INDEX:

ABSTRACT

The objective of this project is to implement an efficient and robust CLR parser using a high-level programming language, that is capable of correctly parsing any input fed to it. The LR(1) parsing is a technique of bottom-up parsing. 'L' says that the input string is scanned from left to right, 'R' says that the parsing technique uses rightmost derivations, and '1' stands for the look-ahead. To avoid some of the invalid reductions, the states need to carry more information. Extra information is put into the state by adding a terminal symbol as the second component in the item.

Thus, the canonical-LR parser makes full use of look-ahead symbols. This method uses a large set of items, called LR(1) items.

The LR(1) parsing method consists of a parser stack, that holds non- terminals, grammar symbols and tokens; a parsing table that specifies parser actions, and a driver function that interacts with the parser stack, table, and scanner. The typical actions of a CLR parser include shift, reduce, and accept or error.

The project work would include a set of predefined grammar and an interface which would convert each phase of the parsing process into a visual representation and would display onto webpage. The implementation is straight forward and simple. Then it would take any input string belonging to the grammar language and would show the acceptance or rejection of that input string and the steps one by one.

INTRODUCTION

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.

LR parsers are also known as LR(k) parsers, where L stands for left-to- right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of look ahead symbols to make decisions. LR parsing does a rightmost derivation in reverse.

LR(1) parser works on complete set of LR(1) grammar, which makes full use of the look-ahead symbols. It generates a large table with many states. An LR(1) item is a two-component element of the form where the first component is a marked production, called the core of the item and the second is a look ahead character that belongs to the super set.

REQUIREMENTS

To run a script implementing a bottom-up parser, you will need:

1. A programming language: You will need a programming language to write the parser script. Popular choices include Python, Java, and **C++.**

2. A parser generator: You can either write the parser by hand, which can be time-consuming and error-prone, or use a parser generator tool such as ANTLR, Bison, or Yacc. These tools can automatically generate parser code from a grammar specification.

3. A grammar specification: You will need a grammar specification that describes the syntax rules of the language you want to parse. This grammar specification should be in a formal grammar notation such as Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).

4. Test input: You will need test input to verify that the parser is correctly recognizing and parsing the language. This test input should cover a range of cases, including valid and invalid input.

5. Integrated Development Environment (IDE): You will need an IDE that supports the programming language you are using. An IDE provides

an environment for writing, testing, and debugging code. Popular choices include **Visual Studio Code**, Eclipse, and IntelliJ IDEA.

6. Understanding of the parsing algorithm: You will need to understand how the bottom-up parsing algorithm works and how to implement it in your chosen programming language. This includes knowledge of parsing techniques such as shift-reduce and reduce-reduce conflicts, and how to resolve them.

ALGORITHM

Algorithm for constructing LR(1) sets of items

Input: An augmented grammar G^l.

Output: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G^l.

```
SetOfItems CLOSURE(I) {
   repeat
         for (each item [A→α.Bβ,a] in I )
           for (each production B→ γ in
            G^l ) for (each terminal b in
            FIRST(βa) )
              add [B→. γ,b] to set
   I;until no more items are added
   to I; return I;
}
```

```
SetOfItems GOTO(I,X) {
   initialize J to be the empty
   set;
        for (each item [A→α.Xβ,a] in I
             ) add item [A→αX.β,a] to
             set J;
         return CLOSURE(J);
}
SetOfItems items(G') {
   initialize C to
   CLOSURE({[S'→.S,$]});repeat
        for (each set of items I in C )
          for (each grammar symbol X )
            if (GOTO(I,X) is not empty and not
              in C )add GOTO(I,X) to C;
  until no new set of items are added to C;
}
```

Algorithm for constructing Canonical LR(1) parsing table

Input: An augmented grammar G^l.

Output: The canonical-LR parsing table functions ACTION and GOTO for G^l.

Method:

Construct $C^l = \{ I0, I1,....,In\}$, the collection of sets of LR(1) items for G^l.

State *i* of the parser is constructed from *Ii*. The parsing action for state i is determined as follows.

A) If $[A \rightarrow \alpha.a\beta,b]$ is in *Ii* and GOTO(*Ii* ,a) = *Ij* , then set ACTION[*i*,a] to "shift *j*". Hence a must be a terminal.

B) If $[A \rightarrow \alpha.,\alpha]$ is in *Ii*, $A \neq S^l$, then set ACTION[*i*,a] to "reduce $A \rightarrow \alpha$."

C) If $[S^l \rightarrow S.,\$]$ is in *Ii*, then set ACTION[*i*,$] to "accept".

If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

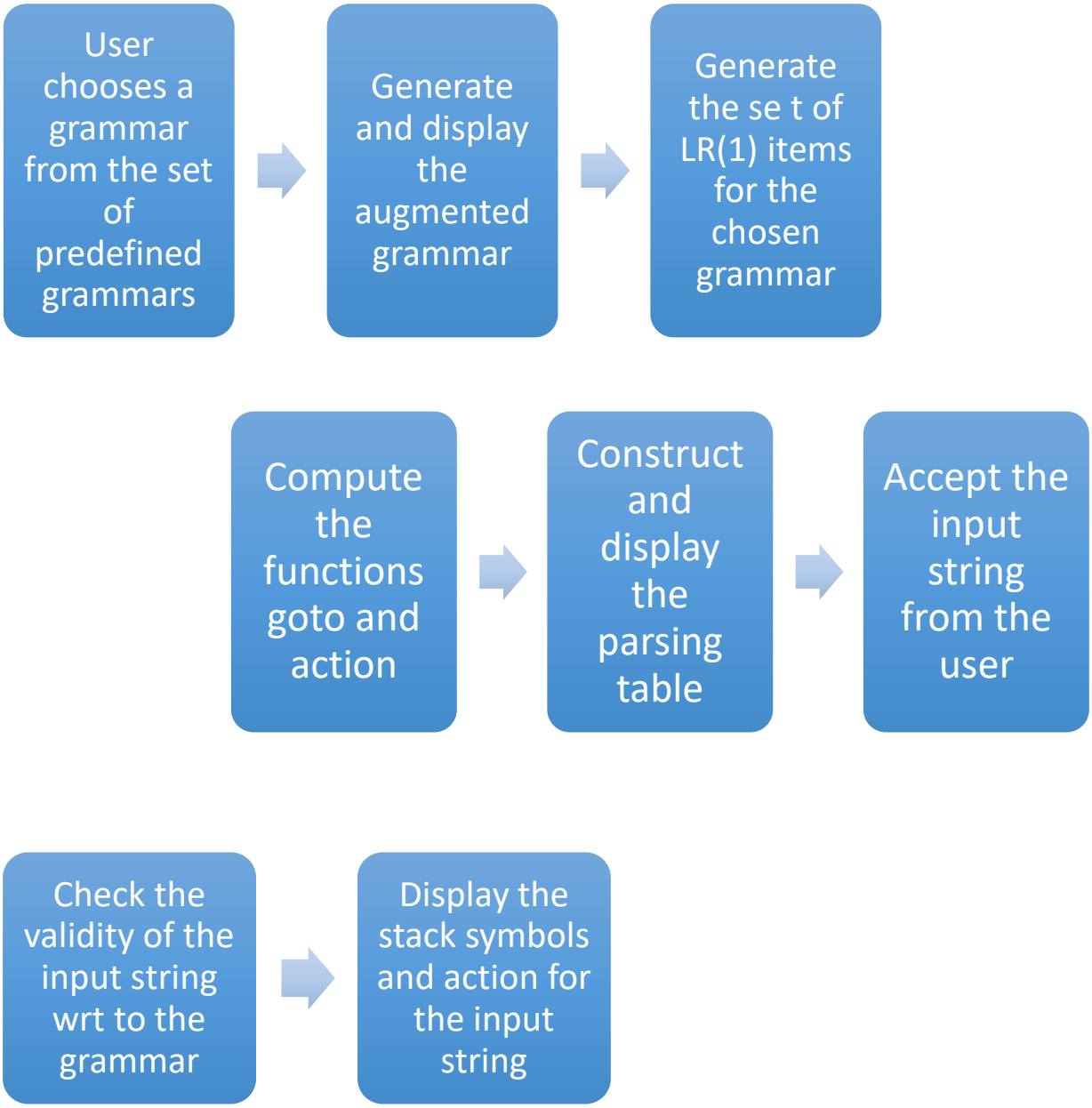
The goto transitions for state *i* are constructed for all

nonterminals A using the rule: If GOTO(*Ii* ,a) = *Ij* ,then GOTO[*i*, A] = *j*.

All entries not defined by rules (2) and (3) are made "error".

The initial state of the parser is the one constructed from the set of items containing [$S^{l} \rightarrow .S,\$$].

DATA FLOW DIAGRAM



SOURCE CODE

```
#include<iostream>
#include<conio.h>
#include<string.h>

using namespace std;

char prod[20][20],listofvar[26]="ABCDEFGHIJKLMNOPQR";
int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];
int noitem=0;

struct Grammar
{
char lhs;
char rhs[8];
}g[20],item[20],clos[20][10];

int isvariable(char variable)
{
for(int i=0;i<novar;i++)
if(g[i].lhs==variable)
        return i+1;
return 0;
}
void findclosure(int z, char a)
{
int n=0,i=0,j=0,k=0,l=0;
for(i=0;i<arr[z];i++)
{
for(j=0;j<strlen(clos[z][i].rhs);j++)
```

```
{
	if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)
	{
		clos[noitem][n].lhs=clos[z][i].lhs;
		strcpy(clos[noitem][n].rhs,clos[z][i].rhs);
		char temp=clos[noitem][n].rhs[j];
		clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];
		clos[noitem][n].rhs[j+1]=temp;
		n=n+1;
	}
}
}
for(i=0;i<n;i++)
{
for(j=0;j<strlen(clos[noitem][i].rhs);j++)
{
	if(clos[noitem][i].rhs[j]=='.' &&
isvariable(clos[noitem][i].rhs[j+1])>0)
	{
		for(k=0;k<novar;k++)
		{
			if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
			{
				for(l=0;l<n;l++)
					if(clos[noitem][l].lhs==clos[0][k].lhs &&
strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)
						break;
				if(l==n)
				{
					clos[noitem][n].lhs=clos[0][k].lhs;
				strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
					n=n+1;
```

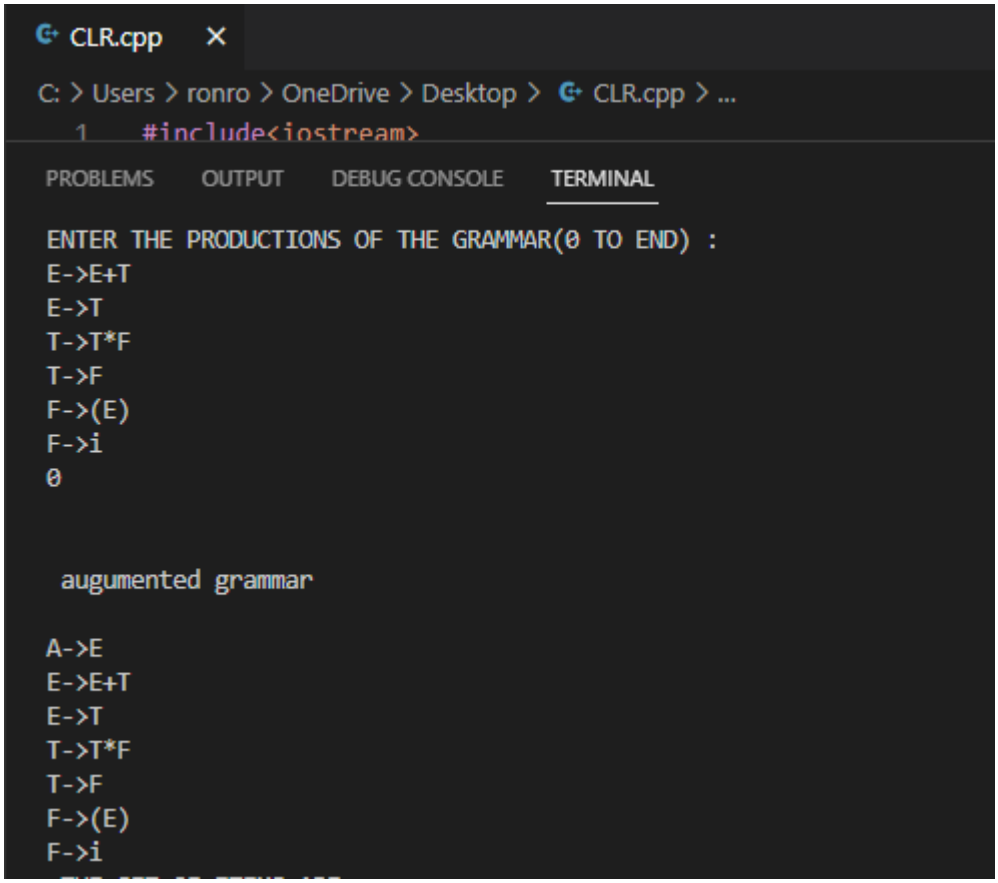
```
                        }
                    }
                }
            }
        }
arr[noitem]=n;
int flag=0;
for(i=0;i<noitem;i++)
{
if(arr[i]==n)
{
        for(j=0;j<arr[i];j++)
        {
                int c=0;
                for(k=0;k<arr[i];k++)
                        if(clos[noitem][k].lhs==clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
                                c=c+1;
                if(c==arr[i])
                {
                        flag=1;
                        goto exit;
                }
        }
}
}
exit:;
if(flag==0)
arr[noitem++]=n;
}
```

```
int main()
{
cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO
END) :\n";
do
{
cin>>prod[i++];
}while(strcmp(prod[i-1],"0")!=0);
for(n=0;n<i-1;n++)
{
m=0;
j=novar;
g[novar++].lhs=prod[n][0];
for(k=3;k<strlen(prod[n]);k++)
{
        if(prod[n][k] != '|')
        g[j].rhs[m++]=prod[n][k];
        if(prod[n][k]=='|')
        {
                g[j].rhs[m]='\0';
                m=0;
                j=novar;
                g[novar++].lhs=prod[n][0];
        }
}
}
for(i=0;i<26;i++)
if(!isvariable(listofvar[i]))
        break;
g[0].lhs=listofvar[i];
char temp[2]={g[1].lhs,'\0'};
strcat(g[0].rhs,temp);
```

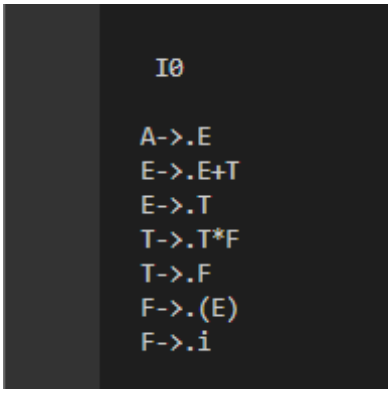
```
cout<<"\n\n augumented grammar \n";
for(i=0;i<novar;i++)
cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";

for(i=0;i<novar;i++)
{
clos[noitem][i].lhs=g[i].lhs;
strcpy(clos[noitem][i].rhs,g[i].rhs);
if(strcmp(clos[noitem][i].rhs,"ε")==0)
        strcpy(clos[noitem][i].rhs,".");
else
{
        for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)
                clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
        clos[noitem][i].rhs[0]='.';
}
}
arr[noitem++]=novar;
for(int z=0;z<noitem;z++)
{
char list[10];
int l=0;
for(j=0;j<arr[z];j++)
{
        for(k=0;k<strlen(clos[z][j].rhs)-1;k++)
        {
                if(clos[z][j].rhs[k]=='.')
                {
                        for(m=0;m<l;m++)
                                if(list[m]==clos[z][j].rhs[k+1])
                                        break;
                        if(m==l)
```

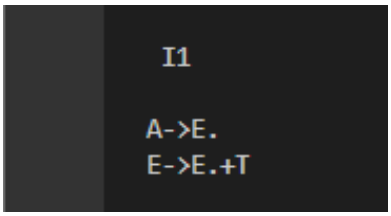
```
                        list[l++]=clos[z][j].rhs[k+1];
                }
        }
}
for(int x=0;x<l;x++)
        findclosure(z,list[x]);
}
cout<<"\n THE SET OF ITEMS ARE \n\n";
for(int z=0; z<noitem; z++)
{
cout<<"\n I"<<z<<"\n\n";
for(j=0;j<arr[z];j++)
        cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";

}

}
```

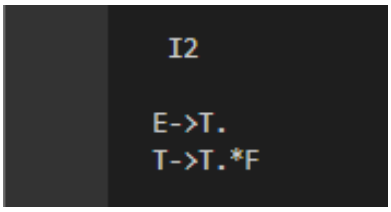
OUTPUT

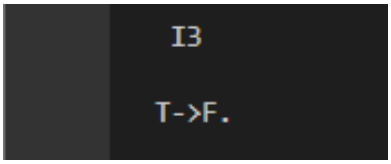


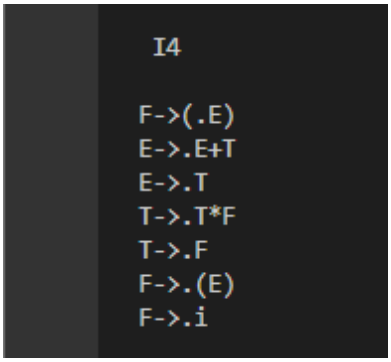
Items: -

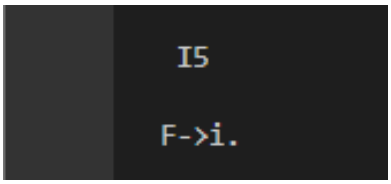


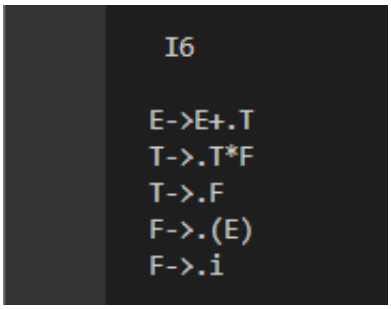

I1
A->E.
E->E.+T

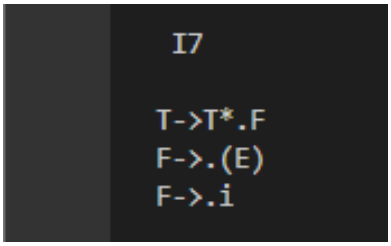

I2
E->T.
T->T.*F

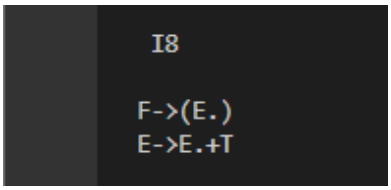

I3
T->F.

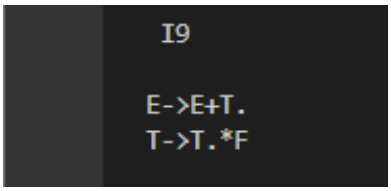

I4
F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.i

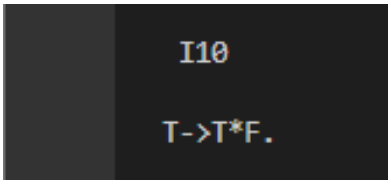

I5
F->i.

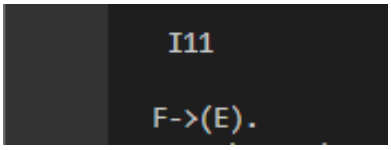

I6
E->E+.T
T->.T*F
T->.F
F->.(E)
F->.i


I7
T->T*.F
F->.(E)
F->.i


I8
F->(E.)
E->E.+T


I9
E->E+T.
T->T.*F


I10
T->T*F.


I11
F->(E).

RELEVANCE with respect to other compiler phases

In a typical compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors w that it can continue promising the remainder of its input.

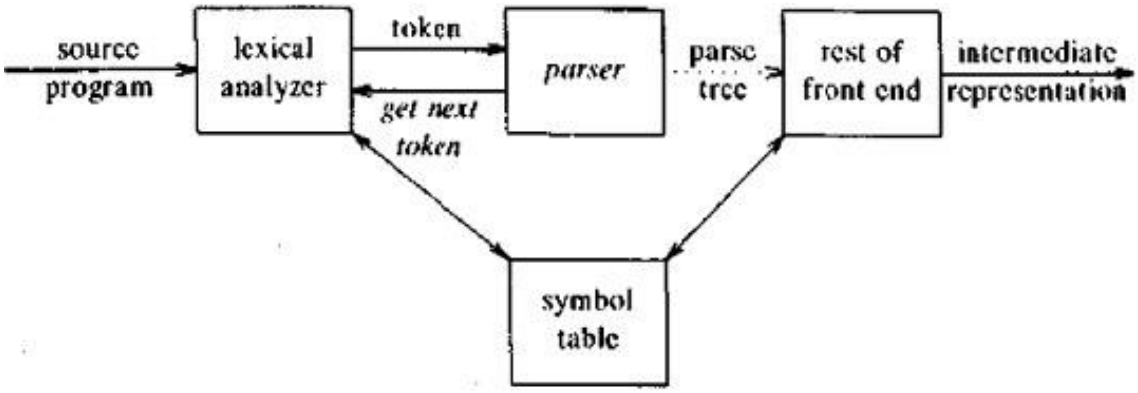


Fig. 4.1. Position of parser in compiler model.

The parsing takes place in the syntax analysis phase. The job of this phase isto build a relationship between the lexeme

values and generate a syntax tree. One more job of the phase to handle errors. Application programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.

The generated tree is then passed onto the intermediate code representation. Output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer. In practice, there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, per- forming type checking and other kinds of semantic analysis, and generating intermediate code.

RESULT

The final application interface would represent a pictorial and a visual figure of each step of the process of CLR parsing. The grammar for the parser is assumed i.e. fixed in the initial phase itself and the user is given the freedom of entering any input string possible. The parser would generate LR(1) set of items and display that. Then it would generate a parsing table given when the generated item set is fed on to it.

Then the parsing of any user input is shown step by step. But the only static thing in the project is the set of pre-defined grammar. The add-on to the project would be to generalize any grammar that is accepted by the user and then generate LR(1) set of tokens on it and then parse the input string.

We could add the other parsing techniques to the implementation too and generate an illustrative case study on which parser would be the best fit for a given grammar and for a given set of input strings.

REFERENCES

- Parsing tables examples and solved grammar: ORCCA
- Compiler Design Lectures by **Ravi Chandra Babu**
- Bottom-Up Parser **Tutorials Point:** https://www.tutorialspoint.com/compiler_design/compiler_design_parser.htm
- **LR(1) Parsing** : Handout written by Maggie Johnson and revised by Julie Zelenski, Stanford University
- **Compilers: Principles, techniques and tools** by Alfred V Rao, Ravi Sethi, Jeffrey D Ullman