An Emulator for Predictive Parsing

PRACTICAL ASSIGNMENT REPORT
SUBMITTED TO

M S RAMAIAH INSTITUTE OF TECHNOLOGY
(Autonomous Institute, Affiliated to VTU)
Bangalore – 560054

SUBMITTED BY

Aahan Singh 1MS13CS001
Abdul Rabbani Shah 1MS13CS004
Akhil Raj Azhikodan 1MS13CS017
Archit Bhatnagar 1MS13CS030

As part of the Course **Compiler Design - CS612**

SUPERVISED BY
Sini Anna Alex

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

M S RAMAIAH INSTITUTE OF TECHNOLOGY

Jan-May 2016

Department of Computer Science and Engineering

M S Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)

Bangalore – 54

CERTIFICATE

This is to certify that Aahan Singh **(1MS13CS001),** Abdul Rabbani Shah **(1MS13CS004),** Akhil Raj Azhikodan**(1MS13CS017),** Archit Bhatnagar **(1MS13CS030)** have completed the **"An Emulator for Predictive Parsing"** as part of practical assignment.

We declare that the entire content embodied in this B.E. 6th Semester report contents are not copied**.**

Submitted by

1. Aahan Singh
2. Abdul Rabbani Shah
3. Akhil Raj Azhikodan
4. Archit Bhatnagar

(Dept of CSE, MSRIT)

Guided by

Prof. Sini Anna Alex

(Assistant Professor, Dept. of CSE, MSRIT)

Department of Computer Science and Engineering

M S Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)

Bangalore – 54

Evaluation Sheet

Sl. No	USN Name	Content and Demonstration (6)	Speaking Skills (1)	Teamwork (1)	Neatness and care (1)	Effectiveness (2)	General Use of elements (2)	Productivity (2)	Total Marks (15)
1	1MS13CS001								
2	1MS13CS004								
3	1MS13CS017								
4	1MS13CS030								

Evaluated By

Name: Sini Anna Alex
Designation: Assistant Professor
Department: Computer Science & Engineering, MSRIT
Signature:

HOD, CSE

Peer Evaluation Guidelines for Compiler Design Practical Assignment

1. On the final day of practical assignment presentation, each group member should tick the point on the scale that describes the overall group's effectiveness.
2. The team's leader should then compile and average the numbered responses and use a highlighter to mark the point on the scoring line that indicates an average of the peer member's responses.
3. The teacher will then review the Rubric and add comment and adjust the score.

Excellent (5)	Very Good (4)	Good (3)	Fair (2)	Poor (1)
○	○	○	○	○

Remarks	

(Tick √ on any one point. In the Remarks justify your evaluation)

Course Coordinator

HOD, Dept. of CSE

Prof. Sini Anna Alex

Dr. K.G.Srinivasa

ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to our teacher Mrs. Sini Anna Alex who gave us the golden opportunity to do this wonderful project on the topic “An Emulator for Predictive Parsing “, which also helped us in doing a lot of research and we came to know about so many new things that we are really thankful for.

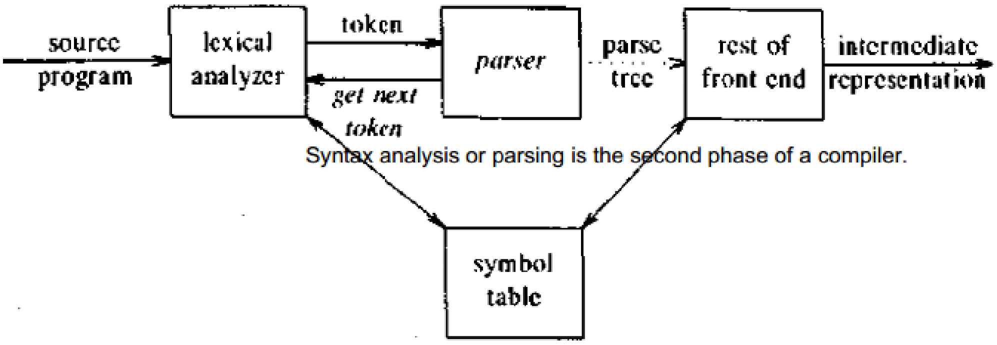
ABSTRACT

The purpose of the project is to develop an emulator for predictive parsing. The emulator would take in the grammar as the input and construct the parsing table. Then for any input string on the grammar, the emulator will validate it by parsing using the parsing table. We have made the front end in HTML and the backend with all algorithms in JavaScript. The input consists of a set of grammars which as passed to the JS file and then follows through with the splitting of input into proper strings for each grammar. After left factoring and removing left recursion the first and follow is calculated for each of the grammar strings. The parsing table is finally constructed at the end and printed to the webpage.

Contents

Introduction

The parser obtains a string of tokens from the lexical analyzer, as and verifies that the string can k generated by the grammar for the source language



Syntax analysis or parsing is the second phase of a compiler.

Predictive Parser:

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

Predictive Parsing:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$**to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

Literature Review

The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:

1) Eliminate Left Recursion
2) Perform Left Factoring

These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing .

Parsing an entire program is initiated by calling the parser function representing the grammar's start symbol.

The most important issue in implementing each function is how the production should be chosen, since there may be multiple productions on a single non-terminal symbol.

One possibility is to make the choice nondeterministic; the parser tries all possible choices of production. Any time the parser reaches a point where an error is raised (because the next input token didn't match the one expected), the parser "backtracks" to the most recent nondeterministic choice.

Nondeterminism and backtracking can be time-consuming, as the parser explores many dead ends in its attempt to find a leftmost derivation.

A better choice is to *look ahead* at the next tokens in the input program, and *predict* which production should be applied. If the input token or tokens immediately following the caret uniquely identify a production any time a non-terminal is expanded, then the non-determinism is unnecessary.

To implement a predictive parser, we can analyze the grammar to compute the FIRST and FOLLOW sets for each non-terminal symbol.

How to compute FIRST ?

➔ Look at the definition of FIRST(α) set:

- if α is a terminal, then FIRST(α) = { α }.
- if α is a non-terminal and α → Ɛ is a production, then FIRST(α) = { Ɛ }.
- if α is a non-terminal and α → γ1 γ2 γ3 … γn and any FIRST(γ) contains t then t is in FIRST(α).

How to compute FOLLOW ?

➔ ALGORITHM FOR CALCULATING FOLLOW SET:

- if α is a start symbol, then FOLLOW() = $
- if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except Ɛ.
- if α is a non-terminal and has a production α → AB, where B Ɛ, then FOLLOW(A) is in FOLLOW(α).

Algorithm for design:

Input:

string ω

parsing table M for grammar G

Output:

If ω is in L(G) then left-most derivation of ω,

error otherwise.

Initial State : $S on stack (with S being start symbol)

ω$ in the input buffer

SET ip to point the first symbol of ω$.

```
repeat
  let X be the top stack symbol and a the symbol pointed by ip.

  if X∈ Vt or $
    if X = a
      POP X and advance ip.
    else
      error()
    endif

  else   /* X is non-terminal */
    if M[X,a] = X → Y1, Y2,... Yk
      POP X
      PUSH Yk, Yk-1,... Y1 /* Y1 on top */
      Output the production X → Y1, Y2,... Yk
    else
      error()
    endif
  endif
until X = $      /* empty stack */
```

Relevance with respect to compiler phases:

The predictive parsing comes in the Syntax Analysis phase of the compiler design i.e. the parser checks if the input string is syntactically correct or not. In other words, the purpose of syntax analysis is to check if we have a valid sequence of tokens. The sequence need not be meaningful logically but has to be correct.

The output of this phase is a Parse Tree.

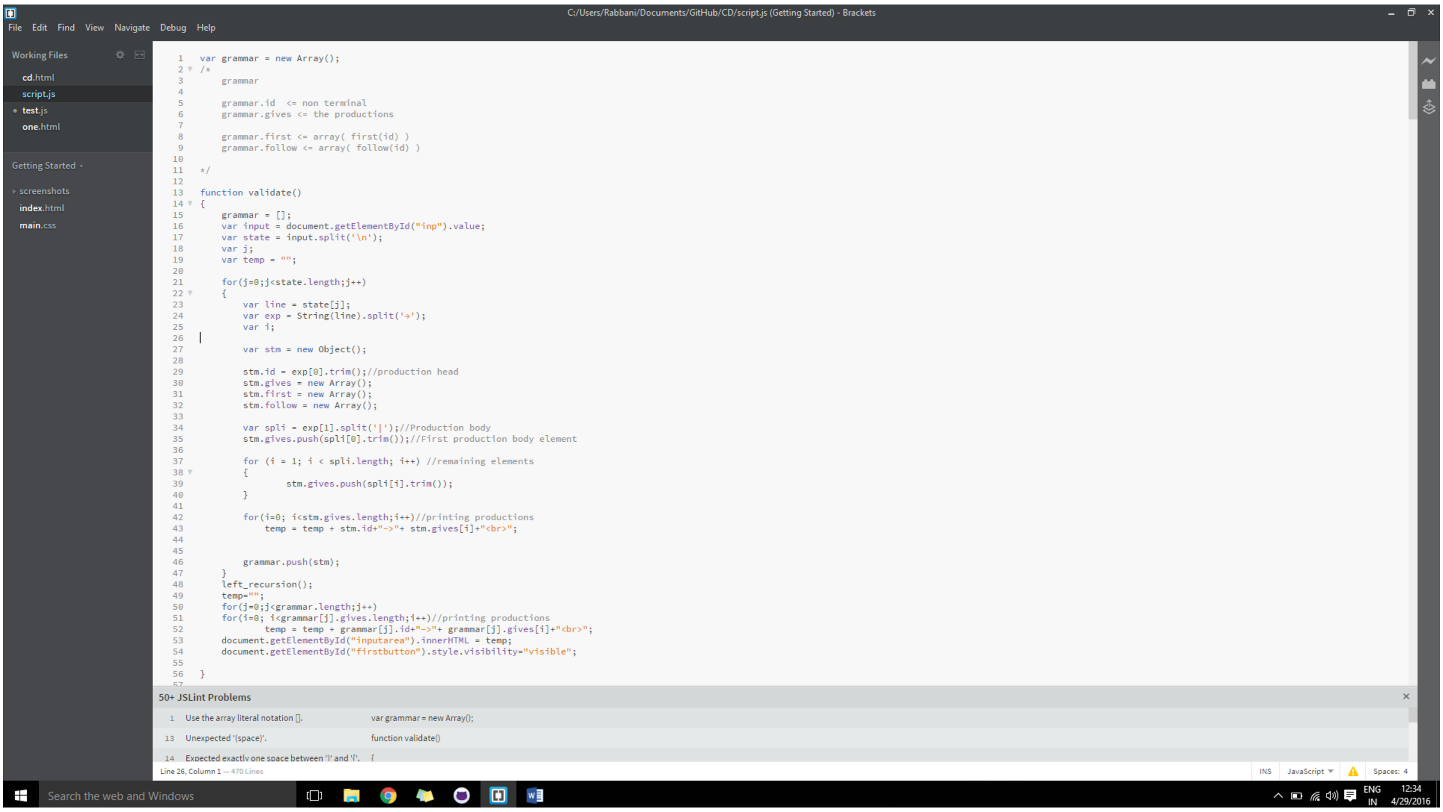
We need to eliminate direct and indirect left recursion and perform left factoring before we pass on the grammar to the Predictive parser. Note that the above two steps are not necessary for Bottom-Up parsing.

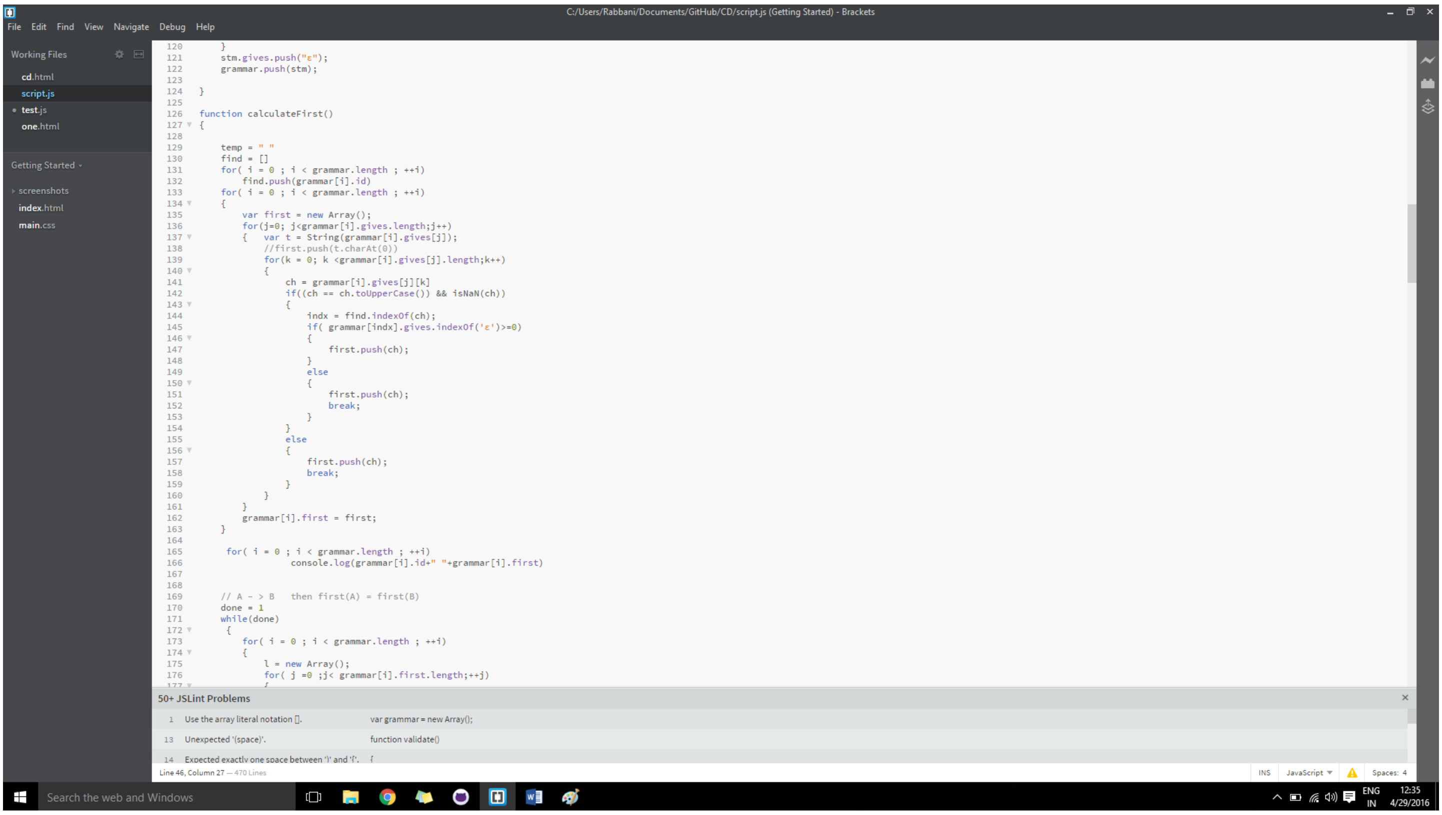
There are a few limitations of the Syntax Analysis phase as stated below.

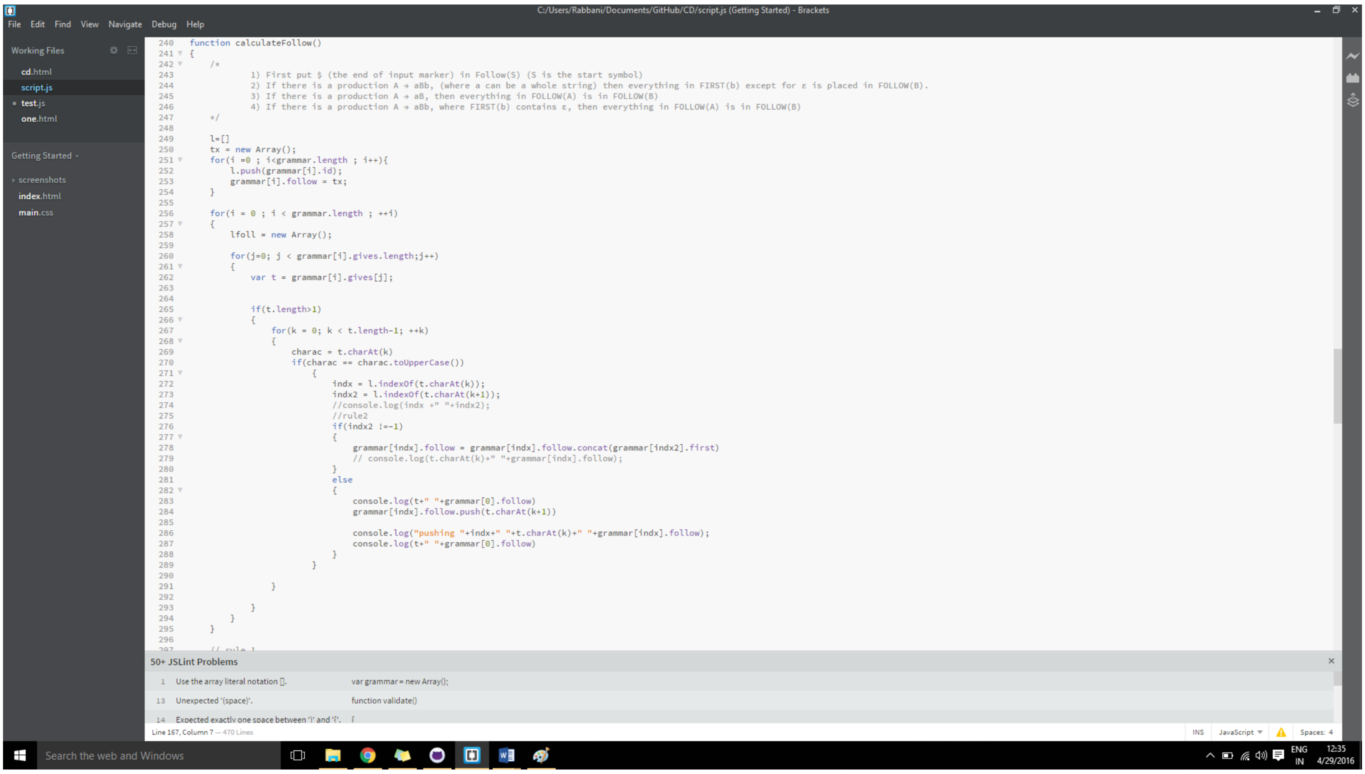
Lexical analyzer determines if there is any fault in the input token stream, however the syntax analyzer does not do that by itself.

Also, it does not determine if the tokens are initialized or declared prior to the parsing process. The syntax analyzer cannot determine the logical validity of the token as well.

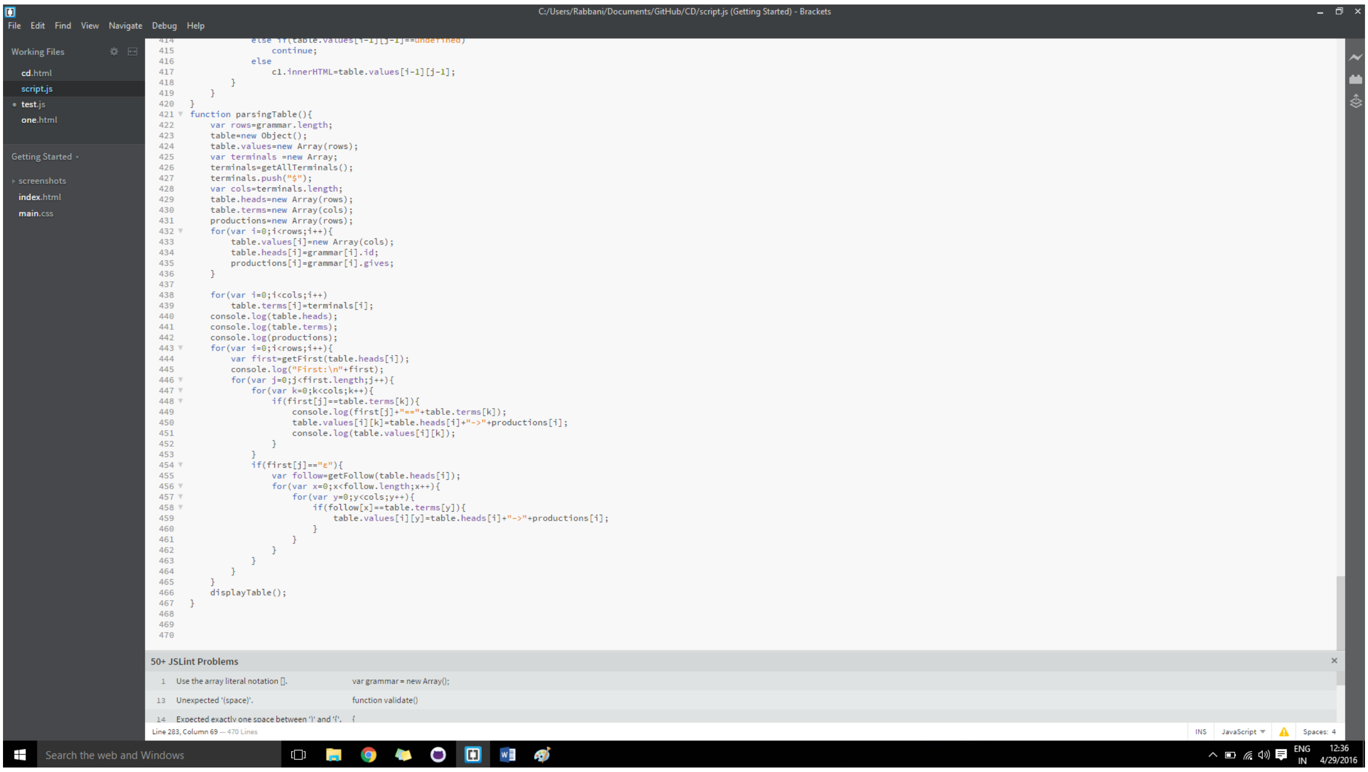
Screenshots:








```
function calculateFollow()
{
    /*
            1) First put $ (the end of input marker) in Follow(S) (S is the start symbol)
            2) If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).
            3) If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)
            4) If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)
    */

    l=[]
    tx = new Array();
    for(i =0 ; i<grammar.length ; i++){
        l.push(grammar[i].id);
        grammar[i].follow = tx;
    }

    for(i = 0 ; i < grammar.length ; ++i)
    {
        lfoll = new Array();

        for(j=0; j < grammar[i].gives.length;j++)
        {
            var t = grammar[i].gives[j];


            if(t.length>1)
            {
                for(k = 0; k < t.length-1; ++k)
                {
                    charac = t.charAt(k)
                    if(charac == charac.toUpperCase())
                        {
                            indx = l.indexOf(t.charAt(k));
                            indx2 = l.indexOf(t.charAt(k+1));
                            //console.log(indx +" "+indx2);
                            //rule2
                            if(indx2 !=-1)
                            {
                                grammar[indx].follow = grammar[indx].follow.concat(grammar[indx2].first)
                                // console.log(t.charAt(k)+" "+grammar[indx].follow);
                            }
                            else
                            {
                                console.log(t+" "+grammar[0].follow)
                                grammar[indx].follow.push(t.charAt(k+1))

                                console.log("pushing "+indx+" "+t.charAt(k)+" "+grammar[indx].follow);
                                console.log(t+" "+grammar[0].follow)
                            }
                        }

                }

            }
        }
    }
```




```
                    continue;
                else
                    c1.innerHTML=table.values[i-1][j-1];
            }
        }
}
function parsingTable(){
    var rows=grammar.length;
    table=new Object();
    table.values=new Array(rows);
    var terminals =new Array;
    terminals=getAllTerminals();
    terminals.push("$");
    var cols=terminals.length;
    table.heads=new Array(rows);
    table.terms=new Array(cols);
    productions=new Array(rows);
    for(var i=0;i<rows;i++){
        table.values[i]=new Array(cols);
        table.heads[i]=grammar[i].id;
        productions[i]=grammar[i].gives;
    }

    for(var i=0;i<cols;i++)
        table.terms[i]=terminals[i];
    console.log(table.heads);
    console.log(table.terms);
    console.log(productions);
    for(var i=0;i<rows;i++){
        var first=getFirst(table.heads[i]);
        console.log("First:\n"+first);
        for(var j=0;j<first.length;j++){
            for(var k=0;k<cols;k++){
                if(first[j]==table.terms[k]){
                    console.log(first[j]+"=="+table.terms[k]);
                    table.values[i][k]=table.heads[i]+"->"+productions[i];
                    console.log(table.values[i][k]);
                }
            }
            if(first[j]=="ε"){
                var follow=getFollow(table.heads[i]);
                for(var x=0;x<follow.length;x++){
                    for(var y=0;y<cols;y++){
                        if(follow[x]==table.terms[y]){
                            table.values[i][y]=table.heads[i]+"->"+productions[i];
                        }
                    }
                }
            }
        }
    }
    displayTable();
}
```

COMPILER DESIGN

List Of Grammers

1.	S → 0 S 1 | 0 1
2.	S → S (S) S | ε
3.	S → A A → BC | DBC B → Bb | ε C → c | ε D → a | d
4.	S → CC C → cC | d
5.	E → E + T | T T → T * F | F F → (E) | id
6.	S → AaAb | BbBa A → ε B → ε
7.	S → 0 | A S → AB B → 1
8.	S → S + S | S * S | a
9.	S → (L) | a L → L,S | S
10.	S → ABe A → dB | aS | c B → AS | b

S → ABe
A → dB | aS | c
B → AS | b

Validate | Calculate First | Calculate Follow

Input

S->ABe
A->dB
A->aS
A->c
B->AS
B->b

First

S - d a c
A - d a c
B - d a c b

Follow

S - e $
A - d a c b d a c
B - e $

Table	e	d	a	c	b	$
S		S->ABe	S->ABe	S->ABe		
A		A->dB	A->aS	A->c		
B		B->AS	B->AS	B->AS	B->b	

Conclusion:

We have read and understood the concepts of Syntax Analysis and in detail the predictive parser(Top Down) and have acquired the knowledge required for designing and making a program that parses an input grammar. We have then deployed this algorithm to create a working example of an emulator and simulates the predictive parsing taking into inputs as the grammar strings and producing the parsing table after calculating the first and follow.
We have used javascript to code the back end and displayed it on a web-page using HTML.

References:

Compilers : Principles, Techniques and Tool by Alfred Aho , Monica Lam , Ravi Sethi ,Jeffery D.

www.tutorialspoint.com - Compiler design

www.umd.edu - CS department.

www.mit.edu/cse - CS department Compiler design course