Emulator for Bottom Up Parsing - CLR

A COURSE PROJECT REPORT

By

ANASUA SAHA (RA2011003010616)
ANANYA RAVICHANDRAN (RA2011003010630)
KONDREDDY BHANU SREE (RA2011003010676)

Under the guidance of

Dr. G. ABIRAMI

In partial fulfillment for the Course

18CSC304J - COMPILER DESIGN in CTECH

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District

CERTIFICATE

This is to certify that **ANASUA SAHA (RA2011003010616), ANANYA RAVICHANDRAN (RA2011003010630) and KONDREDDY BHANU SREE (RA2011003010676)** have completed the **“Emulator for Bottom Up Parsing- CLR”**

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

Submitted by
1. **ANASUA SAHA**
2. **ANANYA RAVICHANDRAN**
3. **KONDREDDY BHANU SREE**

Guided by

Dr. G. Abirami
(Dept of CSE, SRMIST)
(Compiler Design Faculty, Dept. of CSE)

ACKNOWLEDGEMENT

We express our sincere gratitude to Dr. G. Abirami, Dept. of Computer Science and Technology, SRMIST, for her stimulating guidance, continuous encouragement and supervision throughout the course of present work.

Through the course and the time that we invested in completing the project, we understood the steps of parsing in detail and also the art of representing each individual solution in a systematic and graphical way using good front-end tools.

We thank Dr. G. Abirami for the opportunity to implement the algorithms that we had learnt in the theory in a real-time project which gave us an in-depth knowledge and expertise on the topic and the subject as a whole.

Signature(s) of Students

ANASUA SAHA (RA2011003010616)
ANANYA RAVICHANDRAN (RA2011003010630)
KONDREDDY BHANU SREE (RA2011003010676)

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 invalid reductions, the states need to carry more information. Extra information is put into 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 pretty 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 also 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 a large number of 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.

Literature Review

The idea is to generate a good front end for the parsing process and to represent each step of the algorithm in a illustrative graphical pattern which is easy to interpret because of the visual appeal. So the choice of tools and tech used revolve around finding a API which gives us rich UI features and easy to use database to store session id's and to pass values from screen to another.

Choice of programming interfaces:

- **Front end:** The front end is split into 2 things. A general technology which takes care of the overall look and feel and a specific framework which has built in features to achieve graphic utilities.

  - **HTML5 +CSS3:** General functionality, rendering of pages, display of each step, navigation from servlet to another.

  - **JavaScript:** Front end validations of input grammar. After constructing the parsing table, we accept the input string to check whether it gets accepted by the parser or not. To do so, before submitting to the server, we do some front-end checking. If the input string contains an invalid terminal or non-terminal, we pop it out on form submission itself.

  - **Bootstrap:** Twitter Bootstrap is a web-application framework which contains a vast set of library functions to achieve some graphic utilities like modal, tool-tips, carousels, animations, transitions, etc

  - **jQuery library:** jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. Whether you're building highly interactive web applications or you just need to add a date picker to a form control, jQuery UI is the perfect choice.

- **Back –end:** The choice was between PHP which would be the optimal choice for web-programming and Java used as a servlet. But since this involves a lot of string manipulations and set generations we need to have an ample set of built in functions to do so.
  Therefore, we went forward with the decision of making use of Java as a servlet.

  As of now there exists no need of a separate database for storing temporal data, but for handling sessions and maintain synchronization between multiple servlets we may need a user friendly database.

  Mysql is the obvious choice, since it's easy to manipulate and query. It has got good embedding functions between native user code and sql database core operations. The schema of the database, not decided yet but would be pretty simple and would contain not more than 4-5 fields.

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 [$A \rightarrow \alpha.B\beta$,a] in I )
                   for (each  production $B \rightarrow \gamma$ in $G^{l}$ )
                      for (each terminal b in FIRST($\beta$a) )
                         add [$B \rightarrow .\gamma$,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 [$A \rightarrow \alpha.X\beta$,a] in I )
                        add item [$A \rightarrow \alpha X.\beta$,a] to set J;
                return CLOSURE(J);
}

SetOfItems items($G^{l}$) {
     initialize C to CLOSURE({[$S^{l} \rightarrow .S$,$]});
     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:

1. Construct $C^l = \{ I_0, I_1, \ldots, I_n \}$, the collection of sets of LR(1) items for G^l.
2. State *i* of the parser is constructed from I_i. The parsing action for

state i is determined as follows.

- If [A→α.aβ,b] is in I_i and GOTO(I_i ,a) = I_j , then set ACTION[*i*,a] to "shift *j*". Hence a must be a terminal.

- If [A→α.,α] is in I_i, A ≠ S^l, then set ACTION[*i*,a] to

"reduce A→α."
- If [S^l→S.,\$] is in I_i, then set ACTION[*i*,\$] to "accept".

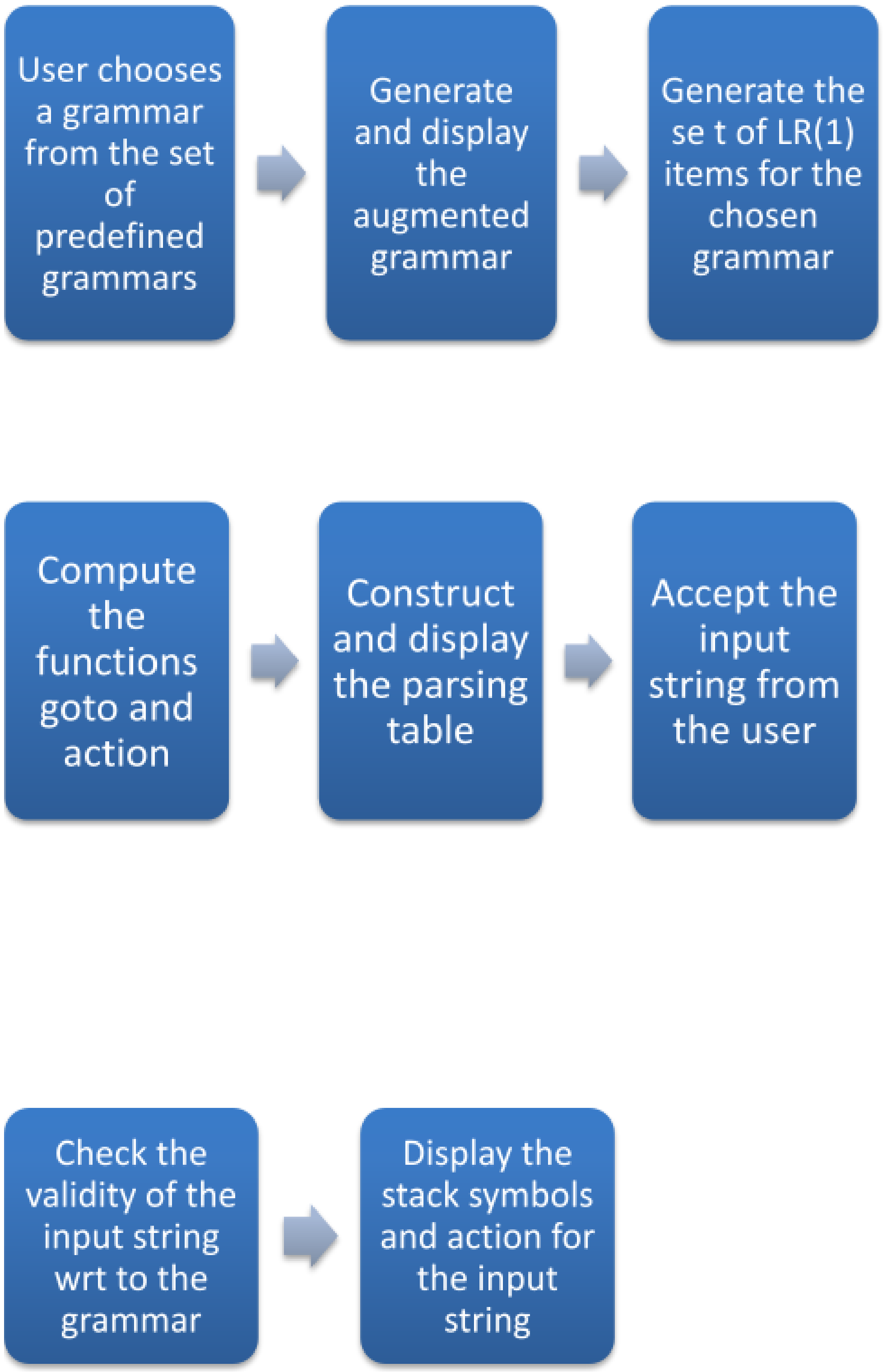
If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

1. The goto transitions for state *i* are constructed for all nonterminals A using the rule: If GOTO(I_i ,a) = I_j ,then GOTO[*i*, A] = *j*.

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

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

Data Flow Diagram


User chooses a grammar from the set of predefined grammars
Generate and display the augmented grammar
Generate the se t of LR(1) items for the chosen grammar
Compute the functions goto and action
Construct and display the parsing table
Accept the input string from the user
Check the validity of the input string wrt to the grammar
Display the stack symbols and action for the input string

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.

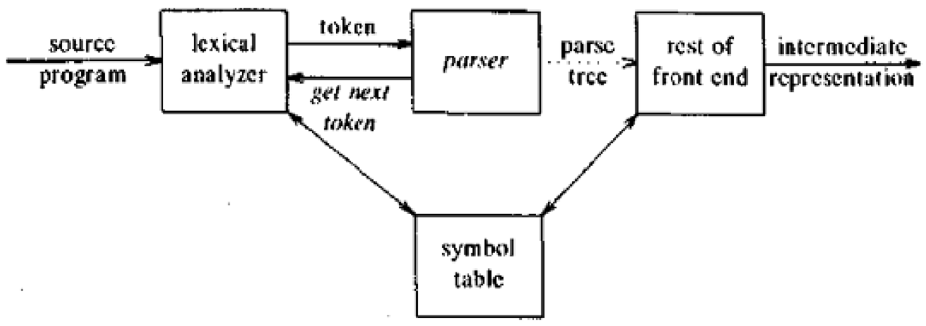
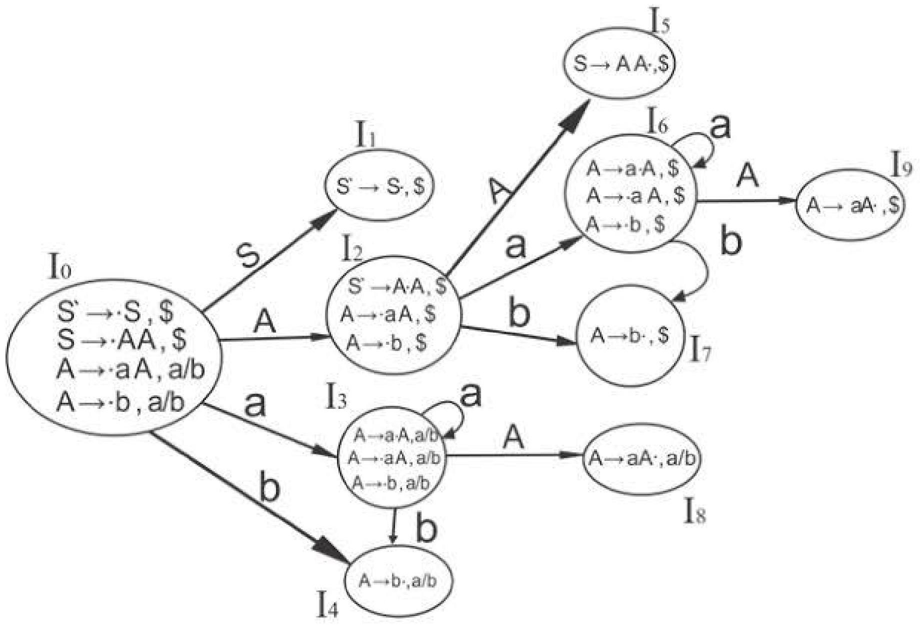


Fig. 4.1. Position of parser in compiler model.

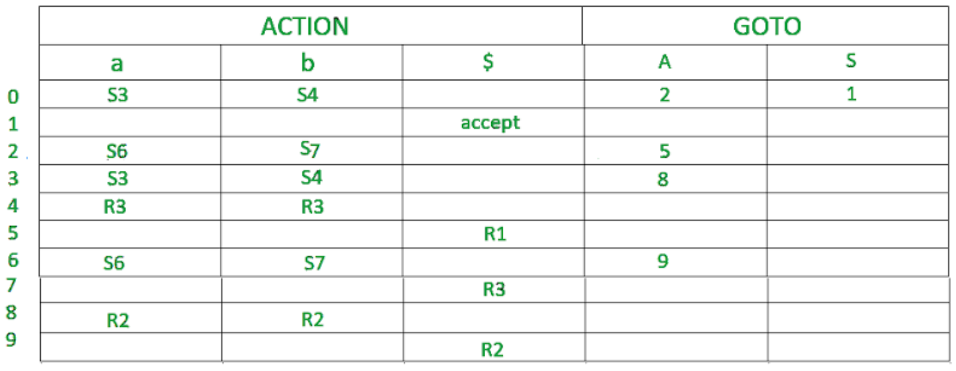
The parsing takes place in the syntax analysis phase. The job of this phase is to build a relationship between the lexime values and generate a syntax tree. One more job of the phase to handle errors. Application programmers frequently write incorrect programs, and a good compiler should assist the programmer 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.

<u>Example of CLR Parser:</u>



CLR PARSING TABLE:



	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S6	S7		5	
3	S3	S4		8	
4	R3	R3			
5			R1		
6	S6	S7		9	
7			R3		
8	R2	R2			
9			R2		

Implementation(Code)

FIRST AND FOLLOW:

```
from re import *
from collections import OrderedDict

t_list=OrderedDict()
nt_list=OrderedDict()
production_list=[]

# ---------------------------------------------------------------

class Terminal:

    def __init__(self, symbol):
        self.symbol=symbol

    def __str__(self):
        return self.symbol

# ---------------------------------------------------------------

class NonTerminal:

    def __init__(self, symbol):
        self.symbol=symbol
        self.first=set()
        self.follow=set()

    def __str__(self):
        return self.symbol

    def add_first(self, symbols): self.first |= set(symbols) #union operation

    def add_follow(self, symbols): self.follow |= set(symbols)

# ---------------------------------------------------------------

def compute_first(symbol): #chr(1013) corresponds (ϵ) in Unicode

    global production_list, nt_list, t_list

# if X is a terminal then first(X) = X
    if symbol in t_list:
        return set(symbol)

    for prod in production_list:
        head, body=prod.split('->')

        if head!=symbol: continue
```

```
# if X -> is a production, then first(X) = epsilon
        if body=='':
            nt_list[symbol].add_first(chr(1013))
            continue



        for i, Y in enumerate(body):
# for X -> Y1 Y2 ... Yn, first(X) = non-epsilon symbols in first(Y1)
# if first(Y1) contains epsilon,
#   first(X) = non-epsilon symbols in first(Y2)
#   if first(Y2) contains epsilon
#   ...
            if body[i]==symbol: continue
            t=compute_first(Y)
            nt_list[symbol].add_first(t-set(chr(1013)))
            if chr(1013) not in t:
                break
# for i=1 to n, if Yi contains epsilon, then first(X)=epsilon
            if i==len(body)-1:
                nt_list[symbol].add_first(chr(1013))

    return nt_list[symbol].first

# ---------------------------------------------------------------

def get_first(symbol): #wrapper method for compute_first

    return compute_first(symbol)

# ---------------------------------------------------------------

def compute_follow(symbol):

    global production_list, nt_list, t_list

# if A is the start symbol, follow (A) = $
    if symbol == list(nt_list.keys())[0]: #this is okay since I'm using an
OrderedDict
        nt_list[symbol].add_follow('$')

    for prod in production_list:
        head, body=prod.split('->')

        for i, B in enumerate(body):
            if B != symbol: continue

# for A -> aBb, follow(B) = non-epsilon symbols in first(b)
            if i != len(body)-1:
                nt_list[symbol].add_follow(get_first(body[i+1]) - set(chr(1013)))

# if A -> aBb where first(b) contains epsilon, or A -> aB then follow(B) =
follow (A)
```

```
        if i == len(body)-1 or chr(1013) in get_first(body[i+1]) and B !=
head:
            nt_list[symbol].add_follow(get_follow(head))

# ---------------------------------------------------------------

def get_follow(symbol):

    global nt_list, t_list

    if symbol in t_list.keys():
        return None

    return nt_list[symbol].follow

# ---------------------------------------------------------------

def main(pl=None):

    print('''Enter the grammar productions (enter 'end' or return to stop)
#(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})''')

    global production_list, t_list, nt_list
    ctr=1

    #t_regex, nt_regex=r'[a-z\W]', r'[A-Z]'

    if pl==None:

        while True:

            #production_list.append(input('{})\t'.format(ctr)))

            production_list.append(input().replace(' ', ''))

            if production_list[-1].lower() in ['end', '']:
                del production_list[-1]
                break

            head, body=production_list[ctr-1].split('->')

            if head not in nt_list.keys():
                nt_list[head]=NonTerminal(head)

            #for all terminals in the body of the production
            for i in body:
                if not 65<=ord(i)<=90:
                    if i not in t_list.keys(): t_list[i]=Terminal(i)
            #for all non-terminals in the body of the production
                elif  i not in nt_list.keys(): nt_list[i]=NonTerminal(i)

            ctr+=1
    '''if pl!=None:

        for i, prod in enumerate(pl):

            if prod.lower() in ['end', '']:
                del pl[i:]
                break

            head, body=prod.split('->')

            if head not in nt_list.keys():
                nt_list[head]=NonTerminal(head)

            #for all terminals in the body of the production
            for i in finditer(t_regex, body):
                s=i.group()
                if s not in t_list.keys(): t_list[s]=Terminal(s)

            #for all non-terminals in the body of the production
            for i in finditer(nt_regex, body):
                s=i.group()
                if s not in nt_list.keys(): nt_list[s]=NonTerminal(s)'''

    return pl
# -----------------------------------------------------------------

if __name__=='__main__':

    main()
```

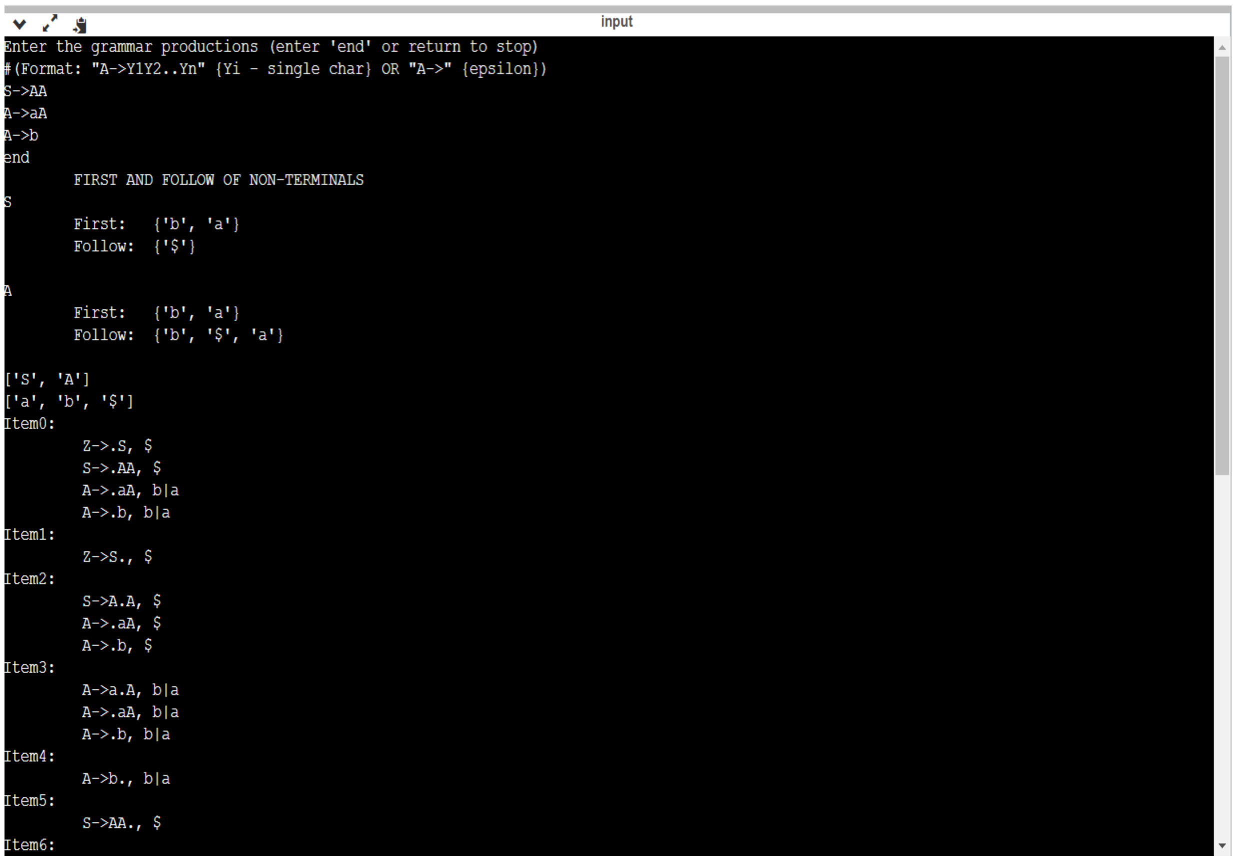
CLR PARSER:

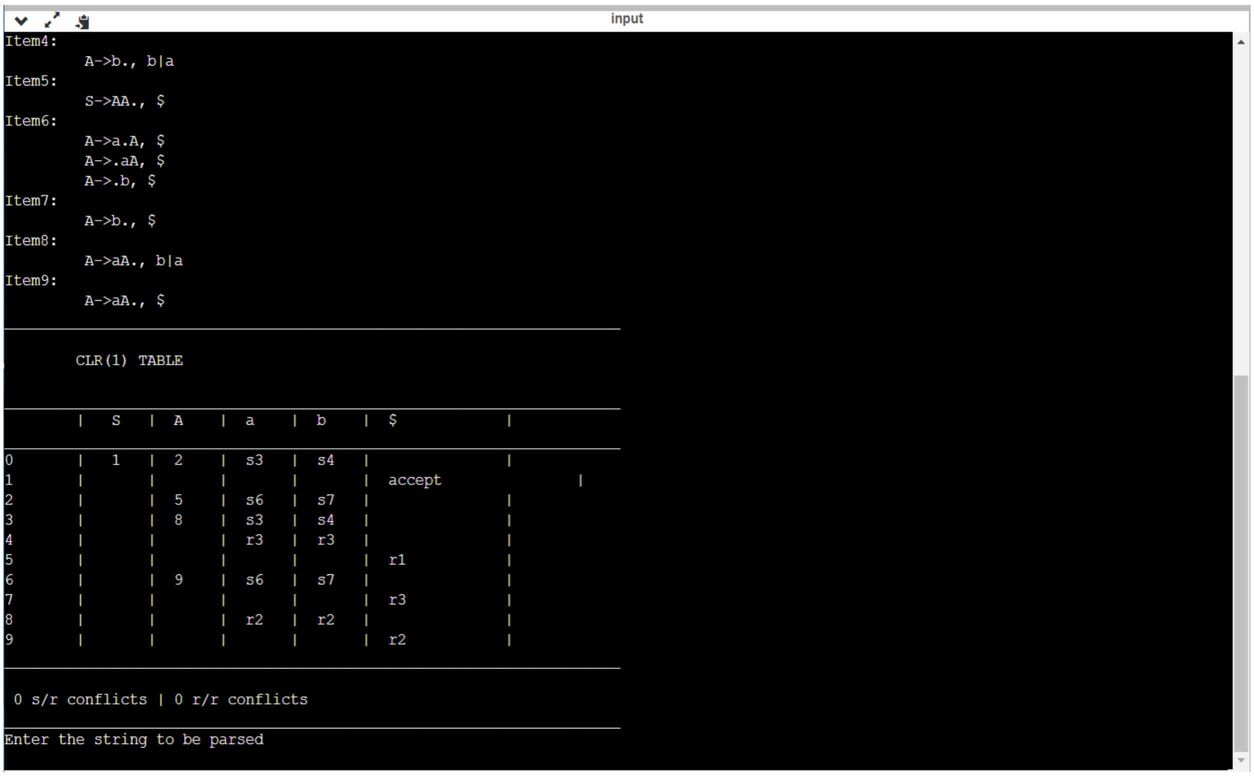
```
from collections import deque
from collections import OrderedDict
from pprint import pprint
import firstfollow
from firstfollow import production_list, nt_list as ntl, t_list as tl
nt_list, t_list=[], []

class State:

    _id=0
    def __init__(self, closure):
        self.closure=closure
        self.no=State._id
        State._id+=1

class Item(str):
    def __new__(cls, item, lookahead=list()):
        self=str.__new__(cls, item)
        self.lookahead=lookahead
        return self

    def __str__(self):
        return super(Item, self).__str__()+", "+'|'.join(self.lookahead)


def closure(items):

    def exists(newitem, items):

        for i in items:
            if i==newitem and
sorted(set(i.lookahead))==sorted(set(newitem.lookahead)):
                return True
        return False


    global production_list

    while True:
        flag=0
        for i in items:

            if i.index('.')==len(i)-1: continue

            Y=i.split('->')[1].split('.')[1][0]

            if i.index('.')+1<len(i)-1:
                lastr=list(firstfollow.compute_first(i[i.index('.')+2])-set(chr(1013)))

            else:
                lastr=i.lookahead

            for prod in production_list:
                head, body=prod.split('->')

                if head!=Y: continue

                newitem=Item(Y+'->.'+body, lastr)

                if not exists(newitem, items):
                    items.append(newitem)
                    flag=1
        if flag==0: break

    return items

def goto(items, symbol):

    global production_list
    initial=[]
    for i in items:
        if i.index('.')==len(i)-1: continue

        head, body=i.split('->')
        seen, unseen=body.split('.')


        if unseen[0]==symbol and len(unseen) >= 1:
            initial.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:],
i.lookahead))

    return closure(initial)


def calc_states():

    def contains(states, t):

        for s in states:
            if len(s) != len(t): continue

            if sorted(s)==sorted(t):
                for i in range(len(s)):
                        if s[i].lookahead!=t[i].lookahead: break
                else: return True

        return False

    global production_list, nt_list, t_list

    head, body=production_list[0].split('->')


    states=[closure([Item(head+'->.'+body, ['$'])])]

    while True:
        flag=0
        for s in states:

            for e in nt_list+t_list:

                t=goto(s, e)
                if t == [] or contains(states, t): continue

                states.append(t)
                flag=1

        if not flag: break

    return states
```

```
def make_table(states):

    global nt_list, t_list

    def getstateno(t):

        for s in states:
            if len(s.closure) != len(t): continue

            if sorted(s.closure)==sorted(t):
                for i in range(len(s.closure)):
                        if s.closure[i].lookahead!=t[i].lookahead: break
                else: return s.no

        return -1

    def getprodno(closure):

        closure=''.join(closure).replace('.', '')
        return production_list.index(closure)

    SLR_Table=OrderedDict()

    for i in range(len(states)):
        states[i]=State(states[i])

    for s in states:
        SLR_Table[s.no]=OrderedDict()

        for item in s.closure:
            head, body=item.split('->')
            if body=='.':
                for term in item.lookahead:
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
                continue

            nextsym=body.split('.')[1]
            if nextsym=='':
                if getprodno(item)==0:
                    SLR_Table[s.no]['$']='accept'
                else:
                    for term in item.lookahead:
                        if term not in SLR_Table[s.no].keys():
                            SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                        else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
                continue

            nextsym=nextsym[0]
            t=goto(s.closure, nextsym)
            if t != []:
                if nextsym in t_list:
```

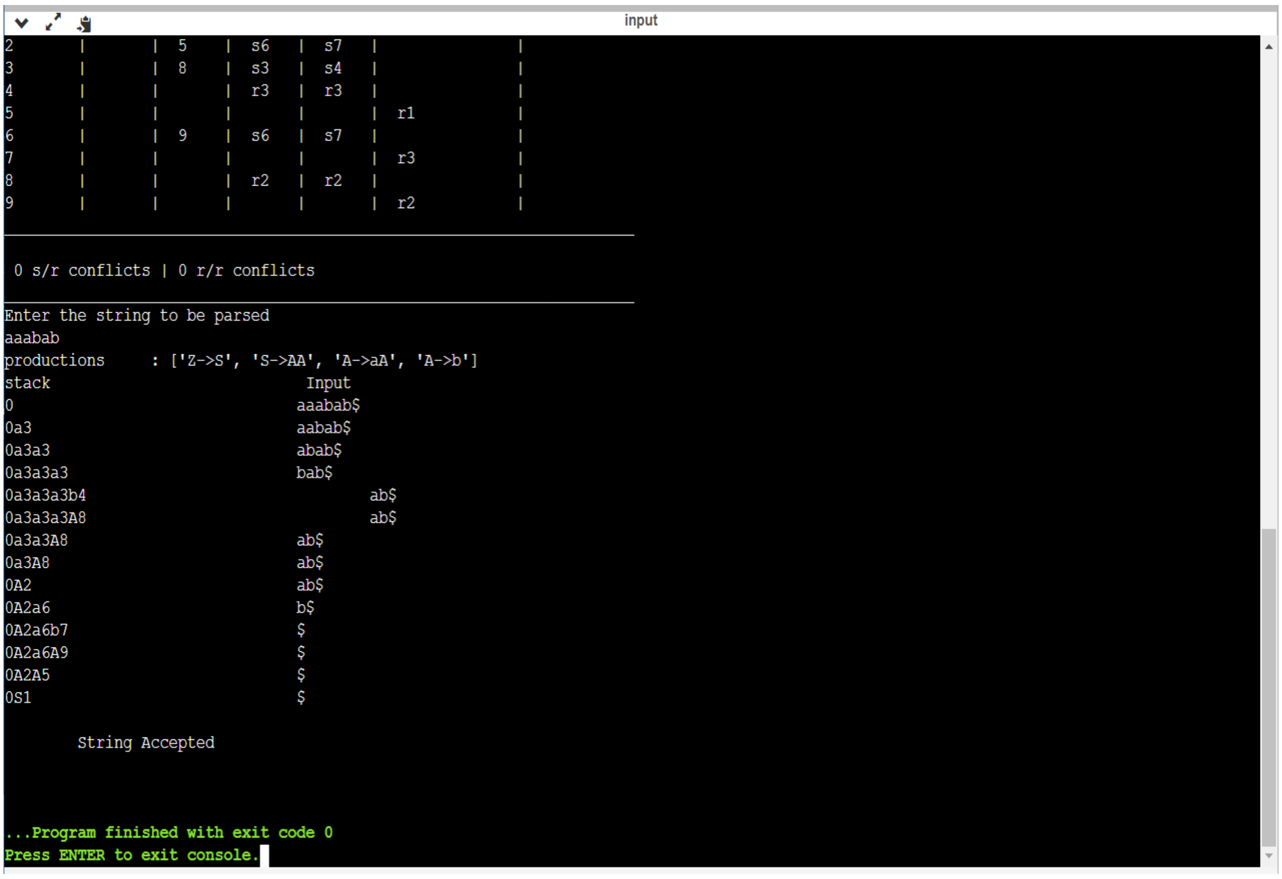
```
                if nextsym not in SLR_Table[s.no].keys():
                    SLR_Table[s.no][nextsym]={'s'+str(getstateno(t))}
                else: SLR_Table[s.no][nextsym] |= {'s'+str(getstateno(t))}

            else: SLR_Table[s.no][nextsym] = str(getstateno(t))

    return SLR_Table

def augment_grammar():

    for i in range(ord('Z'), ord('A')-1, -1):
        if chr(i) not in nt_list:
            start_prod=production_list[0]
            production_list.insert(0, chr(i)+'->'+start_prod.split('->')[0])
            return

def main():

    global production_list, ntl, nt_list, tl, t_list

    firstfollow.main()

    print("\tFIRST AND FOLLOW OF NON-TERMINALS")
    for nt in ntl:
        firstfollow.compute_first(nt)
        firstfollow.compute_follow(nt)
        print(nt)
        print("\tFirst:\t", firstfollow.get_first(nt))
        print("\tFollow:\t", firstfollow.get_follow(nt), "\n")


    augment_grammar()
    nt_list=list(ntl.keys())
    t_list=list(tl.keys()) + ['$']

    print(nt_list)
    print(t_list)

    j=calc_states()

    ctr=0
    for s in j:
        print("Item{}:".format(ctr))
        for i in s:
            print("\t", i)
        ctr+=1

    table=make_table(j)

print('_____________________________________________________________
____________')
    print("\n\tCLR(1) TABLE\n")
    sym_list = nt_list + t_list
    sr, rr=0, 0
print('_________________________________________________________________________')
    print('\t|  ','\t|  '.join(sym_list),'\t\t|')
print('_________________________________________________________________________')
    for i, j in table.items():

        print(i, "\t|  ", '\t|  '.join(list(j.get(sym,' ') if type(j.get(sym))in (str , None)
else next(iter(j.get(sym,' ')))  for sym in sym_list)),'\t\t|')
        s, r=0, 0

        for p in j.values():
            if p!='accept' and len(p)>1:
                p=list(p)
                if('r' in p[0]): r+=1
                else: s+=1
                if('r' in p[1]): r+=1
                else: s+=1
        if r>0 and s>0: sr+=1
        elif r>0: rr+=1
print('_________________________________________________________________________')
    print("\n", sr, "s/r conflicts |", rr, "r/r conflicts")
print('_________________________________________________________________________')
    print("Enter the string to be parsed")
    Input=input()+'$'
    try:
        stack=['0']
        a=list(table.items())
        '''print(a[int(stack[-1])][1][Input[0]])
        b=list(a[int(stack[-1])][1][Input[0]])
        print(b[0][0])
        print(a[0][1]["S"])'''
        print("productions\t:",production_list)
        print('stack',"\t \t\t \t",'Input')
        print(*stack,"\t \t\t \t",*Input,sep="")
        while(len(Input)!=0):
            b=list(a[int(stack[-1])][1][Input[0]])
            if(b[0][0]=="s" ):
                #s=Input[0]+b[0][1:]
                stack.append(Input[0])
                stack.append(b[0][1:])
                Input=Input[1:]
                print(*stack,"\t \t\t \t",*Input,sep="")
            elif(b[0][0]=="r" ):
                s=int(b[0][1:])
                #print(len(production_list),s)
                l=len(production_list[s])-3
                #print(l)
                prod=production_list[s]
                l*=2
                l=len(stack)-l
                stack=stack[:l]
                s=a[int(stack[-1])][1][prod[0]]
                #print(s,b)
                stack+=list(prod[0])
                stack.append(s)
                print(*stack,"\t \t\t \t",*Input,sep="")
            elif(b[0][0]=="a"):
                print("\n\tString Accepted\n")
                break
    except:
        print('\n\tString INCORRECT for given Grammar!\n')
    return

if __name__=="__main__":
    main()
```

OUTPUT:


```
Enter the grammar productions (enter 'end' or return to stop)
#(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})
S->AA
A->aA
A->b
end
        FIRST AND FOLLOW OF NON-TERMINALS
S

        First:   {'b', 'a'}
        Follow:  {'$'}


A

        First:   {'b', 'a'}
        Follow:  {'b', '$', 'a'}

['S', 'A']
['a', 'b', '$']
Item0:
         Z->.S, $
         S->.AA, $
         A->.aA, b|a
         A->.b, b|a
Item1:
         Z->S., $
Item2:
         S->A.A, $
         A->.aA, $
         A->.b, $
Item3:
         A->a.A, b|a
         A->.aA, b|a
         A->.b, b|a
Item4:
         A->b., b|a
Item5:
         S->AA., $
Item6:
```



```
Item4:
         A->b., b|a
Item5:
         S->AA., $
Item6:
         A->a.A, $
         A->.aA, $
         A->.b, $
Item7:
         A->b., $
Item8:
         A->aA., b|a
Item9:
         A->aA., $

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

        CLR(1) TABLE

--------------------------------------------------------------------
        |   S   |  A    |  a    |  b    |  $              |
--------------------------------------------------------------------
0       |   1   |  2    |  s3   |  s4   |                 |
1       |       |       |       |       |  accept         |       |
2       |       |  5    |  s6   |  s7   |                 |
3       |       |  8    |  s3   |  s4   |                 |
4       |       |       |  r3   |  r3   |                 |
5       |       |       |       |       |  r1             |
6       |       |  9    |  s6   |  s7   |                 |
7       |       |       |       |       |  r3             |
8       |       |       |  r2   |  r2   |                 |
9       |       |       |       |       |  r2             |

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

 0 s/r conflicts | 0 r/r conflicts

--------------------------------------------------------------------
Enter the string to be parsed
```



```
2       |       |  5    |  s6   |  s7   |                 |
3       |       |  8    |  s3   |  s4   |                 |
4       |       |       |  r3   |  r3   |                 |
5       |       |       |       |       |  r1             |
6       |       |  9    |  s6   |  s7   |                 |
7       |       |       |       |       |  r3             |
8       |       |       |  r2   |  r2   |                 |
9       |       |       |       |       |  r2             |

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

 0 s/r conflicts | 0 r/r conflicts

--------------------------------------------------------------------
Enter the string to be parsed
aaabab
productions     : ['Z->S', 'S->AA', 'A->aA', 'A->b']
stack                             Input
0                                aaabab$
0a3                              aabab$
0a3a3                            abab$
0a3a3a3                          bab$
0a3a3a3b4                                ab$
0a3a3a3A8                                ab$
0a3a3A8                          ab$
0a3A8                            ab$
0A2                              ab$
0A2a6                            b$
0A2a6b7                          $
0A2a6A9                          $
0A2A5                            $
0S1                              $

        String Accepted


...Program finished with exit code 0
Press ENTER to exit console.
```

Uses

CLR (Canonical LR) parser is a type of LR parser that is more powerful than the SLR (Simple LR) and LALR (Look-Ahead LR) parsers. CLR parsers can handle a wider range of grammars, but they also require more computational resources.

Here are some additional requirements for CLR parsers:

- Grammar must be context-free: CLR parsers require that the grammar be context-free. This means that each production rule in the grammar has only one nonterminal symbol on its left-hand side.

- Grammar must be unambiguous: CLR parsers can only handle unambiguous grammars. If the grammar is ambiguous, the parser may not be able to determine the correct parse tree.

- Grammar must be augmented: The grammar must be augmented by adding a new start symbol and a new production rule that generates the old start symbol. This is necessary because CLR parsers work by building a canonical collection of LR(1) items for the augmented grammar.

- LR(1) items must be constructed: CLR parsers require the construction of LR(1) items for the augmented grammar. LR(1) items represent the states of the parser, and each item consists of a production rule, a position within that rule, and a set of lookahead symbols.

- Canonical collection of LR(1) items must be constructed: The canonical collection of LR(1) items must be constructed by applying closure and goto operations to the LR(0) items. The closure operation adds new LR(1) items to the collection, while the goto operation determines the next state of the parser.

- Parsing table must be constructed: Finally, a parsing table must be constructed from the canonical collection of LR(1) items. The parsing table is used by the parser to determine the actions to take based on the current state and lookahead symbol.

Overall, CLR parsers require a more complex process to construct the parsing table than other types of LR parsers. However, this additional complexity allows CLR parsers to handle a wider range of grammars, making them a valuable tool for many applications.

Conclusion

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

- **Compilers: Principles, techniques and tools** by Alfred V Rao, Ravi Sethi, Jeffrey D Ullman

- **LR(1) Parsing**: Handout written by Maggie Johnson and revised by Julie Zelenski, Stanford University

- **Clemson University:** www.cs.clemson.edu/parsing.pdf

- **Parsing tables examples and solved grammar: ORCCA**

- Compiler Design Lectures by **Ravi Chandra Babu**

- Bottom Up Parser **Tutorials Point:** www.tutorialspoint.com/.../compiler_design_bottom_up_**parser**.htm