CLR Parser with Tkinter

Team members

- Akshitha Komatireddy(RA1911003010**134**)
- Chinnam Lakshmi Durga (RA1911003010**127**)
- Abhishek Kumar(RA1911003010**143**)

Abstract of the project

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's 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.
Our project is to create this parser using python and parse grammar into it to get desired results.

Objective of this project

The objective of this project is to create a CLR Parser that can be used to generate LR(0) and LR(1) Items as well as create the CLR Parsing table that can be used to parse the string and check if that string is valid for any given grammar.
The objective is for the user to give a production as input and the program generates the items, as well as the table all, presented neatly in a frontend program.

Working:

CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces more number of states as compared to the SLR (1) parsing. In the CLR (1), we place the reduced 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 a Canonical collection of LR (0) items

- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table
- 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 lookahead is used to determine where we place the final item.
- The look-ahead always adds a $ symbol for the argument production.

CODE:

```
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:
```

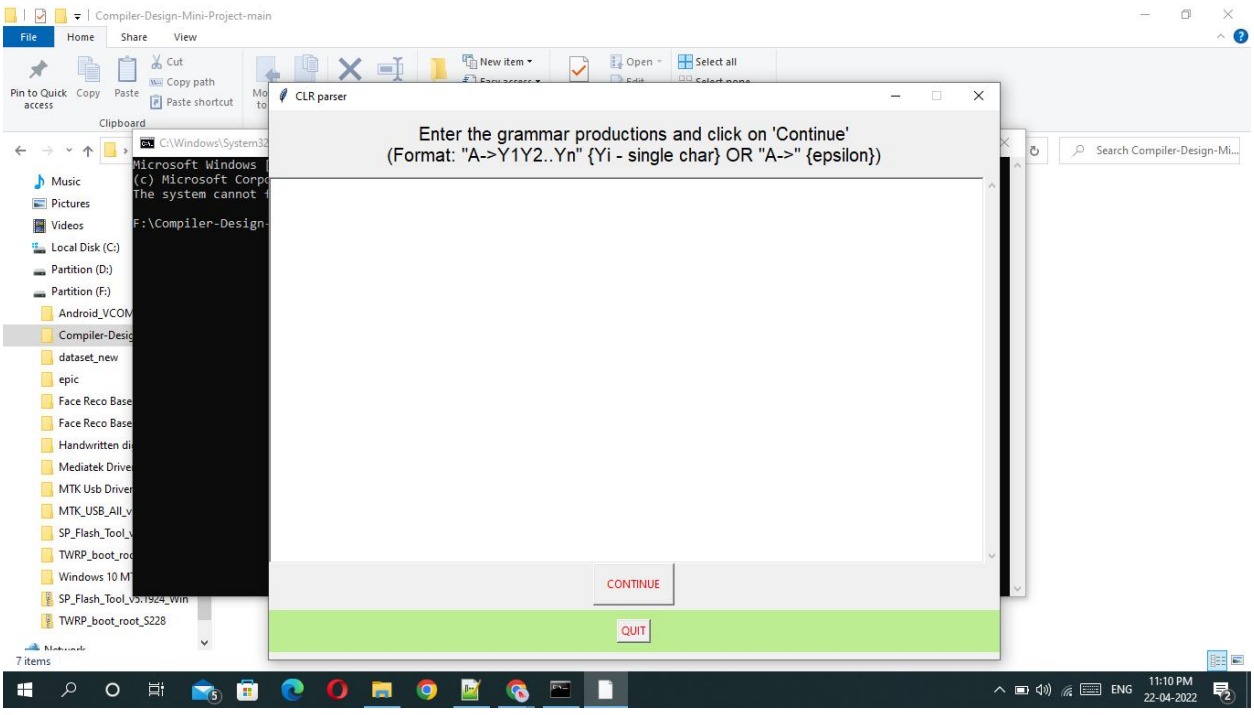
```
                        t=goto(s, e)
                        if t == [] or contains(states, t): continue

                        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):
```

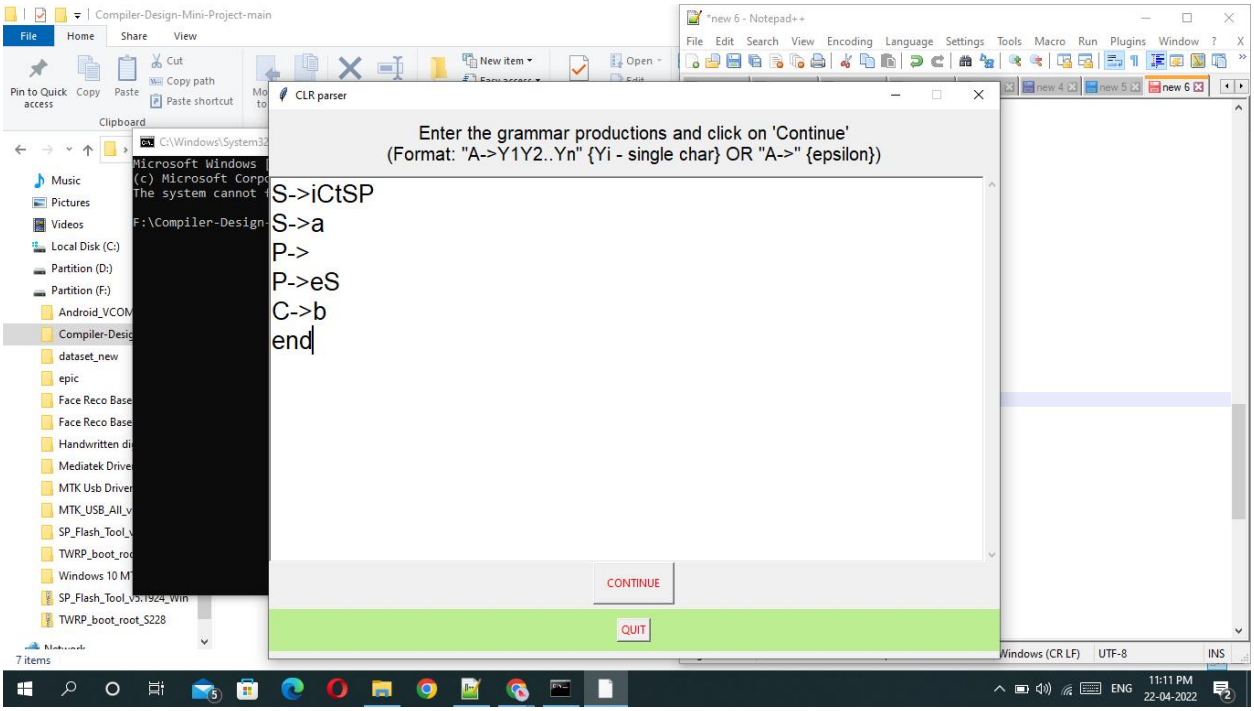
```
            closure=''.join(closure).replace('.', '')
            return firstfollow.production_list.index(closure)

        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():
```

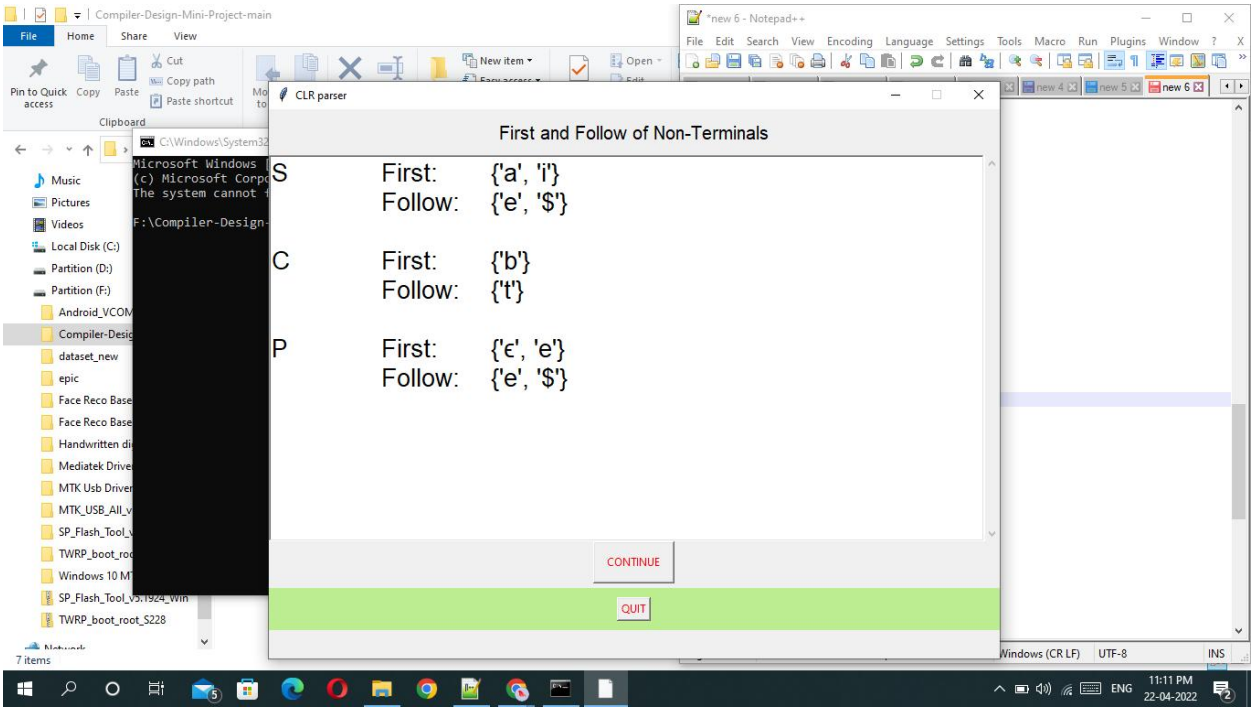
```
        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                                        else: SLR_Table[s.no][term] |=
{'r'+str(getprodno(item))}
                                continue

                        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():
```

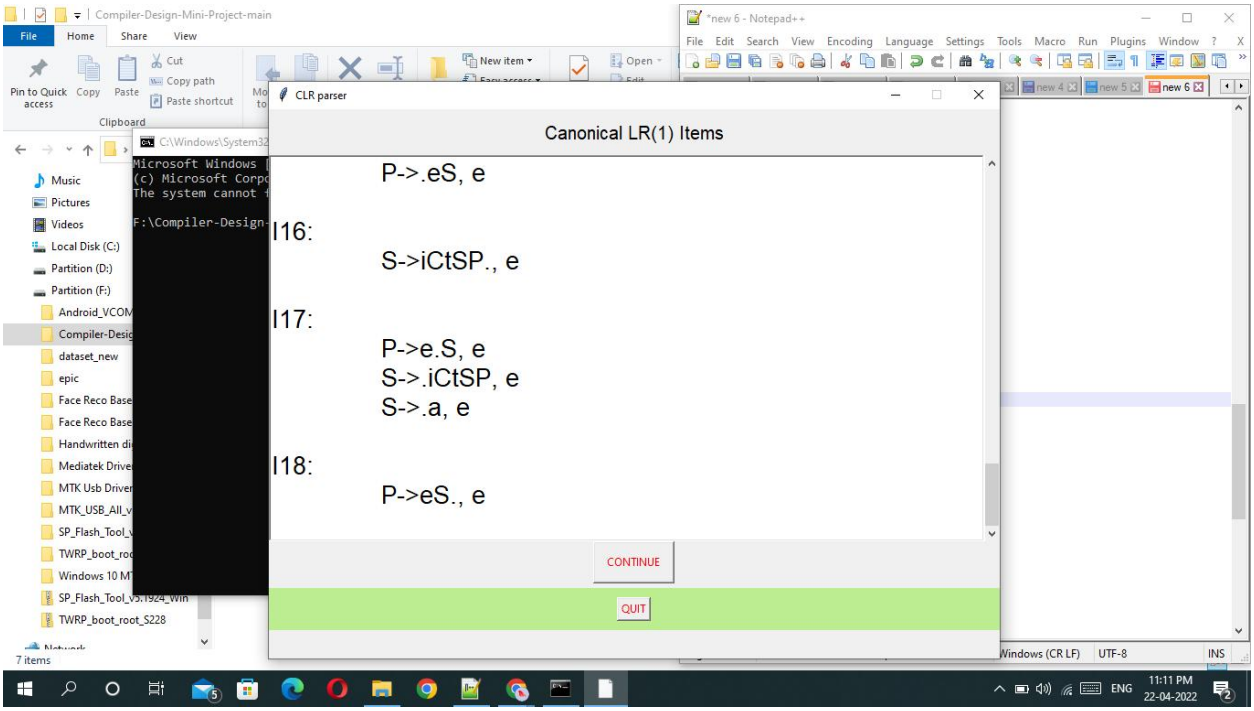
```
        root=Tk()
        app=Application(master=root)
        app.mainloop()

        return

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

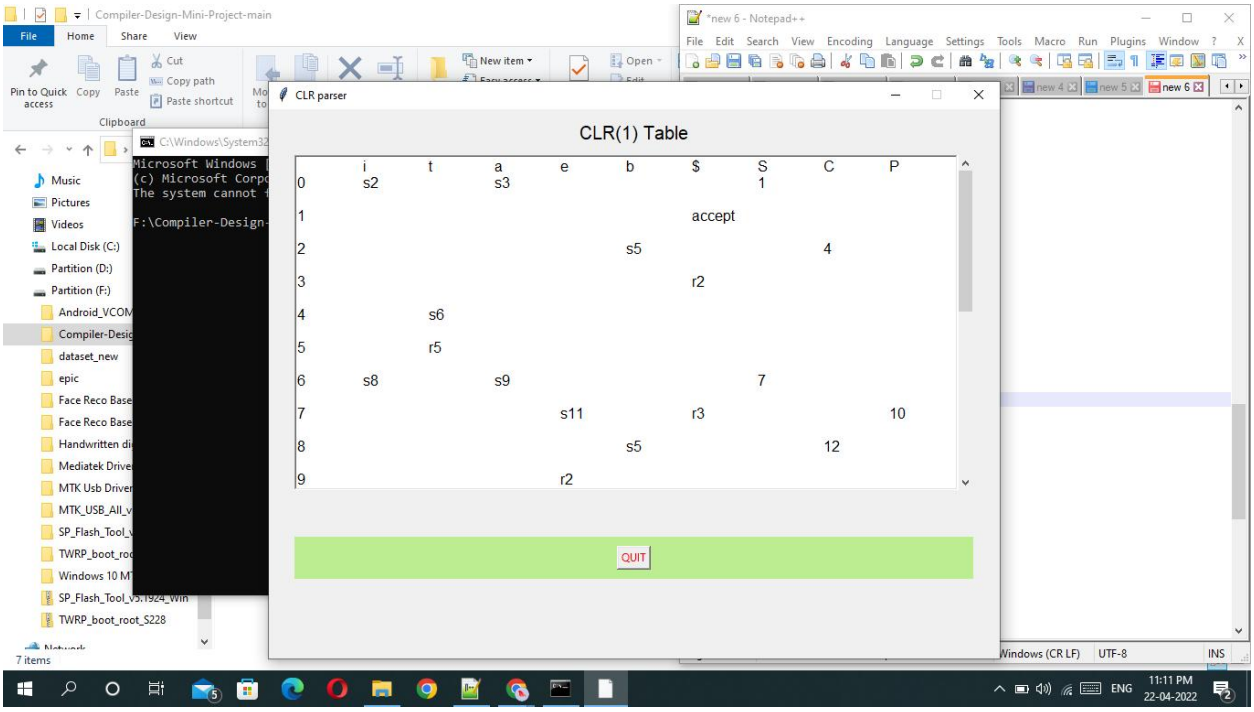
Output:




CLR parser
Enter the grammar productions and click on 'Continue'
(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})
S->iCtSP
S->a
P->
P->eS
C->b
end
CONTINUE
QUIT


CLR parser
First and Follow of Non-Terminals
S First: {'a', 'i'}
Follow: {'e', '$'}
C First: {'b'}
Follow: {'t'}
P First: {'ϵ', 'e'}
Follow: {'e', '$'}
CONTINUE
QUIT





Result:

CLR parser with Tkinter is implemented successfully.