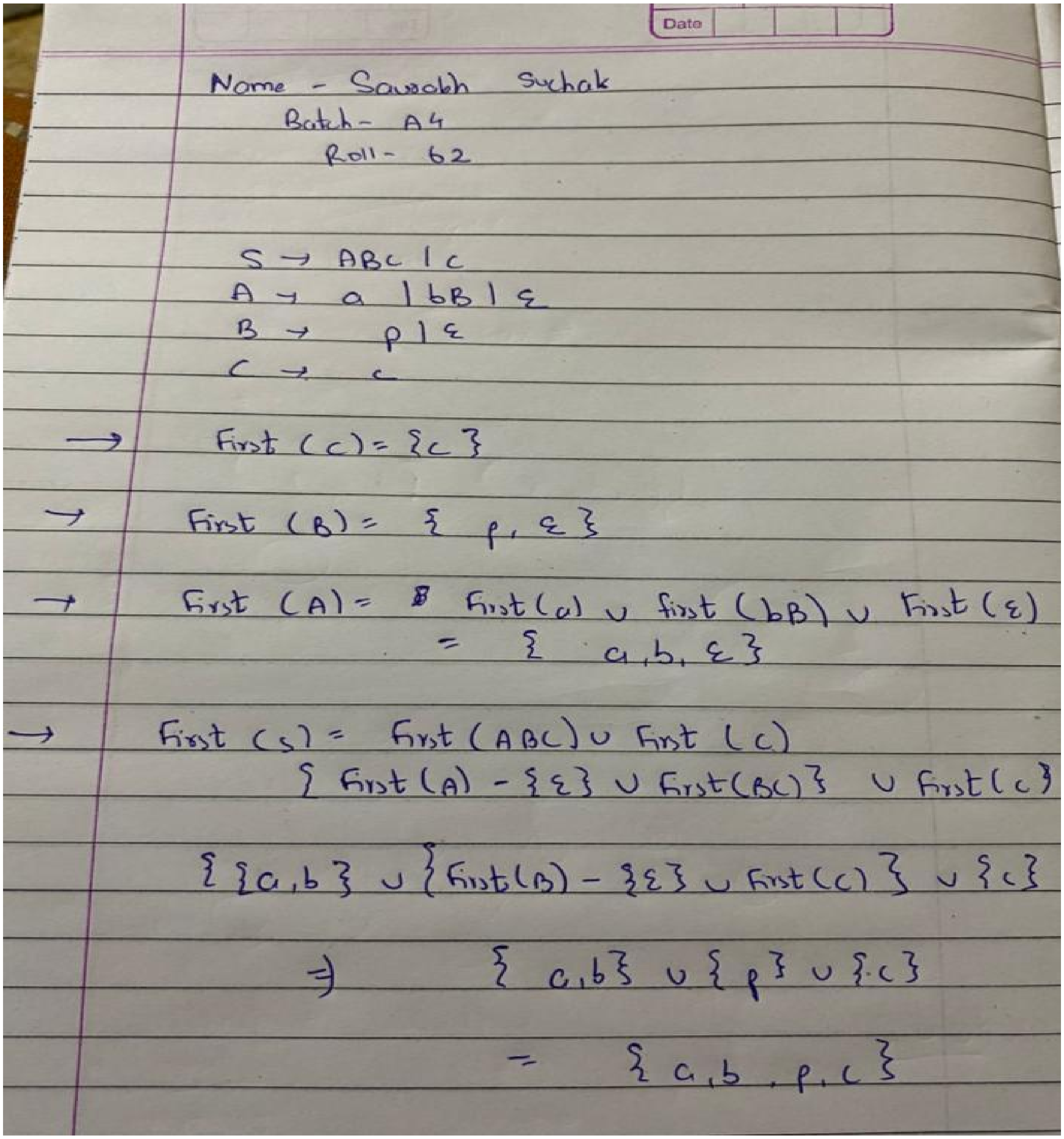
Practical - 03

Aim (A) - Write a program to find FIRST for any grammar. All the following rules of FIRST must be implemented.

Solved Problem -



Date

Name - Saurabh Suchak
Batch- A4
Roll- 62

$S \rightarrow ABC \mid c$
$A \rightarrow a \mid bB \mid \varepsilon$
$B \rightarrow p \mid \varepsilon$
$C \rightarrow c$

→ $\text{First}(C) = \{c\}$

→ $\text{First}(B) = \{p, \varepsilon\}$

→ $\text{First}(A) = \text{First}(a) \cup \text{first}(bB) \cup \text{First}(\varepsilon)$
$= \{a, b, \varepsilon\}$

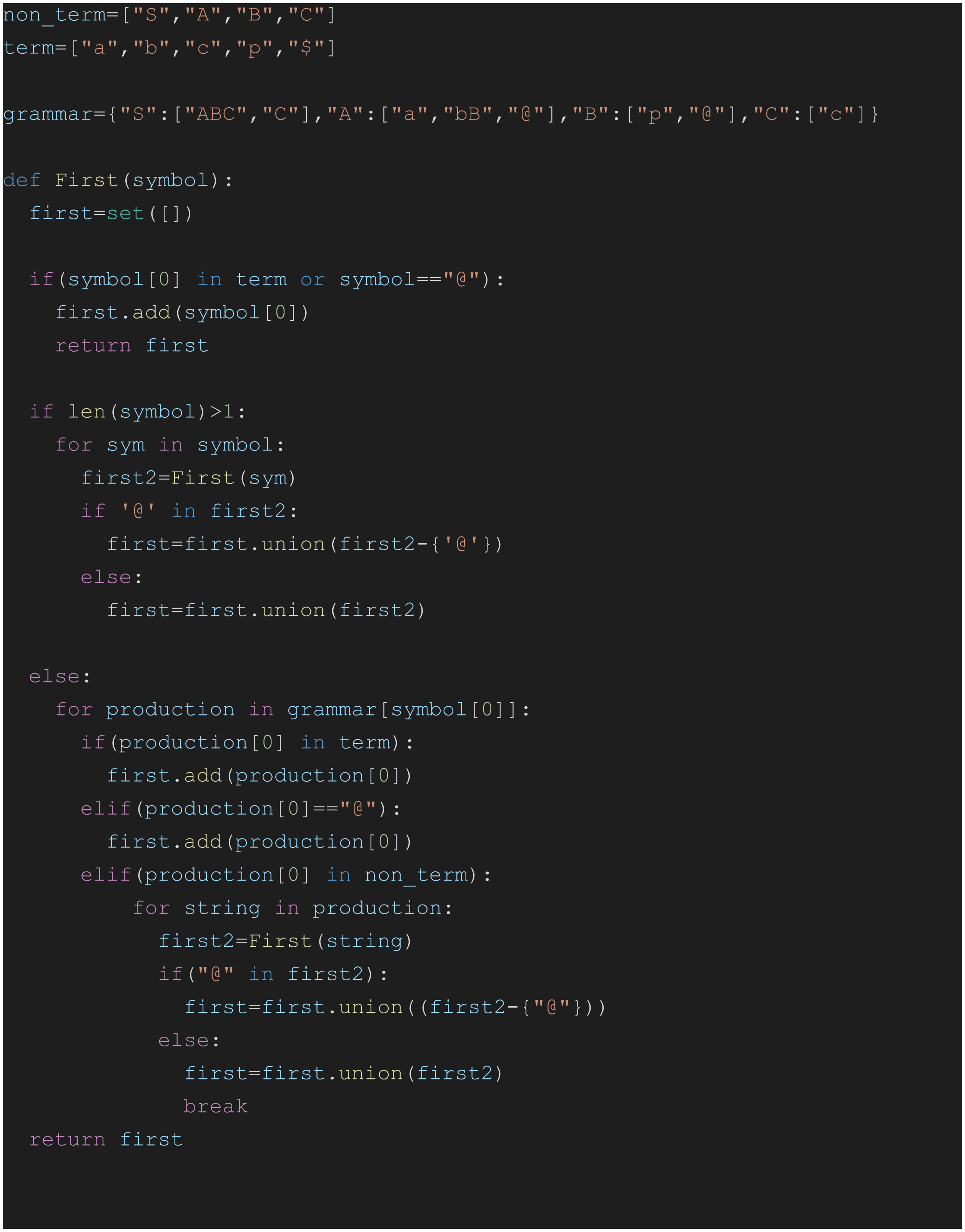
→ $\text{First}(S) = \text{First}(ABC) \cup \text{First}(c)$
$\{\text{First}(A) - \{\varepsilon\} \cup \text{First}(BC)\} \cup \text{First}(c\}$

$\{\{a,b\} \cup \{\text{First}(B) - \{\varepsilon\} \cup \text{First}(C)\} \cup \{c\}$

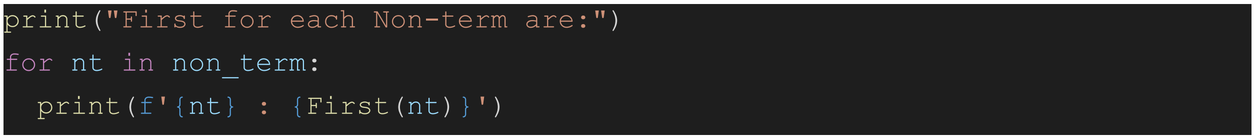
$\Rightarrow \{a,b\} \cup \{p\} \cup \{c\}$

$= \{a, b, p, c\}$

Code -



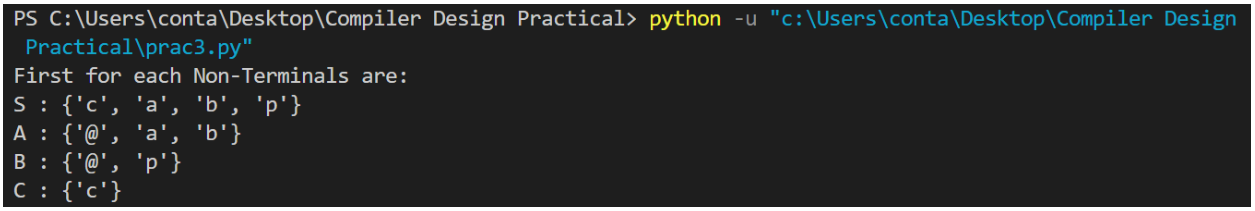
```
non_term=["S","A","B","C"]
term=["a","b","c","p","$"]

grammar={"S":["ABC","C"],"A":["a","bB","@"],"B":["p","@"],"C":["c"]}

def First(symbol):
  first=set([])

  if(symbol[0] in term or symbol=="@"):
    first.add(symbol[0])
    return first

  if len(symbol)>1:
    for sym in symbol:
      first2=First(sym)
      if '@' in first2:
        first=first.union(first2-{'@'})
      else:
        first=first.union(first2)

  else:
    for production in grammar[symbol[0]]:
      if(production[0] in term):
        first.add(production[0])
      elif(production[0]=="@"):
        first.add(production[0])
      elif(production[0] in non_term):
          for string in production:
            first2=First(string)
            if("@" in first2):
              first=first.union((first2-{"@"}))
            else:
              first=first.union(first2)
              break
  return first
```



```
print("First for each Non-term are:")
for nt in non_term:
  print(f'{nt} : {First(nt)}')
```

Output for problem statement (A) -

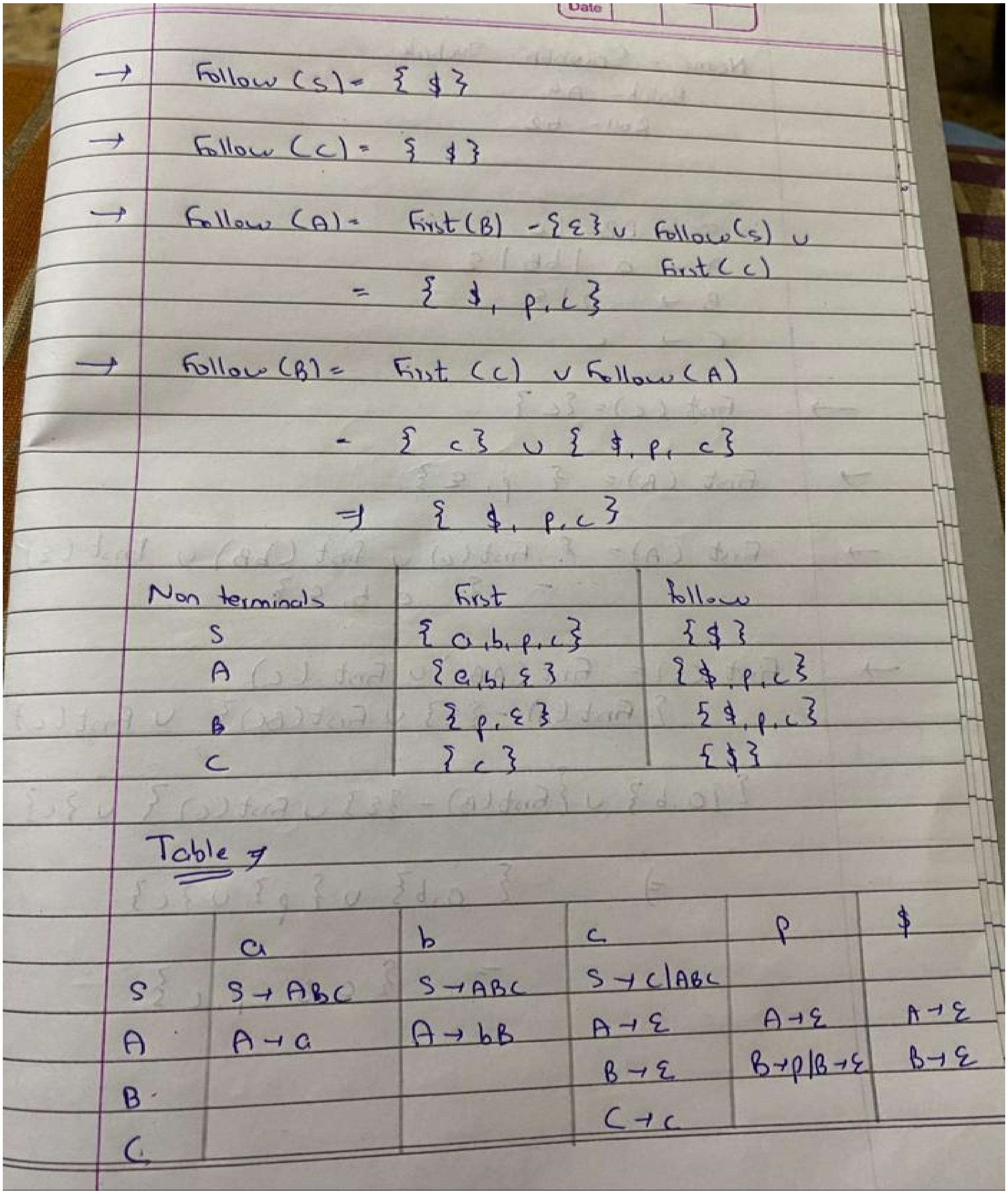
- @ is used in place of epsilon



```
PS C:\Users\conta\Desktop\Compiler Design Practical> python -u "c:\Users\conta\Desktop\Compiler Design
 Practical\prac3.py"
First for each Non-Terminals are:
S : {'c', 'a', 'b', 'p'}
A : {'@', 'a', 'b'}
B : {'@', 'p'}
C : {'c'}
```

Aim (B) - Calculate Follow for the given grammar and Construct the LL (1) parsing table using the FIRST and FOLLOW .

Solved Problem -



Date

→ Follow (S) = { $ }

→ Follow (C) = { $ }

→ Follow (A) = First (B) - {ε} ∪ Follow(S) ∪ First (C)

= { $, p, c }

→ Follow (B) = First (C) ∪ Follow (A)

= { c } ∪ { $, p, c }

⇒ { $, p, c }

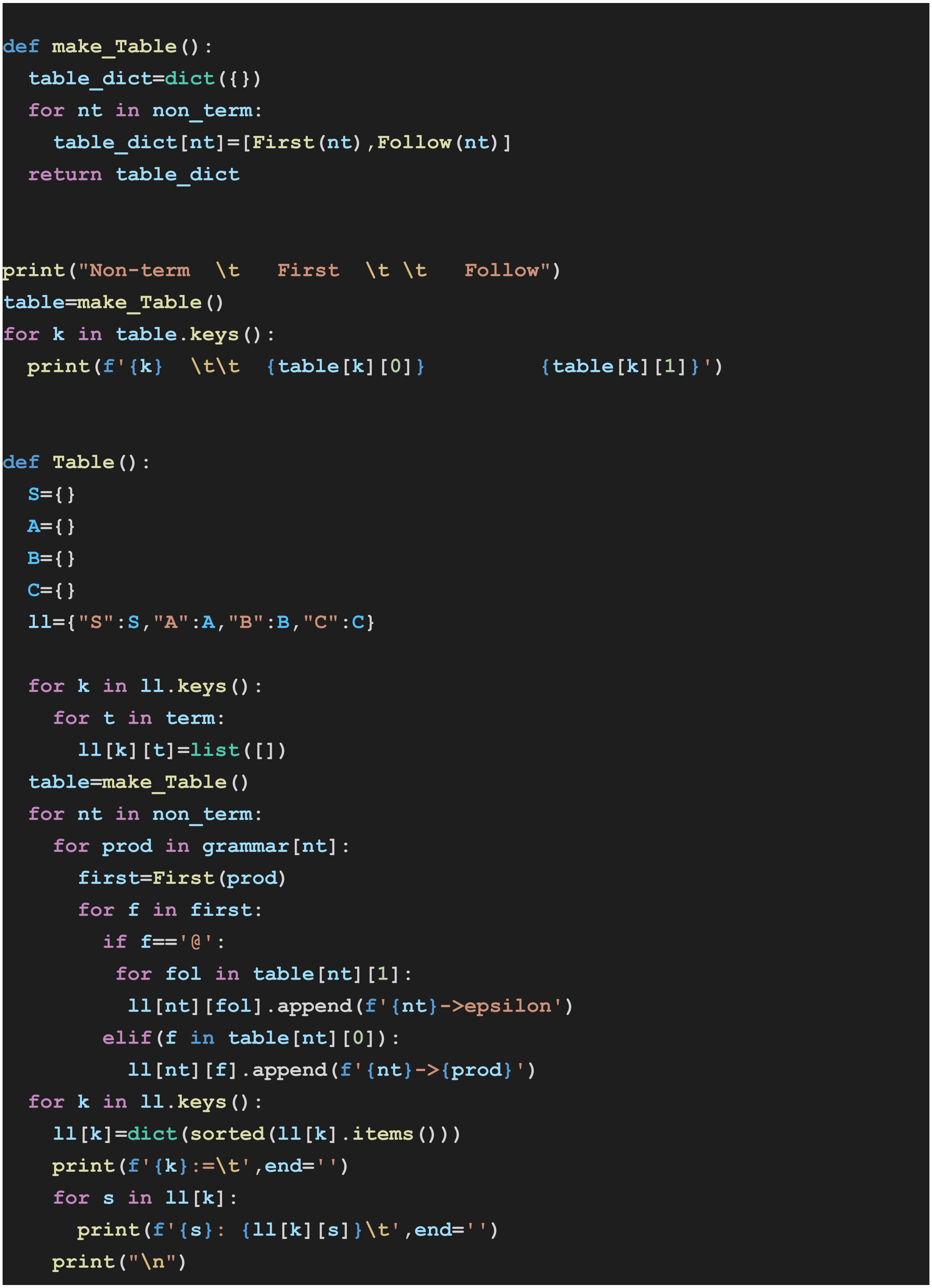
Non terminals	First	Follow
S	{ a, b, p, c }	{ $ }
A	{ a, b, ε }	{ $, p, c }
B	{ p, ε }	{ $, p, c }
C	{ c }	{ $ }

Table ⇒

	a	b	c	p	$
S	S → ABC	S → ABC	S → c|ABC		
A	A → a	A → bB	A → ε	A → ε	A → ε
B			B → ε	B → p|B → ε	B → ε
C			C → c		

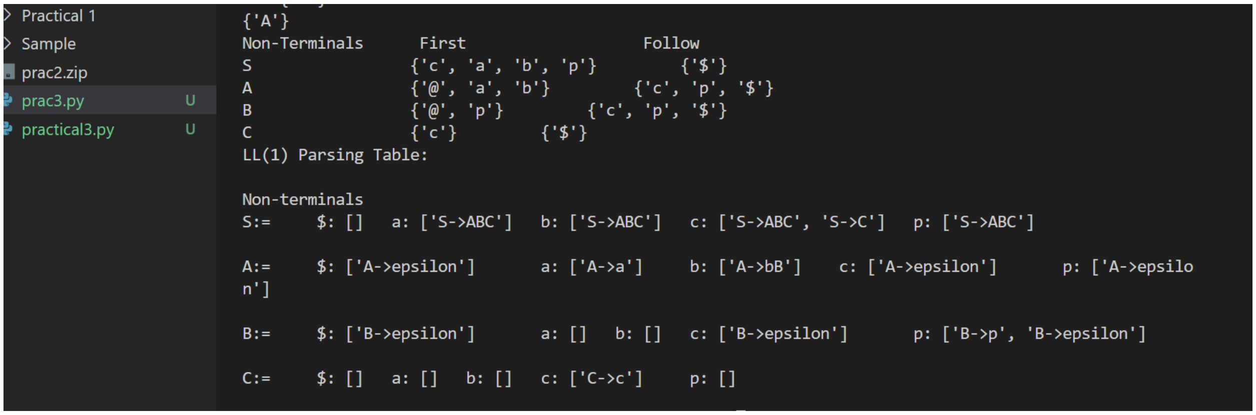
Code for statement (B) -

```
def get_key(val):
    keys=set([])
    for key, value in grammar.items():
        for string in value:
          for letter in string:
           if val == letter:
             keys.add(key)
    return keys

print(get_key("a"))

def Follow(symbol):
  follow=set([])
  if(symbol=="S"):
    follow.add("$")
  keys=get_key(symbol)
  for k in keys:
    for production in grammar[k]:
      for i in range(0,len(production)):
          if production[i]==symbol :
            j=i
            if j!=len(production)-1:
              for j in range(i,len(production)):
                first=First(production[j+1])
                if('@' in  first):
                  follow=follow.union(Follow(k))
                  follow=follow.union((first-{'@'}))
                else:
                  follow=follow.union(first)
                  break
            elif(production[i]==symbol and i==len(production)-1):
                follow=follow.union(Follow(k))
            elif(production[i]==symbol and symbol==k):
                return

  return follow
```



```
def make_Table():
  table_dict=dict({})
  for nt in non_term:
    table_dict[nt]=[First(nt),Follow(nt)]
  return table_dict


print("Non-term  \t   First  \t \t   Follow")
table=make_Table()
for k in table.keys():
  print(f'{k}  \t\t  {table[k][0]}         {table[k][1]}')


def Table():
  S={}
  A={}
  B={}
  C={}
  ll={"S":S,"A":A,"B":B,"C":C}

  for k in ll.keys():
    for t in term:
      ll[k][t]=list([])
  table=make_Table()
  for nt in non_term:
    for prod in grammar[nt]:
      first=First(prod)
      for f in first:
        if f=='@':
         for fol in table[nt][1]:
          ll[nt][fol].append(f'{nt}->epsilon')
        elif(f in table[nt][0]):
          ll[nt][f].append(f'{nt}->{prod}')
  for k in ll.keys():
    ll[k]=dict(sorted(ll[k].items()))
    print(f'{k}:=\t',end='')
    for s in ll[k]:
      print(f'{s}: {ll[k][s]}\t',end='')
    print("\n")
print("LL(1) Parsing Table:\n")
print("Non-term")
Table()
```

Output -



```
{'A'}
Non-Terminals      First                    Follow
S                 {'c', 'a', 'b', 'p'}         {'$'}
A                 {'@', 'a', 'b'}         {'c', 'p', '$'}
B                 {'@', 'p'}         {'c', 'p', '$'}
C                 {'c'}         {'$'}
LL(1) Parsing Table:

Non-terminals
S:=     $: []   a: ['S->ABC']   b: ['S->ABC']   c: ['S->ABC', 'S->C']   p: ['S->ABC']

A:=     $: ['A->epsilon']       a: ['A->a']     b: ['A->bB']    c: ['A->epsilon']       p: ['A->epsilo
n']

B:=     $: ['B->epsilon']       a: []   b: []   c: ['B->epsilon']       p: ['B->p', 'B->epsilon']

C:=     $: []   a: []   b: []   c: ['C->c']     p: []
```