【Experiment name】

Automatically generate LR(0) analysis table

【Purpose】

Input: Arbitrary compressed context-free grammar.
Output: Corresponding LR (0) analysis table

【Experimental principle】

For LR grammars, we can automatically construct corresponding LR analysis tables. In order to construct the LR parsing table, we need to define an important concept - the normalized sentence pattern "living prefix" of the grammar. A prefix without any symbols after such a handle is called a live prefix. At any time during the LR analysis process, the grammatical symbols in the stack (from the bottom up) $X_1X_2 \ldots X_m$ should constitute a live prefix, and after matching the rest of the input string, it should become a canonical sentence pattern (if the entire input string does constitute a sentence). Therefore, as long as the scanned part of the input string remains reducible to a live prefix, it means that the scanned part is error-free.

For a grammar G , we can construct a finite automaton that can recognize all live prefixes of
G , and then convert this automaton into an LR analysis table, and perform LR analysis according to the LR analysis table to ensure that the During the process, if the analyzed sentence is correct, the grammar symbols in the stack (from the bottom of the stack) always constitute a live prefix. If an extension grammar G of a grammar G' Each state (itemset) in the live prefix recognition automaton does not have the following conditions: (1) contains both shifted items and reduced items; (2) contains multiple reduced items, then G is said to be a LR (0) grammar. The state set of the automaton is the LR (0) item set specification family of the grammar.

There are three ways to construct a DFA that recognizes grammar or prefixes :

(1) Calculate the regular expression of live prefix according to the formal definition, then construct NFA from this regular expression and then determine it as DFA ;

(2) Find all the items of the grammar, construct the NFA that recognizes the live prefix according to certain rules , and then determine it into DFA ;

(3) Use the closure function (CLOSURE) and the steering function (GO(I,X)) to construct the item set specification family of the LR(0) of the grammar G' , and then use the conversion function to establish the connection relationship between the states to be identified DFA of live prefixes

.

The prefix of a symbol string refers to any header of the symbol string, including the empty string ε. For example, for the symbol string abc, its prefixes are ε, a, ab, abc. A live prefix of a canonical sentence is a prefix of the sentence if the input string is correct, but it does not contain any symbols after the handle. The reason why it is called a live prefix is that a standard sentence pattern can be formed by connecting uninput symbol strings after the prefix.

The relationship between live prefixes and handles is as follows:

(1) The active prefix already contains all the symbols of the handle, indicating that the right part β of the production A→β has appeared on the top of the stack.

(2) The live prefix only contains part of the handle symbols, indicating that the right substring β_1 of A→$\beta_1\beta_2$ has appeared on the top of the stack, and we expect to see the symbols deduced by β_2 from the input string .

(3) The live prefix does not contain any symbols of the handle, and the string of symbols deduced from the right part of A→β is expected at this time.

Adding a dot anywhere on the right side (candidates) of each production in grammar G constitutes each production called an LR(0) item. Such as production A→xyz has items like:

A→.xyz, A→x.yz, A→xy.z, A→xyz.. How much of the right hand symbol has been recognized (appears on the top of the stack) for each production of the grammar during parsing can be determined with such dotted productions.

(1) A→β. The right part β of the characterizing production A→β has appeared on the top of the stack.

(2) A→β_1.β_2 characterizes A→$\beta_1\beta_2$'s right substring β_1 has appeared on the top of the stack, expecting to see the symbol deduced by β_2 from the input string .

(3) A→.β engraves any symbol without a handle on the top of the stack, and expects the symbol string deduced from the right part of A→β.

(4) For the LR(0) item of A→ε, there is only A→. . Suppose the grammar G=(V_T, V_N, S, P) is a context-free grammar, if there is a canonical derivation S $_{*rm}$⇒a Aw $_{rm}$⇒ab 1 b 2w (where A ®b 1 b 2 Î P), then it is called Item A ®b 1 • b 2 is valid for live prefix g = ab 1 , ie LR(0) valid item.

In an intuitive sense, an LR(0) item indicates how much of the production we see recognized at a certain step in the analysis, and the dots in the LR(0) item can be seen as the relationship between the top of the analysis stack and The dividing line of the input string, the left side of the dot is the part that has entered the analysis stack, and the right side is the symbol string that is currently input or continues to scan.

Different LR(0) items reflect different situations at the top of the analysis stack. We divide LR(0) items into four categories according to their different functions:

(1) Reduction items:

Expression form: A→a. This type of LR(0) item indicates that the handle a happens to be included in the stack, that is, part of the contents of the current top of the stack constitutes the expected handle, and should be reduced according to

A→a. (2) Accept items:

Form of expression: $S\rightarrow$ a. Among them, S is the only start symbol of the grammar. This type of LR(0) item is actually a special reduction item, which means that the content in the analysis stack is exactly a, and if the reduction is performed with $S\rightarrow$ a, the entire analysis is successful.

(3) Move into item:

Form of expression: $A\rightarrow \alpha.b\beta$ ($b \in V_T$)

This type of LR(0) item indicates that the analysis stack is a live prefix that does not completely contain the handle. In order to form a live prefix that happens to have a handle, b needs to be moved into the analysis stack.

(4) To-be-appointed projects:

Form of expression: $A\rightarrow\alpha.B\beta$ ($B \in V_N$)

This type of LR(0) item indicates that the analysis stack is a live prefix that does not completely contain a handle. In order to form a live prefix that just has a handle, the corresponding content in the current input string should be reduced to B first.

After giving the definition and classification of LR(0) items, we start from these LR(0) items to construct a finite automaton that can recognize all prefixes of a grammar. The steps are as follows: first construct a non-deterministic finite automaton that can recognize all live prefixes of the grammar, then determinize and minimize it, and finally obtain the required deterministic finite automaton. A method of constructing a nondeterministic finite automaton that recognizes all live prefixes of a grammar G from the LR(0) terms of a grammar G:

(1) It is stipulated that the first LR(0) item (ie $S'\rightarrow .A$) of the production formula (suppose $S'\rightarrow$ A) containing the grammar start symbol is the only initial state of NFA.

（2）Let all LR(0) items correspond to a state of NFA and the corresponding state in which LR(0) items are reduction items is the final state.

（3）If state i and state j come from the production of the same grammar G and the dots of the two state LR(0) items only differ by one position, that is: if i is $X \to X_1 X_2 \cdot \ldots X_{i-1} \cdot X_i \ldots X_n$, j is $X \to X_1 x2 \ldots X_i \cdot X_{i+1} \ldots X_n$, then draw an arc labeled Xi from state i to state j.

（4）If state i is an item to be reserved (let $X \to \alpha \cdot A\beta$), then lead ε-arc from state i to all states of $A \to \cdot r$.

In order to make the "accepting" state easy to identify, we usually extend the grammar G. Assuming that grammar G is a grammar starting with S as a symbol, we construct a *G′*, which contains the whole of G, but it introduces a nonterminal *S that does not appear in* 'G, and add a new production *S*′→ S, with *S*′→ S *G′* is the start symbol. Then, we call *G′* is an extended grammar of G.

Thus, there will be a state containing only the item *S*′→ S, which is the only "accepting" state.

If I is an itemset of grammar G', the closure CLOSURE(I) of I is defined and constructed as follows:

（1）The items of I are all in CLOSURE(I).

（2）If $A \to \alpha.B\ \beta$belongs to CLOSURE(I), then each item of the form $B \to .\gamma$also belongs to CLOSURE(I).

（3）Repeat（2）until CLOSURE(I) no longer expands.

Define the conversion function as follows:

GO（I, X）=CLOSURE（J）

where: I is the state containing a certain item set, X is a grammar symbol, $J=\{ A \to \alpha X .\beta \mid A \to \alpha.X\ \beta \in I\}$.

The leftmost item whose dot is not in the right part of the production is called the kernel, the only exception is S′→.S, so J obtained by using the GOTO（I, X）state transition function is the kernel of the state closure item set after turning.

Use the closure function（CLOSURE）and the conversion function (GO(I,X)) to construct the item set specification family of LR(0) of the grammar G', the steps are as follows:

（1）Set the item S′→.S as the core of the initial state set, and then find the closure CLOSURE（{S′→.S}）for the core to obtain the initial state closure item set.

（2）Apply the conversion function GO(I, X)=CLOSURE(J) to the initial state set or other constructed itemsets to obtain the closure itemsets of the new state J.

（3）Repeat（2）until no new itemsets appear.

The algorithm pseudocode for computing LR（0）itemset specification family $C=\{I_0, I_1, \ldots In\}$ is as follows:

Procedure itemsets(G');

```
Begin C := { CLOSURE ({S' →.S})}
Repeat
For each item set I and each grammar symbol X in C
Do if GO(I,X) is not empty and does not belong to C
Then put GO(I,X) in C
Until C no longer grows
End;
```

An itemset may contain many kinds of items. If the shift and reduce items exist at the same time, it is called a shift - reduce conflict. If the reduce and reduce items exist at the same time, it is called a reduce - reduce conflict. Let's look at a specific example:

build an LR parser based on the DFA that recognizes the live prefix of the grammar . Therefore, we need to study the different roles of the items in each item set (state) of this DFA .

We say that the item A→β1.β2 is valid for the live prefix αβ1, provided that there is a canonical derivation S A 1 2 . In general, the same item may be valid for several live prefixes (this is the case when an item appears in several different collections). If the reduction item A→β1. is valid for the live prefix 1, it tells us that the symbol string 1 should be reduced to A, that is, the live prefix 1 becomes αA.

If the move-in item A→β1.β2 is valid for the live prefix 1, then it tells us that the handle has not been formed yet, so the next action should be to move in. However, there may be situations where several entries are valid for the same live prefix. And they tell us what to do are different and conflicting. This conflict may be resolved by looking ahead a few more input symbols.

For each live prefix, we can construct its valid itemsets. In fact, the effective itemset of a live prefix γ is exactly the itemset (state) that is reached after reading γ starting from the initial state of the above-mentioned DFA. In other words, at any time, the active item set of the live prefix X1X2...Xm in the analysis stack is exactly the set represented by the state Sm at the top of the stack. This is a basic theorem of LR analysis theory. In fact, the item set (state) at the top of the stack embodies all useful information in the stack - history.

We have defined the LR(0) grammar before, let's take a look at the LR(0) analysis

How the table is structured.

For an LR(0) grammar, we can directly identify from its itemset specification family C and the live prefix from

The state transition function GO of motivation constructs LR analysis table. Below is the algorithm for constructing the LR(0) analysis table.

Assuming C={I0, I1,...,In}, let the subscript k of each item set Ik be a state of the analyzer, therefore, the LR(0) analysis table of G' contains states 0, 1,..., n . Let that contain item S'→. The subscript k of Ik of S is the initial state. The ACTION subtable and GOTO subtable can be constructed as follows:

(1) If the item A→α.aβ belongs to Ik and GO (Ik, a)= Ij, a is a terminal symbol, then set ACTION[k,

a] means "move state j and symbol a onto the stack", abbreviated as "sj";

(2) If item A→α. belongs to Ik, then, for any terminal symbol a, set ACTION[k, a] as "reduce by production A→α", abbreviated as "rj"; among them, assume that A→α is the jth of grammar G' a production;

(3) If item S'→S. If it belongs to Ik, then set ACTION[k, #] as "accept" and abbreviate as " acc";

(4) If GO (Ik, A)= Ij, A is a non-terminal, set GOTO[k, A]=j;

(5) All the blank cells in the analysis form that cannot be filled with information from the above 1 to 4 shall be marked with "error marks". The analysis table constructed according to the above algorithm contains two parts of ACTION and GOTO. If each entry does not contain multiple definitions, it is called an LR(0) analysis table of grammar G. A grammar G with an LR(0) table is called an LR(0) grammar, and an LR(0) grammar is unambiguous.

For example, the extended grammar of grammar G(E) is as follows:

(0)S'→E
(1)E→aA
(2)E→bB
(3)A→cA
(4)A→d
(5) B→cB
(6)B→d

The LR(0) analysis table of this grammar is shown in the table.

【Experimental content】

The main idea of this experiment is to

①Input grammar, identify Vn , Vt

② Construction project set

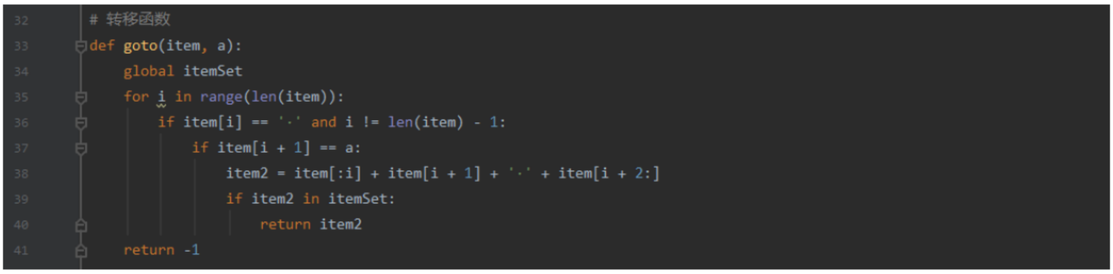
③ Select the initial core

④ Seek closure and transfer

⑥Generate LR(0) table according to DFA

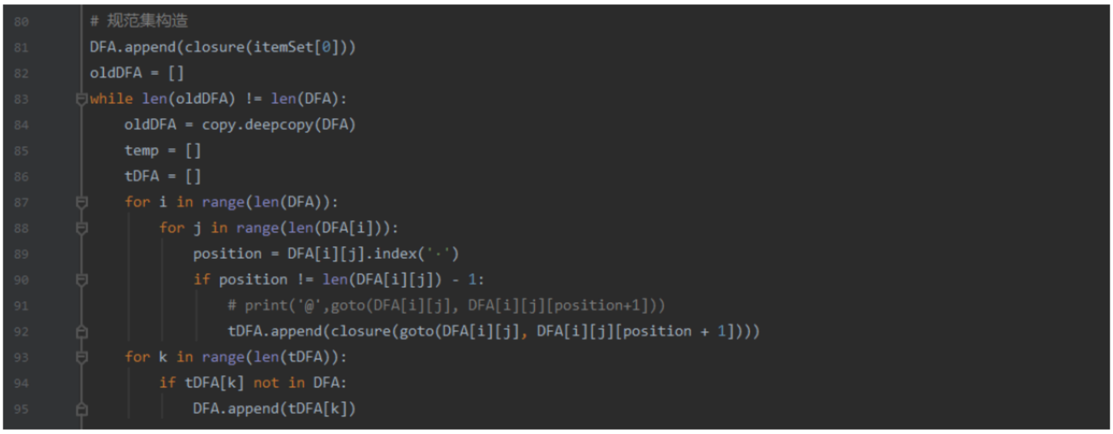
The main parts of the experimental code are as follows:

transfer function:



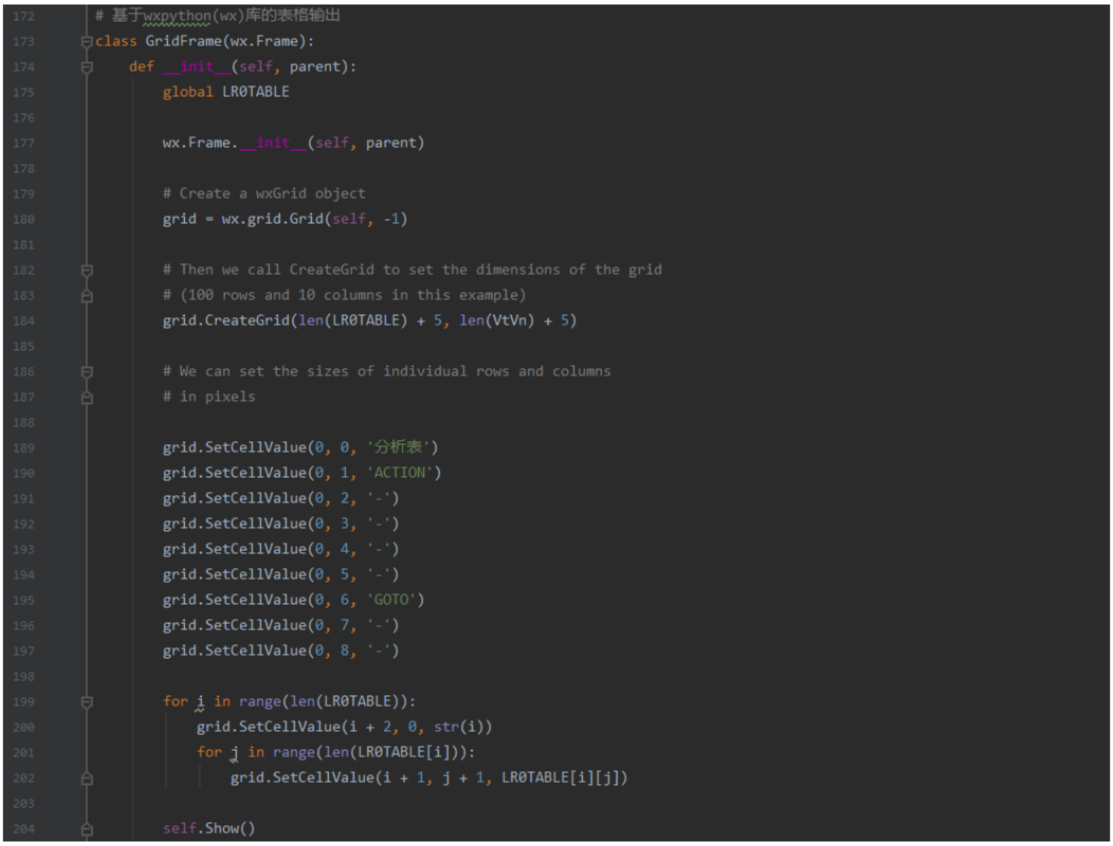
```python
# 转移函数
def goto(item, a):
    global itemSet
    for i in range(len(item)):
        if item[i] == '·' and i != len(item) - 1:
            if item[i + 1] == a:
                item2 = item[:i] + item[i + 1] + '·' + item[i + 2:]
                if item2 in itemSet:
                    return item2
    return -1
```

Specification set constructor:



```python
# 规范集构造
DFA.append(closure(itemSet[0]))
oldDFA = []
while len(oldDFA) != len(DFA):
    oldDFA = copy.deepcopy(DFA)
    temp = []
    tDFA = []
    for i in range(len(DFA)):
        for j in range(len(DFA[i])):
            position = DFA[i][j].index('·')
            if position != len(DFA[i][j]) - 1:
                # print('@',goto(DFA[i][j], DFA[i][j][position+1]))
                tDFA.append(closure(goto(DFA[i][j], DFA[i][j][position + 1])))
    for k in range(len(tDFA)):
        if tDFA[k] not in DFA:
            DFA.append(tDFA[k])
```

based on the w xpython (wx) library:



```python
# 基于wxpython(wx)库的表格输出
class GridFrame(wx.Frame):
    def __init__(self, parent):
        global LR0TABLE

        wx.Frame.__init__(self, parent)

        # Create a wxGrid object
        grid = wx.grid.Grid(self, -1)

        # Then we call CreateGrid to set the dimensions of the grid
        # (100 rows and 10 columns in this example)
        grid.CreateGrid(len(LR0TABLE) + 5, len(VtVn) + 5)

        # We can set the sizes of individual rows and columns
        # in pixels

        grid.SetCellValue(0, 0, '分析表')
        grid.SetCellValue(0, 1, 'ACTION')
        grid.SetCellValue(0, 2, '-')
        grid.SetCellValue(0, 3, '-')
        grid.SetCellValue(0, 4, '-')
        grid.SetCellValue(0, 5, '-')
        grid.SetCellValue(0, 6, 'GOTO')
        grid.SetCellValue(0, 7, '-')
        grid.SetCellValue(0, 8, '-')

        for i in range(len(LR0TABLE)):
            grid.SetCellValue(i + 2, 0, str(i))
            for j in range(len(LR0TABLE[i])):
                grid.SetCellValue(i + 1, j + 1, LR0TABLE[i][j])

        self.Show()
```

code:

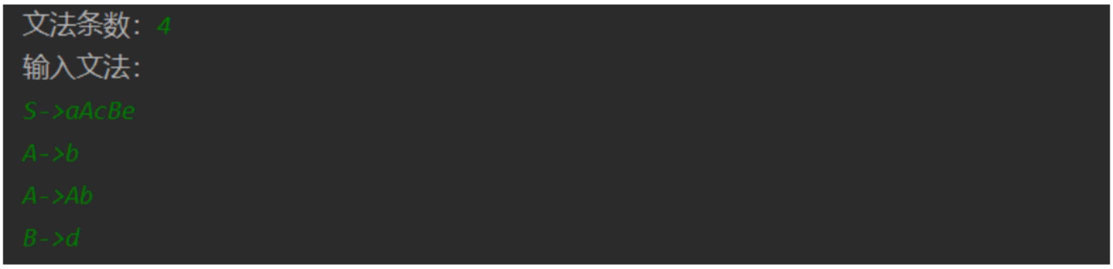
```
import copy
import wx
import wx. grid

grammar = [] # Save each grammar
itemSet = [] #
DFA = []
Vn = []
Vt = []


# closure 闭包
def closure(item):
    global itemSet
    dot = []
    dot.append(item)
    olddot = []
    while len(dot) != len(olddot):
        olddot = copy.deepcopy(dot)
        temp = []
        for i in range(len(dot)):
            for j in range(len(itemSet)):
                if dot[i].index('·') + 1 < len(dot[i]) and dot[i][dot[i].index('·') + 1] == itemSet[j][0] and \
                        itemSet[j][itemSet[j].index('>') + 1] == '·':
                    temp.append(itemSet[j])
        for k in range(len(temp)):
            if temp[k] not in dot:
                dot.append(temp[k])
    return dot


#  转移函数
def goto(item, a):
    global itemSet
    for i in range(len(item)):
        if item[i] == '·' and i != len(item) - 1:
            if item[i + 1] == a:
                item2 = item[:i] + item[i + 1] + '·' + item[i + 2:]
                if item2 in itemSet:
                    return item2
    return -1


def findItem(item):
    global DFA
    for i in range(len(DFA)):
        if item in DFA[i]:
            return i
    return -1


n = int(input('文法条数：'))
print("输入文法：")
for i in range(n):
    temp = input()
    if i == 0:
        grammar.append('S\'->' + temp[0])
    grammar.append(temp)
    for j in range(len(temp)):
        if temp[j].isupper() and temp[j] not in Vn:
            Vn.append(temp[j])
        elif temp[j].islower() and temp[j] not in Vt:
            Vt.append(temp[j])
Vn.sort()
Vt.sort()

for i in range(len(grammar)):
    flag = 0
    for j in range(len(grammar[i])):
        if grammar[i][j] == '>':
            flag = 1
        if flag == 1 and grammar[i][j] != '>':
            temp = grammar[i][:j] + '·' + grammar[i][j:]
            itemSet.append(temp)
    itemSet.append(grammar[i] + '·')

print("Grammar:")
print(grammar)

# 规范集构造
DFA.append(closure(itemSet[0]))
oldDFA = []
while len(oldDFA) != len(DFA):
    oldDFA = copy.deepcopy(DFA)
    temp = []
    tDFA = []
    for i in range(len(DFA)):
        for j in range(len(DFA[i])):
            position = DFA[i][j].index('·')
            if position != len(DFA[i][j]) - 1:
                # print('@',goto(DFA[i][j], DFA[i][j][position+1]))
                tDFA.append(closure(goto(DFA[i][j], DFA[i][j][position + 1])))
    for k in range(len(tDFA)):
        if tDFA[k] not in DFA:
            DFA.append(tDFA[k])

print("DFA:")
print(DFA)

for i in range(len(DFA)):
    for j in range(len(DFA[i])):
        if len(DFA[i][j][-1]) == '·' and len(DFA[i]) != 1:
            print('非 LR(0)文法')
            break

# print("DFA-length:")
# print(len(DFA))
DFAtable = []

for i in range(len(DFA)):
    table = []
    for j in range(len(DFA[i])):
        position = DFA[i][j].index('·')
        if position == len(DFA[i][j]) - 1:
            temp = DFA[i][j][:-1]

            table = [grammar.index(temp)] * (len(Vt) + 1)
            break
        for k in range(len(Vt)):
            if DFA[i][j][position + 1] == Vt[k]:
                temp = Vt[k] + 'S' + str(findItem(goto(DFA[i][j], Vt[k])))
                table.append(temp)
        for m in range(len(Vn)):
            if DFA[i][j][position + 1] == Vn[m]:
                temp = Vn[m] + str(findItem(goto(DFA[i][j], Vn[m])))
table.append(temp)
DFAtable.append(table)

# Determine whether it is an LR(0) grammar
flag = 0
for i in range(len(DFAtable)):
for j in range(len(DFAtable)):
if len(DFA[i][j]) > 2:
print('non-LR(0) grammar')
flag = 1
break
if flag == 1:
break
```

```
# print("DFAtable:")
# print(DFAtable)

VtVn = Vt + ['#'] + Vn
LR0TABLE = [[' ' for col in range(len(VtVn))] for row in range(len(DFA) + 1)]

# flat table output
print('------------------------------------------------')
print('分析表\t\t\t\tAction\t\t\t\t GOTO')
#  状态列
for i in range(len(VtVn)):
    LR0TABLE[0][i] = VtVn[i] + ' '
#  输出 DFAtable
for i in range(len(DFAtable)):
    if 0 in DFAtable[i]:
        LR0TABLE[2][VtVn.index('#')] = 'acc'     #  判断 acc
        continue
    for j in range(len(DFAtable[i])):
        try:
            LR0TABLE[i + 1][VtVn.index(DFAtable[i][j][0])] = DFAtable[i][j][1:]
        except:
            for k in range(len(Vt) + 1):
                LR0TABLE[i + 1][k] = 'r' + str(DFAtable[i][j])

#  输出 LR(0)Table
print('        ')
for i in range(len(LR0TABLE)):
    print('        ', end=' ')
    for j in range(len(LR0TABLE[i])):
        print(LR0TABLE[i][j], end='          ')
    print('')


#  基于 wxpython(wx)库的表格输出
class GridFrame(wx.Frame):
    def __init__(self, parent):
        global LR0TABLE

        wx.Frame.__init__(self, parent)

        # Create a wxGrid object
        grid = wx.grid.Grid(self, -1)

        # Then we call CreateGrid to set the dimensions of the grid
        # (100 rows and 10 columns in this example)
        grid.CreateGrid(len(LR0TABLE) + 5, len(VtVn) + 5)

        # We can set the sizes of individual rows and columns
        # in pixels
        grid.SetCellValue(0, 0, '分析表')
        grid.SetCellValue(0, 1, 'ACTION')
        grid.SetCellValue(0, 2, 'GOTO')

        for i in range(len(LR0TABLE)):
            grid.SetCellValue(i + 2, 0, str(i))
            for j in range(len(LR0TABLE[i])):
                grid.SetCellValue(i + 1, j + 1, LR0TABLE[i][j])

        self.Show()

app = wx.App(0)
frame = GridFrame(None)
app.MainLoop()

'''
# example
(1)
length:6
E->aA
```

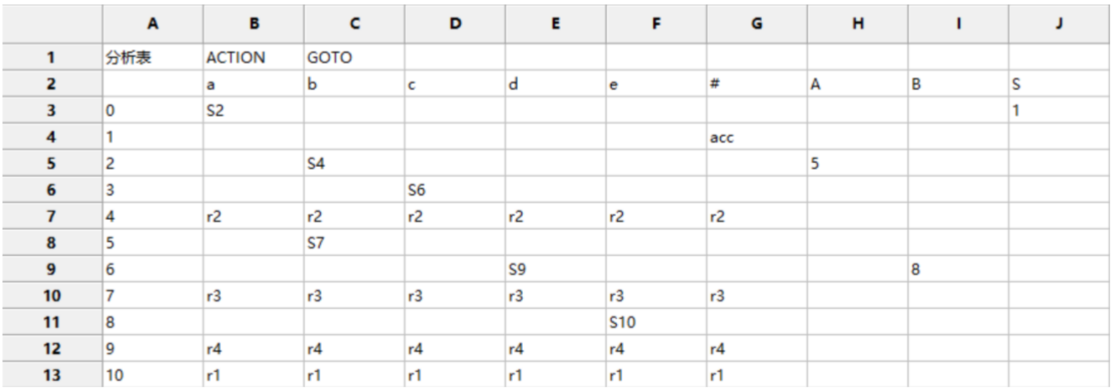
```
E->bB
A->cA
A->d
B->cB
B->d
(2)
length: 4
S->aAcBe
A->b
A->Ab
B->d
'''
```

【Experimental Results】

example 1:

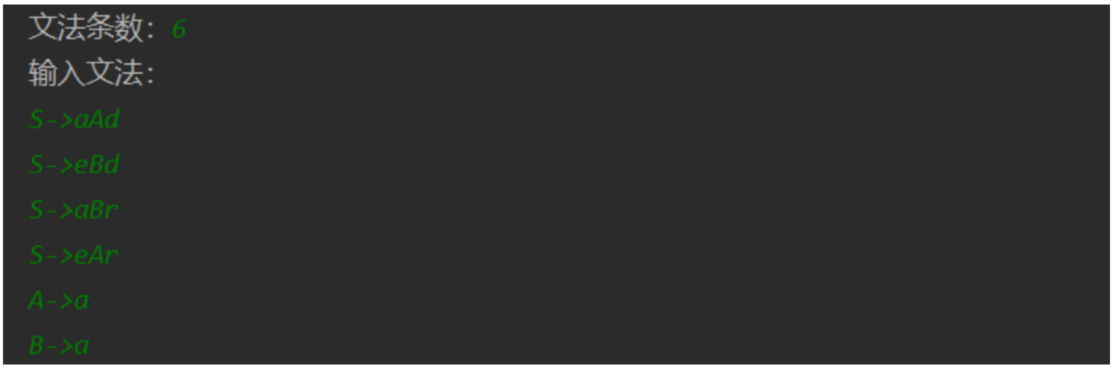


```
文法条数: 4
输入文法:
S->aAcBe
A->b
A->Ab
B->d
```

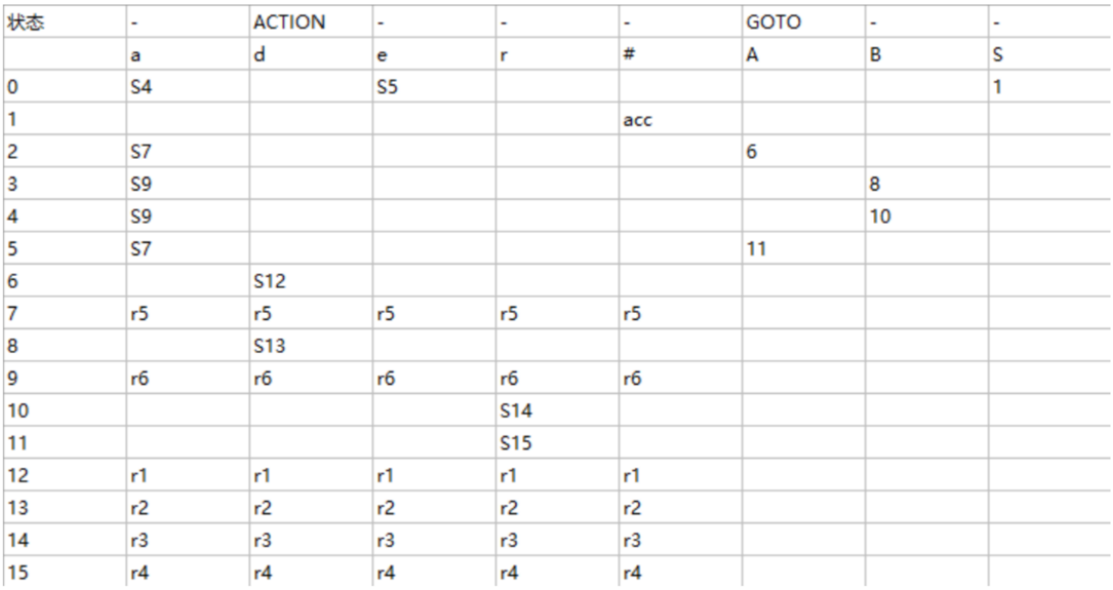


	A	B	C	D	E	F	G	H	I	J
1	分析表	ACTION	GOTO							
2		a	b	c	d	e	#	A	B	S
3	0	S2								1
4	1						acc			
5	2		S4					5		
6	3			S6						
7	4	r2	r2	r2	r2	r2	r2			
8	5		S7							
9	6				S9				8	
10	7	r3	r3	r3	r3	r3	r3			
11	8					S10				
12	9	r4	r4	r4	r4	r4	r4			
13	10	r1	r1	r1	r1	r1	r1			

Example 2:



```
文法条数: 6
输入文法:
S->aAd
S->eBd
S->aBr
S->eAr
A->a
B->a
```



状态	-	ACTION	-	-	-	GOTO	-	-
	a	d	e	r	#	A	B	S
0	S4		S5					1
1					acc			
2	S7					6		
3	S9						8	
4	S9						10	
5	S7					11		
6		S12						
7	r5	r5	r5	r5	r5			
8		S13						
9	r6	r6	r6	r6	r6			
10				S14				
11				S15				
12	r1	r1	r1	r1	r1			
13	r2	r2	r2	r2	r2			
14	r3	r3	r3	r3	r3			
15	r4	r4	r4	r4	r4			

【Experiment Summary】

This experiment focuses on the analysis process and table generation of LR(0) grammar. This experiment is more difficult, I tried many times. Finally, the experimental requirements were successfully realized through python . First input a set of grammars, and then output the corresponding LR (0) analysis table. During the experiment, I reviewed the knowledge about LR(0) in Chapter 6 of the book , mainly about the construction method of the LR(0) table. The main focus of this experiment is the construction of the item set, which can Enter the state update state set, output the statute information for the statute item, and finally generate the analysis table is also a difficult point. It is the same as the learning process of using Graph viz to generate the automaton state diagram in experiments 2 and 3. In this experiment, I used the w xpython library to check the information , and successfully output the analysis table results as required by the experiment format. After this experiment, I have improved my understanding of the analysis and judgment methods of LR (0) grammar, and laid a good foundation for future experimental study.