# CLR PARSER

**A MINI PROJECT REPORT**

***Submitted by***

## NYASA GUPTA [RA2011031010068]

**PRERNA SHARMA [RA2011031010086]**

*Under the guidance of*

## Dr. M. Anand

(Assistant Professor, Department of Networking and Communications)

***In partial satisfaction of the requirements for the degree of***

**BACHELOR OF TECHNOLOGY**

in

## COMPUTER SCIENCE & ENGINEERING

**With specialization in Information Technology**



**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY KATTANKULATHUR - 603203**

**MAY 2023**



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

## BONAFIDE CERTIFICATE

Certified that this project report **“CLR PARSER”** is the bonafide work of **“NYASA GUPTA [RA2011031010068], PRERNA SHARMA [RA2011031010086]”** of III Year/VI Sem B.Tech (CSE) who carried out the mini project work under my supervision for the course 18CSC304J- Compiler Design in SRM Institute of Science and Technology during the academic year 2022-2023(Even Sem).

### SIGNATURE SIGNATURE

Dr. M. Anand Dr. K. Annapurani

Assistant Professor HEAD OF THE DEPARTMENT

Department of Networking Department of Networking

And Communications and Communications

**Acknowledgement**

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy**, for his encouragement towards the completion of this project.

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V. Gopal**, for bringing out novelty in all executions.

We would like to express my heartfelt thanks to **Chairperson, School of Computing Dr. Revathi Venkataraman**, for imparting confidence to complete my course project.

We extend our gratitude to our **HoD Dr. K. Annapurani Panaiyappan, Professor and Head, Department of Networking and Communications**, and my department colleague for their Support.

We are highly thankful to our course Faculty **Dr. M. Anand, Assistant Professor, Department of Networking and Communications**, for his assistance, timely suggestions and guidance throughout the duration of this course project.

# ABSTRACT

The CLR (Canonical LR) parser is a bottom-up parsing tool frequently used in computer science to analyze and comprehend the structure of computer languages. It is based on the LR parsing method and parses the input string using a finite state machine. The CLR parser is superior than the LR parser, which is less efficient and powerful. The CLR parser represents the parsing states with a canonical collection of LR (1) elements, making it more efficient and powerful than the LR parser.

The parser operates by building a parse table that represents the grammar of the input language. Based on the input symbol and the current status of the parsing, the table is then utilized to identify the next step to take. The parser keeps track of parsing states on a stack and moves or decreases the symbols on the stack based on the actions defined in the parse table.

CLR parsing has the benefit of being able to parse a broader range of grammars than other parsing approaches. Because of this, it is a popular choice for parsing sophisticated programming languages like C++, Java, and Python.

Constructing a parse table for a CLR parser, on the other hand, may be a difficult and time- consuming job, and the table size might be huge for sophisticated grammars. Despite these obstacles, CLR parsing is a powerful and widely used technique in computer science.

**CONTENTS**

Chapter 1: Introduction

* 1. Introduction……………………………………………………………………………...1
  2. Problem Statement………………………………………………………………………2
  3. Objectives………………………………………………………………………………..3

Chapter 2: Methodology and Technique

* 1. Steps Involved …………………………………………………………………………..4
  2. Example Solving ………………………………………………………………………..5

Chapter 3: Literature Survey

* 1. Existing System …………………………………………………………………………9

Chapter 4: System Architecture and Design

4.1 Architecture Diagram ………………………………………………………………….11

4.2 Interface Design ………………………………………………………………………..12

Chapter 5: Coding and Testing

5.1 Code Implementation …………………………………………………………………..13

5.2 Testing …………………………………………………………………………………36

Chapter 6: Conclusion and Future Enhancement

6.1 Future Enhancement …………………………………………………………………..41

6.2 Conclusion …………………………………………………………………………….42

Chapter 7 References ……………………………………………………………………………….43

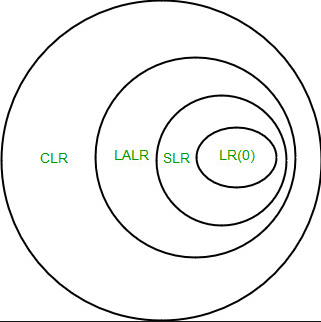
**Chapter 1**

**Introduction**

## 1.1 Introduction

The CLR parser stands for canonical LR parser. It is a more powerful LR parser. It makes use of lookahead symbols. This method uses a large set of items called LR (1) items. The main difference between LR (0) and LR (1) items is that, in LR (1) items, it is possible to carry more information in a state, which will rule out useless reduction states. This extra information is incorporated into the state by the lookahead symbol. The general syntax becomes [A->∝. B, a]

where A->∝. B is the pn and a is a terminal or right end marker $



**LR (1) item**

LR (1) item is a collection of LR (0) items and a look ahead symbol.

**LR (1) item = LR (0) item + look ahead**

The look ahead is used to determine that where we place the final item.

The look ahead always adds $ symbol for the argument production

**1.2 Problem Statement**

Implementing a parser for a given grammar using the canonical lookahead algorithm. The parser should be able to analyse the input source code and generate a parse tree or report syntax errors if the input code does not conform to the grammar rules.

The project requires a thorough understanding of compiler design principles, including lexical analysis, syntax analysis, and parsing techniques. Additionally, the project demands proficiency in programming languages such as C, C++, Java, or Python.

The key objectives of the project are:

* To develop a parser that uses the canonical lookahead algorithm to parse the input source code.
* To implement the parser in a programming language of your choice.
* To generate a parse tree or report syntax errors if the input code does not conform to the grammar rules.
* To validate the parser's correctness by testing it with a variety of inputs, including valid and invalid source code.

The project also requires the submission of a detailed report that includes a description of the grammar rules, the implementation of the parser, testing methodology, and analysis of the results obtained.

**1.3 Objectives**

1. To design and implement a parser for a given grammar using the canonical lookahead algorithm.

2. To gain a thorough understanding of compiler design principles, including lexical analysis, syntax analysis, and parsing techniques.

3. To gain proficiency in programming languages such as C, C++, Java, or Python.

4. To develop a parser that is capable of analysing the input source code and generating a parse tree or reporting syntax errors if the input code does not conform to the grammar rules.

5. To validate the correctness of the parser by testing it with a variety of inputs, including valid and invalid source code.

6. To develop effective testing methodologies to ensure the parser's reliability and accuracy.

7. To document the process of designing, implementing, and testing the parser in a detailed report that includes a description of the grammar rules, the implementation of the parser, testing methodology, and analysis of the results obtained.

8. To demonstrate the ability to apply theoretical concepts in compiler design to practical applications by successfully completing the project.

**Chapter 2**

**Methodology and Technique**

**2.1 Steps Involved**

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the greater number of states as compare to the SLR (1) parsing.

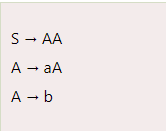
In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

* For the given input string write a context free grammar
* Check the ambiguity of the grammar
* Add Augment production in the given grammar
* Create Canonical collection of LR (0) items
* Draw a data flow diagram (DFA)
* Construct a CLR (1) parsing table

**2.2 Example Solving**

**Example:**

****

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $

S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, A) = closure (S → A•A, $ )

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $

A → •aA, $

A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

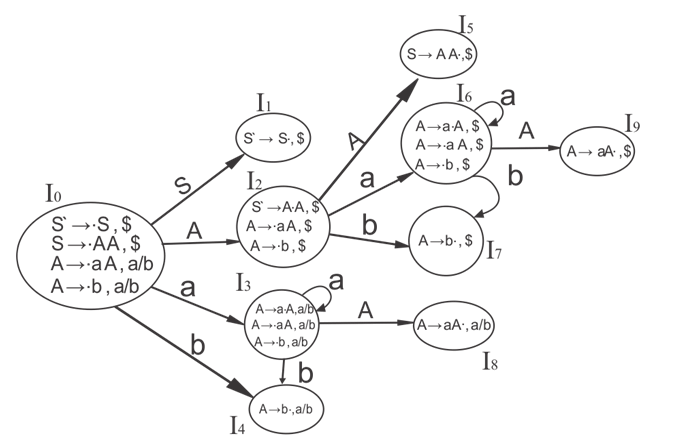
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

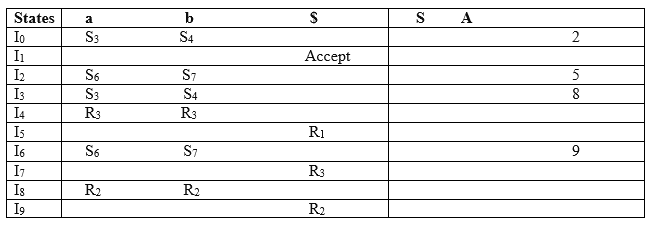
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:



CLR (1) Parsing Table



Productions are numbered as follows:

S → AA ... (1)

A → aA ... (2)

A → b ... (3)

The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.

I4 contains the final item which drives (A → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.

I5 contains the final item which drives (S → AA•, $), so action {I5, $} = R1.

I7 contains the final item which drives (A → b•, $), so action {I7, $} = R3.

I8 contains the final item which drives (A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.

I9 contains the final item which drives (A → aA•, $), so action {I9, $} = R2.

**Chapter 3**

**Literature survey**

**3.1 Existing System**

The CLR parser is an important parsing technique used in compiler design, which is capable of generating a parser that can parse any context-free grammar. CLR stands for Canonical LR, and it is an extension of the LR (1) parser. The CLR parser is a bottom-up parsing technique that uses a canonical collection of LR (1) items, which includes all the LR (1) items that can be derived from a given grammar. The parsing algorithm of CLR parser constructs a parse tree from the bottom up, starting from the input symbols, and following the rules of the grammar.

A literature survey on CLR parser reveals that there are several resources available that provide a comprehensive introduction to the theory and practical implementation of CLR parser. These resources include books, research papers, and online resources, which cover the principles of parsing techniques, the theory of parsing, translation and compiling, and parsing algorithms. Moreover, CLR parser has been compared with other parsing techniques, such as LL and LR parsers, in terms of their performance and efficiency. It is generally considered that the CLR parser is more powerful than LL and LR parsers, and it is capable of parsing any context-free grammar.

In conclusion, the CLR parser is a powerful parsing technique used in compiler design, which provides a comprehensive solution for parsing context-free grammars. CLR parser offers an efficient and powerful parsing algorithm that can generate a parser capable of parsing any context-free grammar. The literature survey on CLR parser provides a good foundation for understanding the theory and practical implementation of CLR parser, which can be used to design and develop efficient and effective compilers

1. "Introduction to Compiler Design" by Torben Ægidius Mogensen: This book provides a comprehensive introduction to compiler design and includes a detailed explanation of CLR parser.

2. "Parsing Techniques - A Practical Guide" by Dick Grune and Ceriel J.H. Jacobs: This book covers the principles of parsing techniques, including bottom-up parsing techniques, such as CLR parser.

3. "Parsing Theory: Volume II LR(k) and LL(k) Parsing" by Seppo Sippu and Eljas Soisalon-Soininen: This book provides a theoretical foundation for LR(k) parsing techniques, including CLR parser.

4. "LR Parsing: Theory and Practice" by John E. Hopcroft and Jeffrey D. Ullman: This paper provides a detailed explanation of the CLR parser and its variants, as well as their practical implementation.

5. "Efficient Computation of Canonical LR(k) Items" by Frank DeRemer and Thomas Pennello: This paper introduces an algorithm for efficiently computing the canonical LR(k) items, which are the building blocks of the CLR parser.

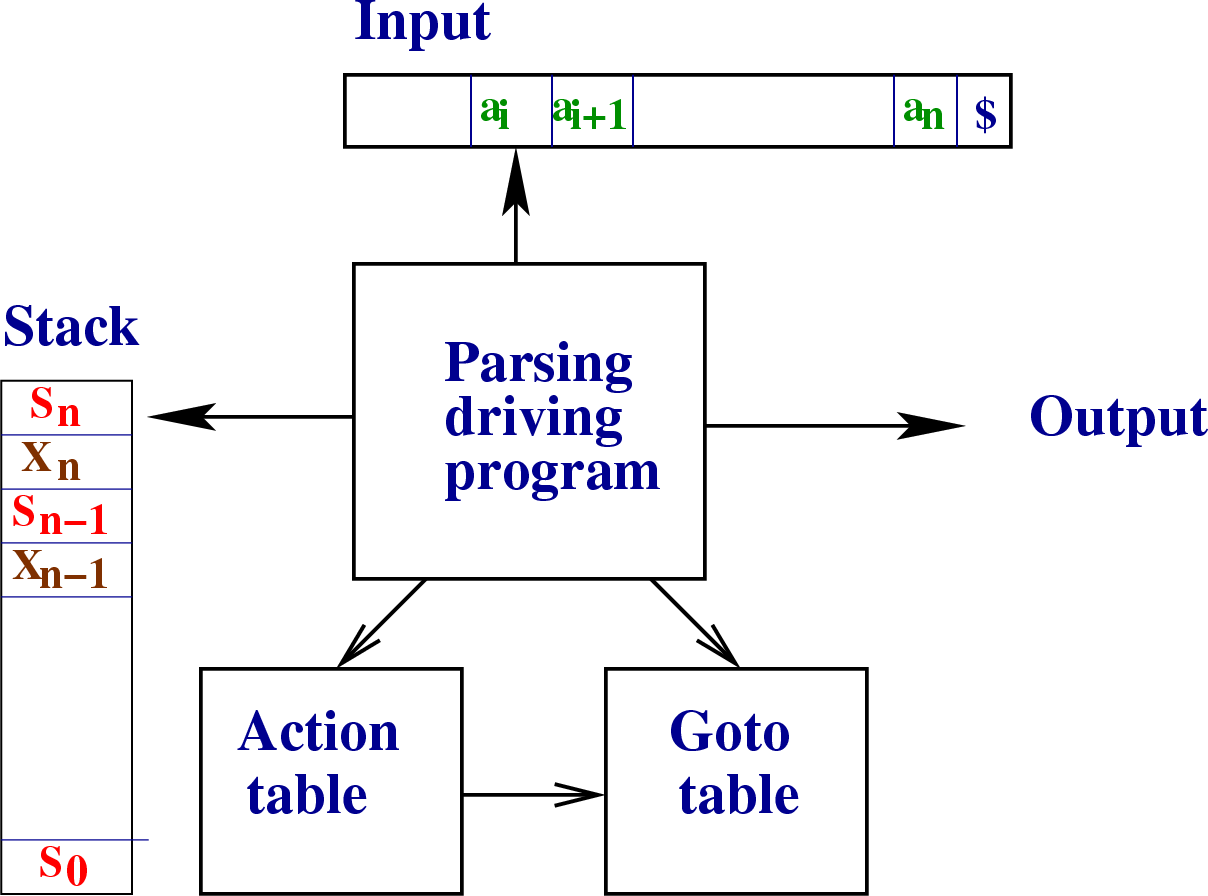
These resources provide a good foundation for understanding CLR parser and its implementation in practice. They also offer insights into the strengths and weaknesses of the CLR parser and how it compares to other parsing techniques.

## Chapter 4

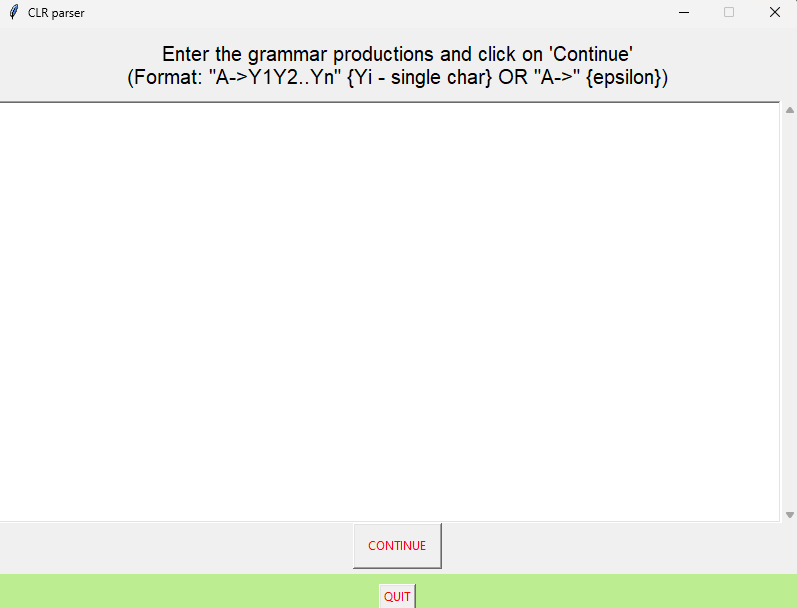
**System Architecture and Design**

## 4.1 Architecture Diagram

## 



**4.2 Interface Design**

****

## Chapter 5

## Coding and Testing

## 5.1 Code Implementation

## Clr\_gui.py

from tkinter import \*

from collections import deque, OrderedDict

from pprint import pprint

import firstfollow

from firstfollow import production\_list, nt\_list as ntl, t\_list as tl

nt\_list, t\_list = [], []

j = None

class Application(Frame):

    def \_\_init\_\_(self, master=None):

        Frame.\_\_init\_\_(self, master)

        self.master = master

        master.title('CLR parser')

        master.geometry("800x600")

        master.resizable(0, 0)

        self.pack()

        self.createWidgets(master)

    def center(self, toplevel):

        toplevel.update\_idletasks()

        w = toplevel.winfo\_screenwidth()

        h = toplevel.winfo\_screenheight()

        size = tuple(int(\_)

                     for \_ in toplevel.geometry().split('+')[0].split('x'))

        x = w/2 - size[0]/2

        y = h/2 - size[1]/2

        toplevel.geometry("%dx%d+%d+%d" % (size + (x, y)))

    def createWidgets(self, master):

        self.center(master)

        self.mframe = Frame(master)

        self.mframe.pack(padx=0, pady=0, ipadx=0, ipady=0)

        frame = Frame(self.mframe)

        frame.pack(side=TOP)

        frame2 = Frame(self.mframe)

        frame2.pack()

        bottomframe = Frame(self.mframe, bd=10, bg="#BCED91")

        bottomframe.pack(side=BOTTOM, fill=BOTH, pady=5)

        self.head = Label(frame, text='''Enter the grammar productions and click on 'Continue'

(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})''', font='Helvetica -20', fg="black")

        self.head.pack(padx=10, pady=10)

        self.make\_tb(frame)

        self.cont = Button(

            frame2, fg="red", text="CONTINUE", command=self.start)

        self.cont.pack(ipadx=10, ipady=10, expand=1, side=BOTTOM)

        Button(bottomframe, text="QUIT", fg="red", command=master.destroy).pack(

            fill=Y, expand=1, side=RIGHT)

    def start(self):

        pl = self.text.get("1.0", END).split("\n")+['']

        # print(pl)

        self.head.config(text="First and Follow of Non-Terminals")

        self.text.delete("1.0", END)

        self.master.geometry("800x600")

        self.cont.config(command=self.more)

        global nt\_list, t\_list

        firstfollow.production\_list = firstfollow.main(pl)

        for nt in ntl:

            firstfollow.compute\_first(nt)

            firstfollow.compute\_follow(nt)

            self.text.insert(END, nt)

            self.text.insert(END, "\tFirst:\t{}\n".format(

                firstfollow.get\_first(nt)))

            self.text.insert(END, "\tFollow:\t{}\n\n".format(

                firstfollow.get\_follow(nt)))

        # self.text.config(state=DISABLED)

        augment\_grammar()

        nt\_list = list(ntl.keys())

        t\_list = list(tl.keys()) + ['$']

        #self.text.insert(END, "{}\n".format(nt\_list))

        #self.text.insert(END, "{}\n".format(t\_list))

        self.text.see(END)

        self.text.config(state=DISABLED)

    def more(self):

        self.text.config(state=NORMAL)

        global j

        j = calc\_states()

        global nt\_list, t\_list

        self.head.config(text="Canonical LR(1) Items")

        self.text.delete("1.0", END)

        self.cont.config(command=self.more2)

        ctr = 0

        for s in j:

            self.text.insert(END, "\nI{}:\n".format(ctr))

            for i in s:

                self.text.insert(END, "\t{}\n".format(i))

            ctr += 1

        self.text.see(END)

        self.text.config(state=DISABLED)

    def more2(self):

        self.text.config(state=NORMAL)

        global j

        self.head.config(text="CLR(1) Table")

        self.text.delete("1.0", END)

        self.cont.destroy()

        table = make\_table(j)

        sr, rr = 0, 0

        self.text.config(font='-size 12', height=20)

        self.text.insert(END, "\t{}\t{}\n".format(

            '\t'.join(t\_list), '\t'.join(nt\_list)))

        for i, j in table.items():

            self.text.insert(END, "{}\t".format(i))

            for sym in t\_list+nt\_list:

                if sym in table[i].keys():

                    if type(table[i][sym]) != type(set()):

                        self.text.insert(END, "{}\t".format(table[i][sym]))

                    else:

                        self.text.insert(END, "{}\t".format(

                            ', '.join(table[i][sym])))

                else:

                    self.text.insert(END, "\t")

            self.text.insert(END, "\n")

            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

        self.text.insert(

            END, "\n\n{} s/r conflicts | {} r/r conflicts\n".format(sr, rr))

        self.text.see(END)

        self.text.config(state=DISABLED)

    def make\_tb(self, frame):

        self.text = Text(frame, wrap="word", height=13, bd=2, font='-size 20')

        self.vsb = Scrollbar(frame, orient="vertical", command=self.text.yview)

        #self.hsb=Scrollbar(frame, orient="horizontal", command=self.text.xview)

        self.text.configure(yscrollcommand=self.vsb.set)

        # self.text.configure(xscrollcommand=self.hsb.set)

        self.vsb.pack(side="right", fill="y")

        #self.hsb.pack(side="bottom", fill="x")

        self.text.pack(side="left", fill="both", expand=True)

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 firstfollow.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 = firstfollow.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 firstfollow.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 = firstfollow.production\_list[0]

            firstfollow.production\_list.insert(

                0, chr(i)+'->'+start\_prod.split('->')[0])

            return

def main():

    root = Tk()

    app = Application(master=root)

    app.mainloop()

    return

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Calculating 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

        if body[0]==symbol: 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

#   ...

            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 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)

            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\_backend.py

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("\n\tCLR(1) TABLE\n")

    sr, rr=0, 0

    for i, j in table.items():

        print(i, "\t", j)

        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("\n", sr, "s/r conflicts |", rr, "r/r conflicts")

    return

if \_\_name\_\_=="\_\_main\_\_":

    main()

Slr\_backend.py

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

def closure(items):

    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]

            for prod in production\_list:

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

                if head!=Y: continue

                newitem=Y+'->.'+body

                if newitem not in 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(head+'->'+seen+unseen[0]+'.'+unseen[1:])

    return closure(initial)

def calc\_states():

    def contains(states, t):

        for s in states:

            if sorted(s)==sorted(t): return True

        return False

    global production\_list, nt\_list, t\_list

    head, body=production\_list[0].split('->')

    states=[closure([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(closure):

        for s in states:

            if sorted(s.closure)==sorted(closure): return s.no

    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 t\_list:

                    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 ntl[head].follow: #for term in t\_list:

                        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()) + ['$']

    table=make\_table(calc\_states())

    print("\tSLR(1) TABLE\n")

    for i, j in table.items():

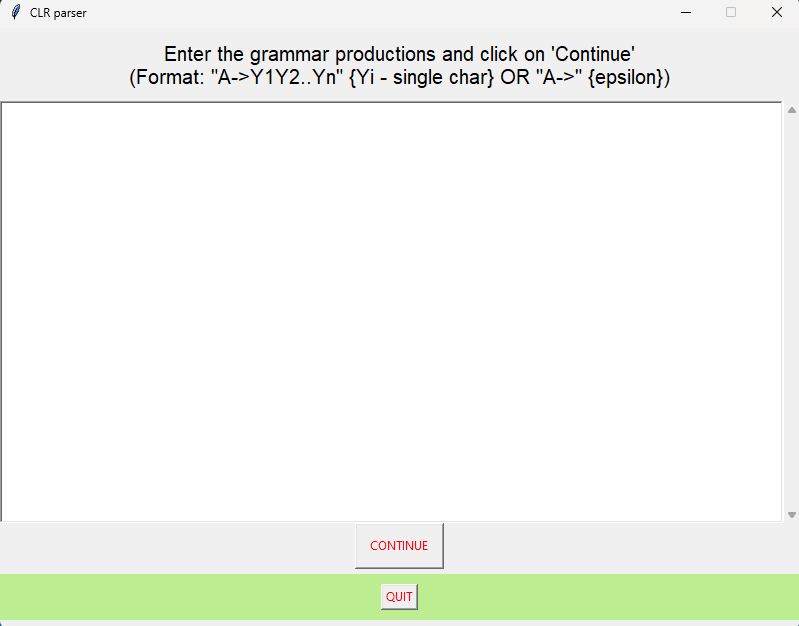
        print(i, "\t", j)

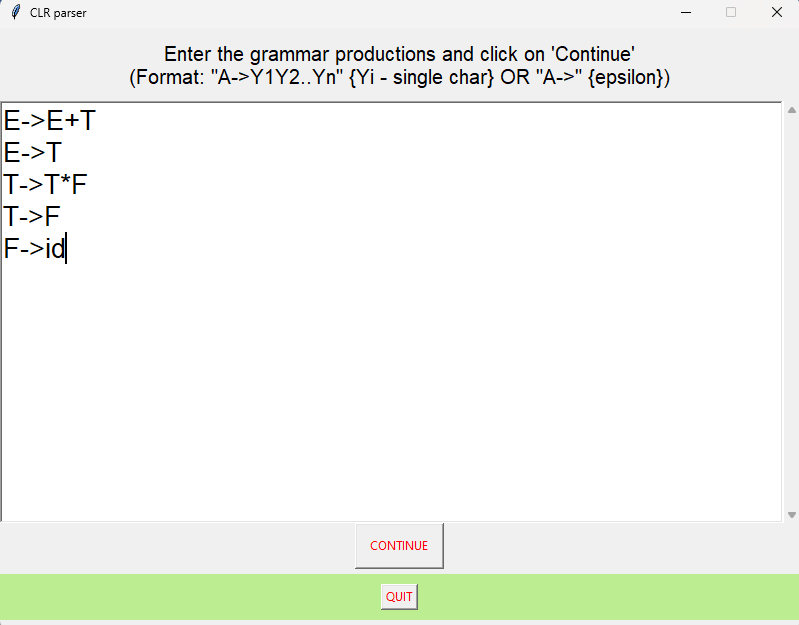
    return

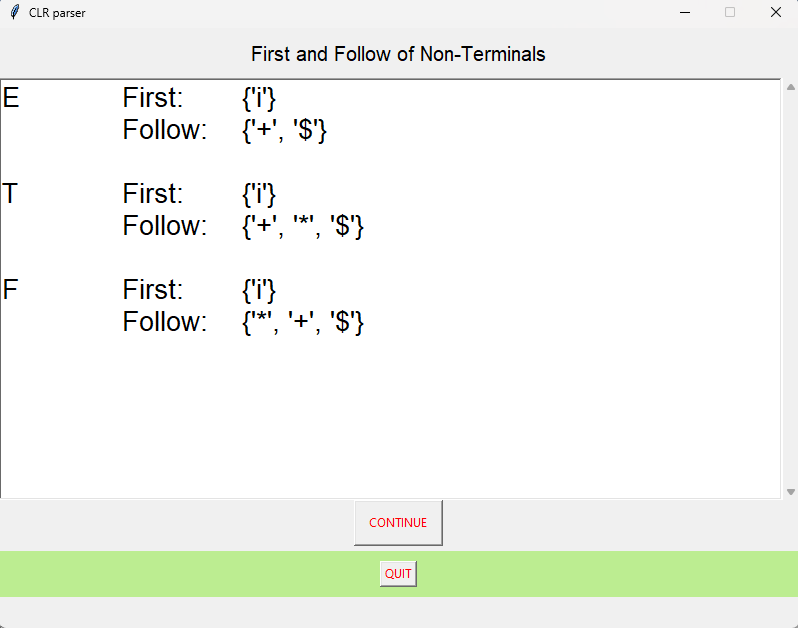
if \_\_name\_\_=="\_\_main\_\_":

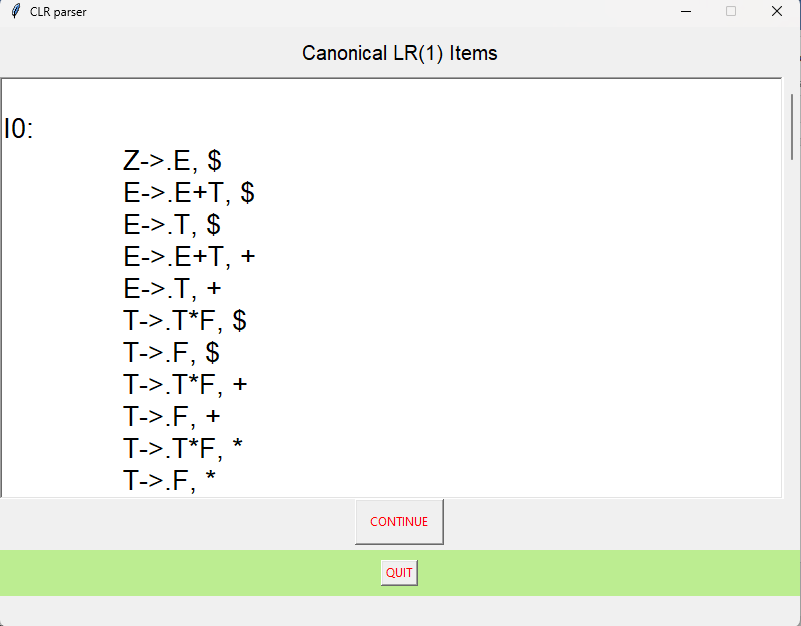
    main()

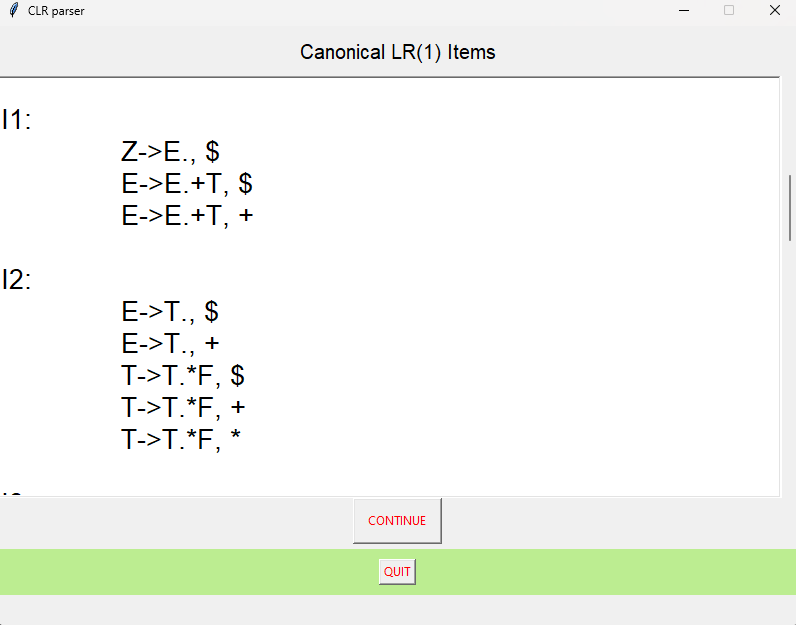
**5.2 Testing**

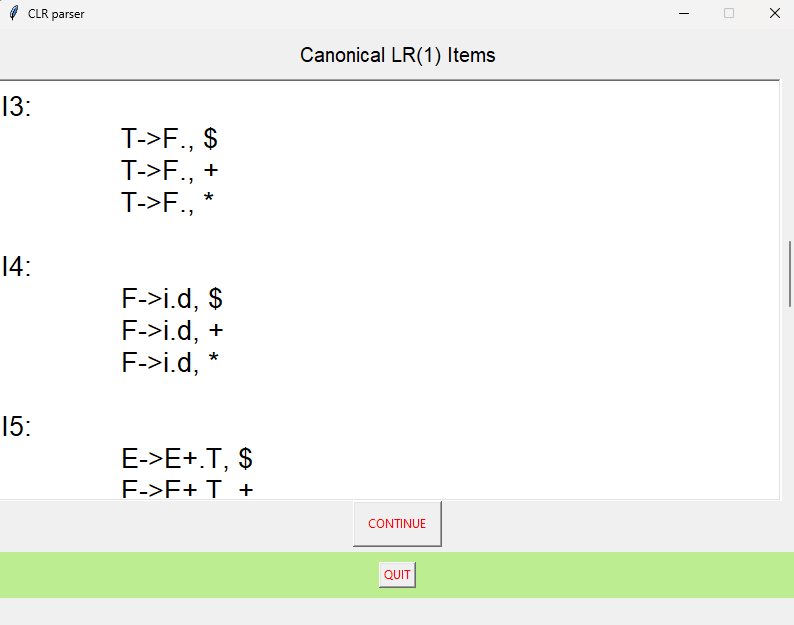


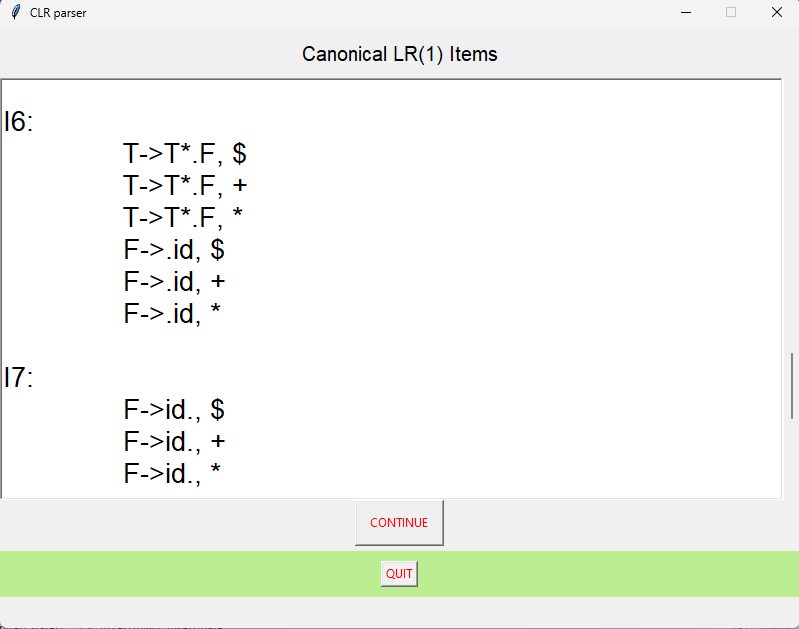


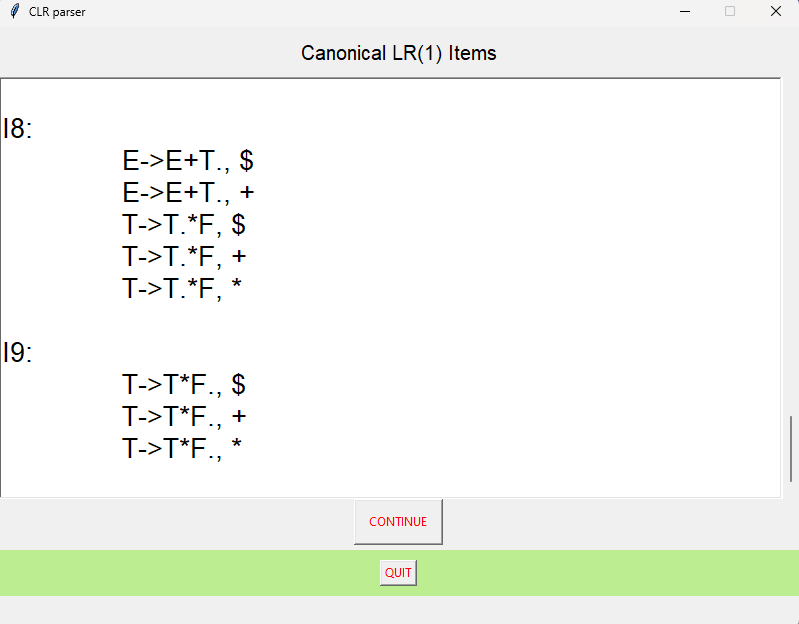


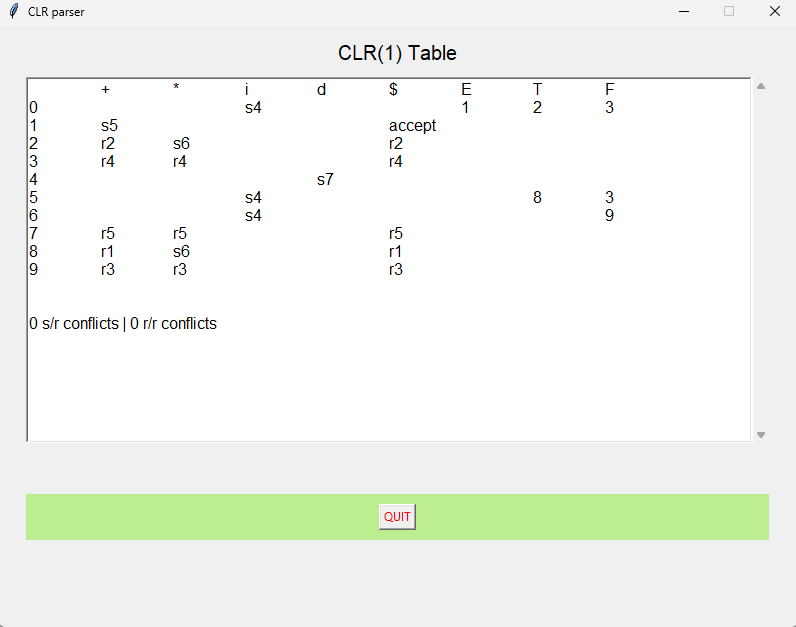












**Chapter 6**

**Conclusion and Future Enhancement**

**6.1 Future Enhancement**

For future enhancements of a CLR parser project, some areas of improvement include optimizing the parser's performance by reducing the time complexity of the parsing algorithm. This can be achieved by improving the efficiency of constructing the parse table, which is an essential step in the parsing process. Another area of improvement is enhancing the error reporting mechanism of the parser, which helps in identifying and diagnosing syntax errors in the input code. Finally, improving the parser's ability to handle ambiguous grammars can be another area of future enhancement, which can be achieved by incorporating techniques such as conflict resolution and syntax-directed translation.

**6.2 Conclusion**

In conclusion, the CLR parser is an important and powerful parsing technique used in compiler design, which provides a comprehensive solution for parsing context-free grammars. The literature survey reveals that there are several resources available that provide a good foundation for understanding the theory and practical implementation of CLR parser, which can be used to design and develop efficient and effective compilers.

Overall, a CLR parser project can be a useful and practical implementation of a powerful parsing technique that can help in designing and developing efficient compilers. With continuous development and improvement, a CLR parser project can be an essential tool in the field of compiler design and development.

**Chapter 7**

**References**

1. Aho, A. V., & Ullman, J. D. (1977). Principles of Compiler Design. Addison-Wesley..
2. Grune, D., & Jacobs, C. (2008). Parsing Techniques: A Practical Guide. Springer.
3. Rosu, G., & Sagiv, Y. (2009). An Introduction to Parsing Techniques. Foundations and Trends in Programming Languages, 1(4), 277-416.
4. Scott, E., & Johnstone, A. (2010). GLL Parsing. ACM Transactions on Programming Languages and Systems, 32(4), 1-44.
5. Fischer, C. N., & LeBlanc, R. J. (2011). Crafting a Compiler. Prentice Hall.