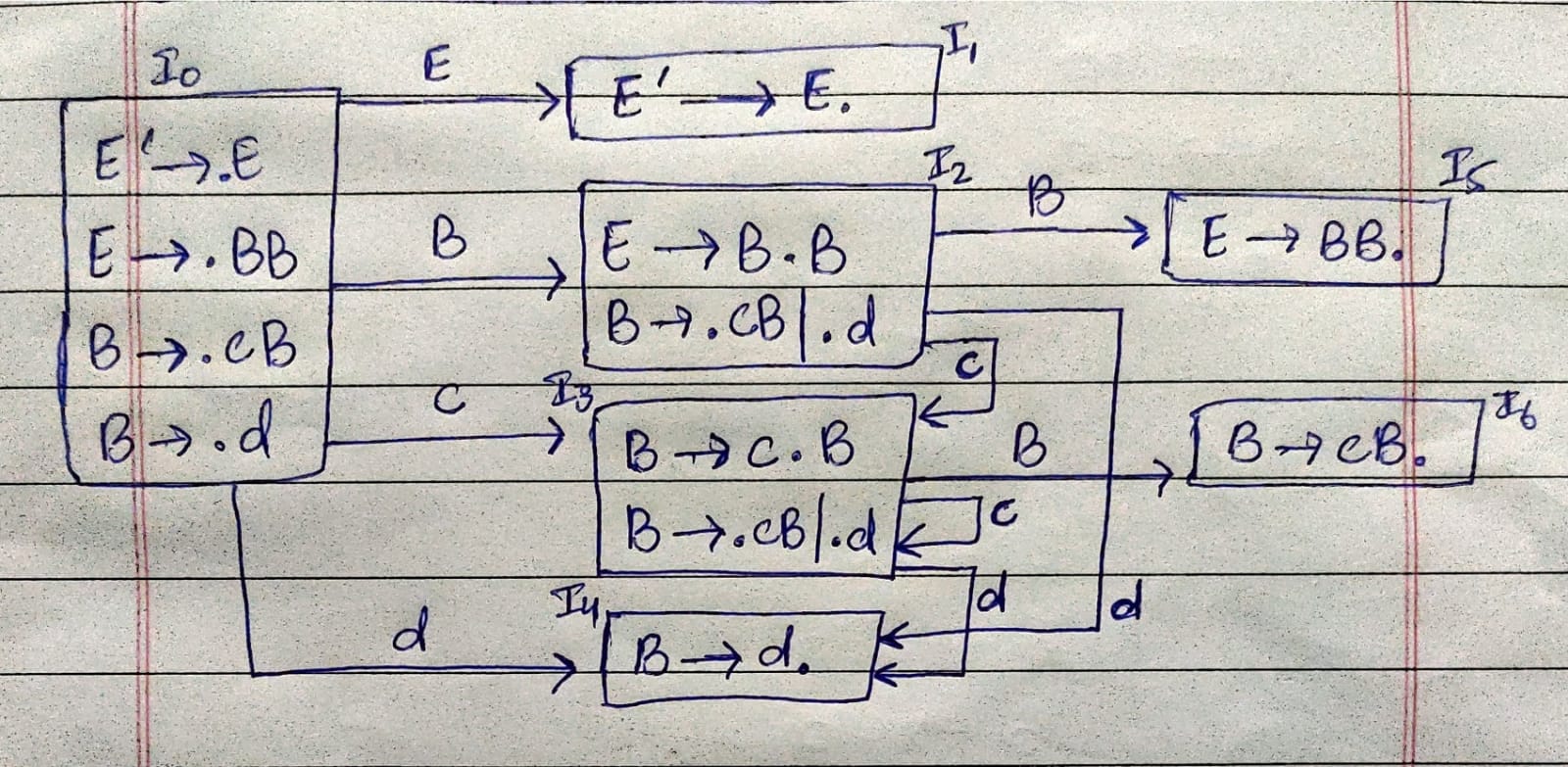
E**’** 🡪 E 🡪 **(1)**

E 🡪 B B 🡪 **(2)**

B 🡪 c B 🡪 **(3)**

B 🡪 d 🡪 **(4)**

**CANONICAL ITEM SETS:**



**PARSE TABLE:**

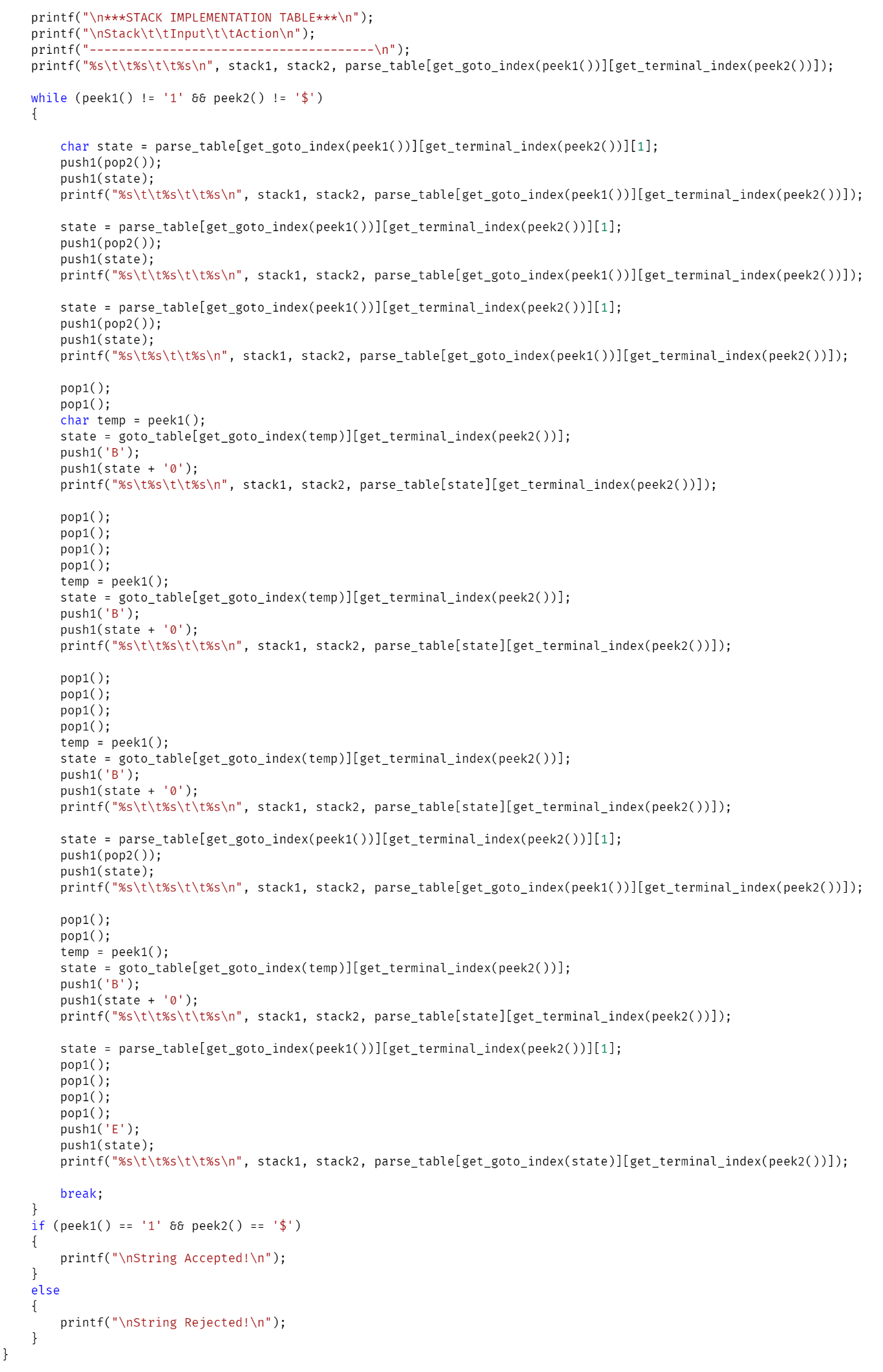
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| States | Action | | | Goto | |
|  | **c** | **d** | **$** | **E** | **B** |
| I0 | s3 | s4 |  | I1 | I2 |
| I1 |  |  | Accept |  |  |
| I2 | s3 | s4 |  |  | I5 |
| I3 | s3 | s4 |  |  | I6 |
| I4 | r3 | r3 | r3 |  |  |
| I5 | r1 | r1 | r1 |  |  |
| I6 | r2 | r2 | r2 |  |  |

Graphical user interface, application

Description automatically generated with medium confidence

Chart

Description automatically generated with medium confidence



**OUTPUT:**

