```
import numpy as np
import pandas as pd
```

```
LEXER_TABLE=pd.DataFrame(columns=['TOKEN','IDENTITY'],index=range(1))

def lexer(filename):
  with open(filename, 'r') as myfile:
    string = myfile.read().replace('\n',' ')
  SYMBOLS=['(',')',';',',',':','\'']
  SYMBOLS_1=['(',
             ')',
             ';',
             ',',
             ':',
             '\'',
             '+',
             '=',
             '>'
             ]

  SYMBOLS_1_DUP=[['Open bracket','('],
                 ['Close bracket',')'],
                 ['Semi colon',';'],
                 ['Comma',','],
                 ['Colon',':'],
                 ['Single quote','\''],
                 ['Plus','+'],
                 ['Equal','='],
                 ['Greater than','>']]

  keywords=['integer','main','while','begin','end',
            'expr',' ','\n','Procedure','If','elseif',
            'else','then','printf','and','or']

  keywords_def=[['t','integer'],
  ['b','begin'],['d','end'],['s',' '],
  ['o','+'],['o','='],['n','\n'],['p','Procedure'],
  ['x','If'],['z','elseif'],['y','else'],['f','then'],
  ['c','printf'],['&','and'],['|','or'],['o','>']]

  KEYWORDS=SYMBOLS_1 + keywords
  white_space=' '
  lexeme=''
  list=[]
  string=string.replace('\t','')
  for i,char in enumerate(string):
    if char != white_space:
```

```
      lexeme += char
    if(i+1 < len(string)):
      if string[i+1] == white_space or string[i+1] in KEYWORDS or
lexeme in KEYWORDS:
        if lexeme!='':
          if string[i+1]=='i' and lexeme=='else':
            continue
          else:
            list.append(lexeme.replace('\n','<newline>'))
            lexeme=''
            list.append(lexeme.replace('\n','<newline>'))
  s=''
  j=0
  try:
    while(True):
      list.remove('')
  except ValueError:
    pass
  for item in list:
    for i in keywords_def:
      if i[1]==item:
        s=s+i[0]
    if item in SYMBOLS:
      s=s+item
    elif item.isdigit():
      s=s+'a'
    elif item not in KEYWORDS:
      s=s+'v'
  for i in list:
    for k in SYMBOLS_1_DUP:
      if i==k[1]:
        LEXER_TABLE.at[j,'TOKEN']=i
        LEXER_TABLE.at[j,'IDENTITY']=k[0]
        j=j+1
        break
    if i in keywords:
      LEXER_TABLE.at[j,'TOKEN']=i
      LEXER_TABLE.at[j,'IDENTITY']='Keyword'
      j=j+1
      continue
    if i.isdigit():
      LEXER_TABLE.at[j,'TOKEN']=i
      LEXER_TABLE.at[j,'IDENTITY']='Digit'
      j=j+1
      continue
    elif i not in KEYWORDS:
      LEXER_TABLE.at[j,'TOKEN']=i
      LEXER_TABLE.at[j,'IDENTITY']='Identifier'
```

```
      j=j+1
  print(LEXER_TABLE)
  return s

EPSILON = "ε"
```

```
def get_productions(X):
  productions = []
  for prod in grammar:
    lhs, rhs = prod.split('->')
    if lhs == X:
      rhs = '.'+rhs
      productions.append('->'.join([lhs, rhs]))
  return productions
```

```
def closure(I):
  for production, a in I:
    if production.endswith("."):
      continue
    lhs, rhs = production.split('->')
    alpha, B_beta = rhs.split('.')
    B = B_beta[0]
    beta = B_beta[1:]
    beta_a = a
    if beta:
      beta_a = beta[0]+a
    # print(beta_a)
    first_beta_a = first(beta_a)
    # print(first_beta_a)
    for b in first_beta_a:
      B_productions = get_productions(B)
      # print(B_productions)
      for gamma in B_productions:
        new_item = (gamma, b)
        # print(new_item)
        if (new_item not in I):
          I.append(new_item)
    # print(I)
  return I
```

```
def get_symbols(grammar):
  terminals = set()
  non_terminals = set()
  for production in grammar:
    lhs, rhs = production.split('->')
    non_terminals.add(lhs)
    for x in rhs:
      terminals.add(x)
      terminals = terminals.difference(non_terminals)
  terminals.add('$')
  print(terminals, non_terminals)
  return terminals, non_terminals
```

```
def first(symbols):
  final_set = set()
  # print(symbols)
  for X in symbols:
    first_set = set()
    if isTerminal(X):
      final_set.add(X)
      # print(final_set)
      # return final_set
    else:
      for production in grammar:
        lhs, rhs = production.split('->')
        if lhs == X:
          for i in range(len(rhs)):
            y = rhs[i]
            if y == X:
              continue
            first_y = first(y)
            # print(first_y)
            if EPSILON in first_y:
              first_y.replace(EPSILON,"")
              first_set.add(first_y)
              continue
            else:
              first_set.add(first_y)
              break
    for i in first_set:
      if EPSILON not in i:
        final_set.add(i)
      else:
        i.replace(EPSILON,"")
        final_set.add(i)
  # print("".join(list(final_set)))
  return "".join(list(final_set))
```

```
def isTerminal(symbol):
  return symbol in terminals
```

```
def shift_dot(production):
  lhs, rhs = production.split('->')
  x, y = rhs.split(".")
  if(len(y) == 0):
    print("Dot at the end!")
    return
  elif len(y) == 1:
    y = y+"."
  else:
    y = y[0]+"."+y[1:]
  rhs = "".join([x, y])

  # print("->".join([lhs, rhs]))
  return "->".join([lhs, rhs])
```

```
def goto(I, X):
  J = []
  for production, look_ahead in I:
    lhs, rhs = production.split('->')
    if "."+X in rhs and not rhs[-1] == '.':
      new_prod = shift_dot(production)
      J.append((new_prod, look_ahead))
  # print(J)
  return closure(J)
```

```
def set_of_items(display=False):
  num_states = 1
  states = ['I0']
  items = {'I0': closure([('P->.S', '$')])}
  for I in states:
      for X in pending_shifts(items[I]):
        goto_I_X = goto(items[I], X)
        if goto_I_X != None:
          if len(goto_I_X) > 0 and goto_I_X not in items.values():
            new_state = "I"+str(num_states)
            states.append(new_state)
            items[new_state] = goto_I_X
            num_states += 1
  if display:
    for i in items:
      print("State", i, ":")
      for x in items[i]:
        print(x)
      print()
```

```
  return items
```

```
def pending_shifts(I):
  symbols = []
  for production, _ in I:
    lhs, rhs = production.split('->')
    if rhs.endswith('.'):
      continue
    beta = rhs.split('.')[1][0]
    if beta not in symbols:
      symbols.append(beta)
  return symbols
```

```
def done_shifts(I):
  done = []
  for production, look_ahead in I:
    if production.endswith('.') and production != 'P->S.':
      done.append((production[:-1], look_ahead))
  return done
```

```
def get_state(C, I):
  key_list = list(C.keys())
  val_list = list(C.values())
  i = val_list.index(I)
  return key_list[i]
```

```
def CLR_construction(num_states):
  C = set_of_items()
  ACTION = pd.DataFrame(columns=list(terminals),
index=range(num_states))
  GOTO = pd.DataFrame(columns=list(non_terminals),
index=range(num_states))
  for Ii in C.values():
    i = int(get_state(C, Ii)[1:])
    pending = pending_shifts(Ii)
    for a in pending:
      Ij = goto(Ii, a)
      j = int(get_state(C, Ij)[1:])
      if isTerminal(a):
        ACTION.at[i, a] = "Shift "+str(j)
      else:
        GOTO.at[i, a] = j
    for production, look_ahead in done_shifts(Ii):
      # print(production,look_ahead)
```

```
      ACTION.at[i, look_ahead] = "Reduce " +
str(grammar.index(production)+1)
    if ('P->S.', '$') in Ii:
      ACTION.at[i, '$'] = "Accept"
  ACTION.replace(np.nan, '', regex=True, inplace=True)
  GOTO.replace(np.nan, '', regex=True, inplace=True)
  print(ACTION)
  print(GOTO)
  return ACTION, GOTO
```

```
def parse_string(string, ACTION, GOTO):
  row = 0
  cols = ['Stack', 'Input', 'Output']
  if not string.endswith('$'):
    string = string+'$'
  ip = 0
  PARSE = pd.DataFrame(columns=cols)
  input = list(string)
  stack = ['$', '0']
  while True:
    S = int(stack[-1])
    a = input[ip]
    action = ACTION.at[S, a]
    new_row = ["".join(stack), "".join(input[ip:]), action]
    if 'S' in action:
      S1 = action.split()[1]
      stack.append(a)
      stack.append(S1)
      ip += 1
    elif "R" in action:
      i = int(action.split()[1])-1
      A, beta = grammar[i].split('->')
      for _ in range(2*len(beta)):
        stack.pop()
      S1 = int(stack[-1])
      stack.append(A)
      stack.append(str(GOTO.at[S1, A]))
      new_row[-1] = "Reduce "+grammar[i]
    elif action == "Accept":
      PARSE.loc[row] = new_row
      print(PARSE,"\n")
      print("\nThe Source code entered is according to the grammar\n")
      return PARSE
    else:
      print(PARSE,"\n")
      print("\nThe Source code entered is incorrect\n")
      break
```

```
    PARSE.loc[row] = new_row
    row += 1
```

```
def get_grammar(filename):
  grammar = []
  F = open(filename, "r")
  for production in F:
    grammar.append(production[:-1])
  # print(grammar)
  return grammar
```

```
grammar = get_grammar("gramm3.txt")
terminals, non_terminals = get_symbols(grammar)
symbols = terminals.union(non_terminals)
C = set_of_items()
ACTION, GOTO = CLR_construction(num_states=len(C))
string = "".join(lexer('program.txt'))
print(string)
PARSE_TABLE = parse_string(string, ACTION, GOTO)
```