**Program-1**

**Aim**- **To identify keyword, variables, operators and constant in a program.**

**Theory-**

**Variables**- A variable is simple terms in storage place which has some memory allocated to it. Basically, a variable used to store some form of data. Different types of variables require different amount of memory, and have some specific set of operations which can be applied to them.

Rules to define variables - A variable name can consist of alphabets (both upper and lower case), numbers and the underscore ‘\_’ character. However, the name must not start with a number. e.g. name, roll\_num, name2.

**Keywords**- In programming a keyword is a word that is reserved by a program because the word has a special meaning. Keywords can be commands or parameters. Every Programming Languages has a set of keywords that cannot be used as [variable](https://www.webopedia.com/definitions/variable/) [names](https://www.webopedia.com/definitions/name/). Keywords are sometimes called reserved names. e.g. for, while, break, continue etc.

**Operators** - An operator, in computer programming, is a symbol that usually represents an action or process. These symbols were adapted from mathematics and logic. An operator is capable of manipulating a certain value or operand.

**i)** **Unary operators**: Those which require only one operand to operate upon. e.g. unary -, unary +, ++, - -!.

**(ii) Binary operators**: Binary operators require two operands to operate upon. e.g. +, \*, /, -, etc.

**(iii) Ternary Operator:** Ternary operator require three operands to operate upon. Conditional operator (? :) is a ternary operator in C.

**Algorithm:**

function\_name ( input ):

1. Read the input and make a list of each line.
2. split lines on each word and store in a list words.
3. define list of all operators(+,\*,/,-,//).
4. define list keywords and store names of all valid keywords.
5. initialize counts=0
6. loop for i=0 to words.length
7. if words[i] in keywords
8. count\_keywords +=1
9. elif words[i] in operators
10. count\_operators +=1
11. else:
12. character = words[i]
13. if((character[0] != digits) and (delimiter not in character)):
14. count\_identifiers +=1
15. return counts.

**Program-2**

**Aim:** Write a program to count number of comment lines with in a program.

**Theory:** Comments are text notes added to the program to provide explanatory information about the source code. They are used in a programming language to document the program and remind programmers of what tricky things they just did with the code and also helps the later generation for understanding and maintenance of code. The compiler considers these as non-executable statements. Different programming.

C language has 2 types of comments.

1. Single Line Comments (starts with //).

2. Multi Line comments (starts with /\* and ends with \*/).

**Algorithm:**

CommentsCount(input)

1. open input file as 'r'.

2. read file line by line.

3. initialize count=0

4. loop until end of file.

5. Now, if // in a line increase count by 1.

6. if ' /\* ' in a line then use loop until ' \*/ ' is found in a line

and increase count on every line.

7. print count.

**Program:**

def commentsCount(file):

readfile=open(file,'r')

line=readfile.readline()

count=0

while(line):

if('//' in line):

count+=1

if('/\*' in line):

while('\*/' not in line):

count+=1

line=readfile.readline()

count+=1

line=readfile.readline()

print('No. of comment line in a c program is: ',count)

commentsCount('main.c')

**Input File: (main.c)**

# include<stdio.h>

/\*it is the programme

to add two numbers

a and b \*/

int main(){

int a,b,c; //initialize a,b,c

a=10;

b=12;

c=a+b;

printf("%d",c); //printing statements

return 0;

}

**Output:**

****

**Program-3**

**Aim:** Write a Program to take as input assembly program and create the symbol table.

**Theory:**

**Assembly language:** Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler.

**Symbol table:** Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

The symbol table used for following purposes:

* It is used to store the name of all entities in a structured form at one place.
* It is used to verify if a variable has been declared.
* It is used to determine the scope of a name.
* It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

**Algorithm:**

SymbolTable(input)

1. make lists of all keywords.

2. read file line by line using Readline functon.

3. loop until end of file.

a. Make a list of all words in a line using split().

b. set relocation to ‘R’ and change to ‘A’ only when there is EQU instruction in a line.

c. if words[0] == NULL then

1. if words[1] == ‘START then assign address.
2. if words[1] in four\_bytes increment address by 4.
3. if words[1] in two\_bytes increment address by 2.

d. else

1. at each line append words[0] which have labels to label\_list.
2. if words[1] == ‘START then assign address and append the address to address\_list.
3. elif words[1] in four\_bytes then append the address to address\_list and increment address by 4.
4. elif words[1] in two\_bytes then append the address to address\_list and increment address by 2.
5. elif words[1] in declare then.

* if words[1] is DS then append the address to address\_list and increment address by fetch the limit and multiply it by 4 if F (full word) or 2 if H (half word).
* Else words[1] is DC now if H in words[2] then increment address by 2 else if F in words[2] increment address by 4.

vi. elif words[1] is EQU then set relocation to ‘A’ and append words[2] value to address\_list.

e. Read next line. END

Display\_Table(label\_list, address\_list,relocation\_list)

1. loop until end of list.

Print in proper row column format.

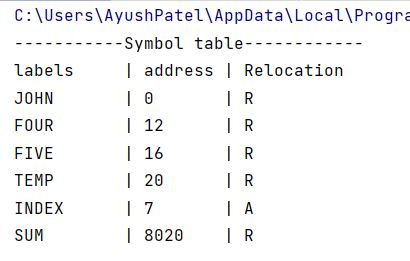
**Program:**

global address\_list, label\_list, relocation\_list  
address\_list = []  
label\_list = []  
relocation\_list = []  
  
*# function to display the table.*def display\_Table(label\_list, address\_list, relocation\_list):  
 print("-----------Symbol table------------")  
 print("{0:10s} | {1:<7} | {2:<4}".format(  
 "labels", "address", "Relocation"))  
 for i in range(len(address\_list)):  
 print("{0:10s} | {1:<7} | {2:<4}".format(  
 label\_list[i], address\_list[i], relocation\_list[i]))  
  
  
def symbolTable(input\_file):  
 global address\_list, label\_list, relocation\_list  
 four\_bytes = ['LA', 'L', 'A', 'ST', 'C']  
 two\_bytes = ['SR', 'AR', 'BNE', 'LR', 'BR']  
 declare = ['DS', 'DC']  
 zero\_bytes = ['USING']  
 file = open(input\_file, 'r')  
 line = file.readline()  
 while(line):  
 relocation = 'R'  
 words = line.split()  
 if(words[0] == 'NULL'):  
 if(words[1] == 'START'):  
 address = int(words[2])  
 if(words[1] in four\_bytes):  
 address += 4  
 elif (words[1] in two\_bytes):  
 address += 2  
  
 else:  
 label\_list.append(words[0])  
 if(words[1] == 'START'):  
 address = int(words[2])  
 address\_list.append(address)  
 elif(words[1] in four\_bytes):  
 address\_list.append(address)  
 address += 4  
 elif(words[1] in two\_bytes):  
 address\_list.append(address)  
 address += 2  
 elif(words[1] in declare):  
 i = 0  
 while True:  
 if(words[2][i] == 'F' or words[2][i] == 'H'):  
 break  
 i += 1  
 if(words[1] == 'DS'):  
 address\_list.append(address)  
 no\_of\_words = int(words[2][:i])  
 if(words[2][i] == 'F'):  
 address += no\_of\_words\*4  
 else:  
 address += no\_of\_words \* 2  
 else:  
 address\_list.append(address)  
 if(words[2][i] == 'F'):  
 address += 4  
 else:  
 address += 2  
 elif(words[1] == 'EQU'):  
 value = words[2]  
 relocation = 'A'  
 address\_list.append(value)  
 relocation\_list.append(relocation)  
 line = file.readline()  
  
*#Function calls.*symbolTable('assembly.txt')  
display\_Table(label\_list, address\_list, relocation\_list)

**Input:** assembly.txt

JOHN START 0  
NULL USING \*,15  
NULL L 1,FIVE  
NULL A 1,FOUR  
NULL ST 1,TEMP  
FOUR DC F'4'  
FIVE DC F'5'  
TEMP DS 2000F  
INDEX EQU 7  
SUM A FOUR,FIVE  
NULL END

**Output:**



**PROGRAM 5**

# AIM: Write a program to check whether a given grammar is Operator precedence or not.

# THEORY:

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

* No R.H.S. of any production has a∈.
* No two non-terminals are adjacent.

Operator precedence can only be established between the terminals of the grammar. It ignores the non-terminal.

**ALGORITHM:**

isOperatorPrecedence(production)

1. Read RHS of production character by character.
2. Now, if character == ‘epsilon’ return 0 (i.e. grammer is not operator precedence).
3. Else
4. if two adjacent characters are non-terminal then return 0.
5. Else return 1 (i.e. grammer is operator precedence)

**PROGRAM:**

import re

def isOperatorPrecedence(production):

characters\_in\_production = production.split()

for i in range(2,len(characters\_in\_production)):

if(characters\_in\_production[i]=="epsilon"):

return 0

else:

nonterminals = re.compile('[A-Z]')

if(nonterminals.match(characters\_in\_production[i]) and nonterminals.match(characters\_in\_production[i+1])):

return 0

return 1

production = input("Enter Production write each character with a space: ")

flag=isOperatorPrecedence(production)

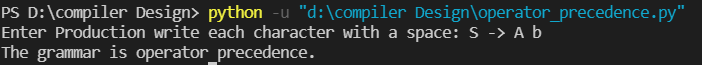
if flag == 1:

print("The grammar is operator precedence.")

else:

print("The grammar is not operator grammar.")

**OUTPUT:**



**PROGRAM 6**

# AIM: Write a program to remove left recursion and left factoring from a grammar

**THEORY:**

**Left recursion**:

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

A → A α |β.

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

A → βA′

A → αA′|ϵ

**Left Factoring:**

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

**Example**

If a top-down parser encounters a production like

A ⟹ αβ | α𝜸 | …

Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

**Example**

The above productions can be written as

A => αA'

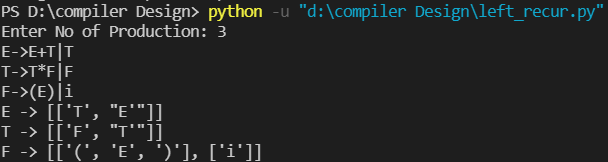
A'=> β | 𝜸 | …

**PROGRAM:**

**Left recursion:**

from itertools import takewhile  
  
gram = {}  
  
  
def add(str): # to rules together  
 x = str.split("->")  
 y = x[1]  
 x.pop()  
 z = y.split("|")  
 x.append(z)  
 gram[x[0]] = x[1]  
  
  
def removeDirectLR(gramA, A):  
 *"""gramA is dictonary"""* temp = gramA[A]  
 tempCr = []  
 tempInCr = []  
 for i in temp:  
 if i[0] == A:  
 # tempInCr.append(i[1:])  
 tempInCr.append(i[1:] + [A + "'"])  
 else:  
 # tempCr.append(i)  
 tempCr.append(i + [A + "'"])  
 tempInCr.append(["e"])  
 gramA[A] = tempCr  
 gramA[A + "'"] = tempInCr  
 return gramA  
  
  
def checkForIndirect(gramA, a, ai):  
 if ai not in gramA:  
 return False  
 if a == ai:  
 return True  
 for i in gramA[ai]:  
 if i[0] == ai:  
 return False  
 if i[0] in gramA:  
 return checkForIndirect(gramA, a, i[0])  
 return False  
  
  
def rep(gramA, A):  
 temp = gramA[A]  
 newTemp = []  
 for i in temp:  
 if checkForIndirect(gramA, A, i[0]):  
 t = []  
 for k in gramA[i[0]]:  
 t = []  
 t += k  
 t += i[1:]  
 newTemp.append(t)  
  
 else:  
 newTemp.append(i)  
 gramA[A] = newTemp  
 return gramA  
  
  
def rem(gram):  
 c = 1  
 conv = {}  
 gramA = {}  
 revconv = {}  
 for j in gram:  
 conv[j] = "A" + str(c)  
 gramA["A" + str(c)] = []  
 c += 1  
  
 for i in gram:  
 for j in gram[i]:  
 temp = []  
 for k in j:  
 if k in conv:  
 temp.append(conv[k])  
 else:  
 temp.append(k)  
 gramA[conv[i]].append(temp)  
  
 # print(gramA)  
 for i in range(c - 1, 0, -1):  
 ai = "A" + str(i)  
 for j in range(0, i):  
 aj = gramA[ai][0][0]  
 if ai != aj:  
 if aj in gramA and checkForIndirect(gramA, ai, aj):  
 gramA = rep(gramA, ai)  
  
 for i in range(1, c):  
 ai = "A" + str(i)  
 for j in gramA[ai]:  
 if ai == j[0]:  
 gramA = removeDirectLR(gramA, ai)  
 break  
  
 op = {}  
 for i in gramA:  
 a = str(i)  
 for j in conv:  
 a = a.replace(conv[j], j)  
 revconv[i] = a  
  
 for i in gramA:  
 l = []  
 for j in gramA[i]:  
 k = []  
 for m in j:  
 if m in revconv:  
 k.append(m.replace(m, revconv[m]))  
 else:  
 k.append(m)  
 l.append(k)  
 op[revconv[i]] = l  
  
 return op  
  
  
n = int(input("Enter No of Production: "))  
for i in range(n):  
 txt = input()  
 add(txt)  
  
result = rem(gram)  
  
for x, y in result.items():  
 print(f'{x} -> {y}')

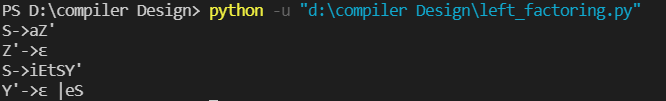
**OUTPUT:**



**Left factoring:**

from itertools import takewhile  
  
  
def groupby(ls):  
 d = {}  
 ls = [y[0] for y in rules]  
 initial = list(set(ls))  
 for y in initial:  
 for i in rules:  
 if i.startswith(y):  
 if y not in d:  
 d[y] = []  
 d[y].append(i)  
 return d  
  
  
def prefix(x):  
 return len(set(x)) == 1  
  
starting = ""  
rules = []  
common = []  
alphabetset = ["A'", "B'", "C'", "D'", "E'", "F'", "G'", "H'", "I'", "J'", "K'", "L'", "M'", "N'", "O'", "P'", "Q'",  
 "R'", "S'", "T'", "U'", "V'", "W'", "X'", "Y'", "Z'"]  
  
s = "S->iEtS|iEtSeS|a"  
while True:  
 rules = []  
 common = []  
 split = s.split("->")  
 starting = split[0]  
 for i in split[1].split("|"):  
 rules.append(i)  
  
 # logic for taking commons out  
 for k, l in groupby(rules).items():  
 r = [l[0] for l in takewhile(prefix, zip(\*l))]  
 common.append(''.join(r))  
 # end of taking commons  
 for i in common:  
 newalphabet = alphabetset.pop()  
 print(starting + "->" + i + newalphabet)  
 index = []  
 for k in rules:  
 if (k.startswith(i)):  
 index.append(k)  
 print(newalphabet + "->", end="")  
 for j in index[:-1]:  
 stringtoprint = j.replace(i, "", 1) + "|"  
 if stringtoprint == "|":  
 print("\u03B5", "|", end="")  
 else:  
 print(j.replace(i, "", 1) + "|", end="")  
 stringtoprint = index[-1].replace(i, "", 1) + "|"  
 if stringtoprint == "|":  
 print("\u03B5", "", end="")  
 else:  
 print(index[-1].replace(i, "", 1) + "", end="")  
 print("")  
 break

**OUTPUT:**



**PROGRAM 7**

# AIM: Write a program to recognize the string a\*, a\*b+ and abb

**THEORY:**

* The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
* The languages accepted by some regular expressions are referred to as Regular languages.
* A regular expression can also be described as a sequence of pattern that defines a string.
* Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

**For instance:**

* In a regular expression, x\* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, ....}
* In a regular expression, x+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx, ....}

**PROGRAM:**

def finite\_automata(expression):

state=0

for i in expression:

if state == 0:

if i=='a':

state=1

elif i=='b':

state=2

else:

state=6

elif state == 1:

if i=='a':

state=3

elif i=='b':

state=4

else:

state=6

elif state == 2:

if i=='b':

state=2

else:

state=6

elif state == 3:

if i=='a':

state=3

elif i=='b':

state=2

else:

state=6

elif state == 4:

if i=='b':

state=5

else:

state=6

elif state == 5:

if i=='b':

state=2

else:

state=6

else :

break

if state == 6:

print(f'{expression} is not recognized')

elif state == 0 or state == 1 or state == 3:

print(f'{expression} is accepted under a\*')

elif state == 5:

print(f'{expression} is accepted under abb')

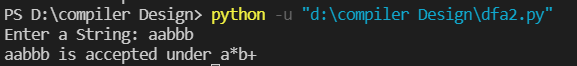
else:

print(f'{expression} is accepted under a\*b+')

string = input("Enter a String: ")

finite\_automata(string)

**OUTPUT:**



**PROGRAM 8**

# AIM: WAP to check whether a given grammar is LL (1) or not.

# THEORY:

# *LL* (1) GRAMMARS AND LANGUAGES. A context-free grammar *G* = (*V*T, *V*N, *S*, *P*) whose parsing table has no multiple entries is said to be *LL* (1). In the name *LL* (1),

* the first *L* stands for scanning the input from **l**eft to right,
* the second *L* stands for producing a **l**eftmost derivation,
* and the 1 stands for using **one** input symbol of lookahead at each step to make parsing action decision.

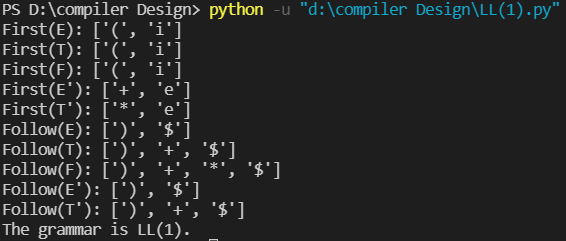
A language is said to be LL (1) if it can be generated by a LL (1) grammar. It can be shown that LL (1) grammars are:

* not ambiguous and
* not left-recursive.

**PROGRAM:**

gram = {  
 "E": ["E+T", "T"],  
 "T": ["T\*F", "F"],  
 "F": ["(E)", "i"]  
}  
  
  
def removeDirectLR(gramA, A):  
 *"""gramA is dictonary"""* temp = gramA[A]  
 tempCr = []  
 tempInCr = []  
 for i in temp:  
 if i[0] == A:  
 tempInCr.append(i[1:] + [A + "'"])  
 else:  
 tempCr.append(i + [A + "'"])  
 tempInCr.append(["e"])  
 gramA[A] = tempCr  
 gramA[A + "'"] = tempInCr  
 return gramA  
  
  
def checkForIndirect(gramA, a, ai):  
 if ai not in gramA:  
 return False  
 if a == ai:  
 return True  
 for i in gramA[ai]:  
 if i[0] == ai:  
 return False  
 if i[0] in gramA:  
 return checkForIndirect(gramA, a, i[0])  
 return False  
  
  
def rep(gramA, A):  
 temp = gramA[A]  
 newTemp = []  
 for i in temp:  
 if checkForIndirect(gramA, A, i[0]):  
 t = []  
 for k in gramA[i[0]]:  
 t = []  
 t += k  
 t += i[1:]  
 newTemp.append(t)  
  
 else:  
 newTemp.append(i)  
 gramA[A] = newTemp  
 return gramA  
  
  
def rem(gram):  
 c = 1  
 conv = {}  
 gramA = {}  
 revconv = {}  
 for j in gram:  
 conv[j] = "A" + str(c)  
 gramA["A" + str(c)] = []  
 c += 1  
  
 for i in gram:  
 for j in gram[i]:  
 temp = []  
 for k in j:  
 if k in conv:  
 temp.append(conv[k])  
 else:  
 temp.append(k)  
 gramA[conv[i]].append(temp)  
  
 # print(gramA)  
 for i in range(c - 1, 0, -1):  
 ai = "A" + str(i)  
 for j in range(0, i):  
 aj = gramA[ai][0][0]  
 if ai != aj:  
 if aj in gramA and checkForIndirect(gramA, ai, aj):  
 gramA = rep(gramA, ai)  
  
 for i in range(1, c):  
 ai = "A" + str(i)  
 for j in gramA[ai]:  
 if ai == j[0]:  
 gramA = removeDirectLR(gramA, ai)  
 break  
  
 op = {}  
 for i in gramA:  
 a = str(i)  
 for j in conv:  
 a = a.replace(conv[j], j)  
 revconv[i] = a  
  
 for i in gramA:  
 l = []  
 for j in gramA[i]:  
 k = []  
 for m in j:  
 if m in revconv:  
 k.append(m.replace(m, revconv[m]))  
 else:  
 k.append(m)  
 l.append(k)  
 op[revconv[i]] = l  
  
 return op  
  
result = rem(gram)  
  
def first(gram, term):  
 a = []  
 if term not in gram:  
 return [term]  
 for i in gram[term]:  
 if i[0] not in gram:  
 a.append(i[0])  
 elif i[0] in gram:  
 a += first(gram, i[0])  
 return a  
  
firsts = {}  
for i in result:  
 firsts[i] = first(result, i)  
 print(f'First({i}):', firsts[i])  
# temp = follow(result,i,i)  
# temp = list(set(temp))  
# temp = [x if x != "e" else "$" for x in temp]  
# print(f'Follow({i}):',temp)  
  
def follow(gram, term):  
 a = []  
 for rule in gram:  
 for i in gram[rule]:  
 if term in i:  
 temp = i  
 indx = i.index(term)  
 if indx + 1 != len(i):  
 if i[-1] in firsts:  
 a += firsts[i[-1]]  
 else:  
 a += [i[-1]]  
 else:  
 a += ["e"]  
 if rule != term and "e" in a:  
 a += follow(gram, rule)  
 return a  
  
  
follows = {}  
for i in result:  
 follows[i] = list(set(follow(result, i)))  
 if "e" in follows[i]:  
 follows[i].pop(follows[i].index("e"))  
 follows[i] += ["$"]  
 print(f'Follow({i}):', follows[i])  
  
flag = 0  
for i in result:  
 for x in firsts[i]:  
 if x in follows[i]:  
 flag = 1  
if flag == 0:  
 print("The grammar is LL(1).")  
else:  
 print("The grammar is not LL(1).")

**OUTPUT:**



**PROGRAM 9**

# AIM: Write a program to compute leading and trailing of a grammar.

**THEORY:**

**LEADING**

If production is of form A → aα or A → Ba α where B is non-terminal, and α can be any string, then the first terminal symbol on R.H.S is

**Leading(A) = {a}**

If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

**TRAILING**

If production is of form A→ αa or A → αaB where B is non-terminal, and α can be any string then,

**TRAILING (A) = {a}**

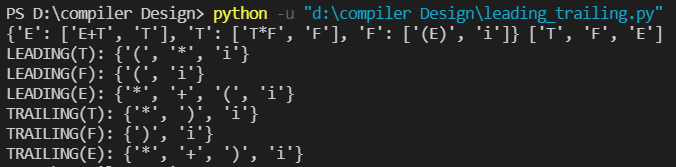
If production is of form A → αB. If a is in TRAILING (B), then a will be in TRAILING (A).

**PROGRAM:**

a = ["E=E+T",  
 "E=T",  
 "T=T\*F",  
 "T=F",  
 "F=(E)",  
 "F=i"]  
  
rules = {}  
terms = []  
for i in a:  
 temp = i.split("=")  
 terms.append(temp[0])  
 try:  
 rules[temp[0]] += [temp[1]]  
 except:  
 rules[temp[0]] = [temp[1]]  
  
terms = list(set(terms))  
print(rules, terms)  
  
  
def leading(gram, rules, term, start):  
 s = []  
 if gram[0] not in terms:  
 return gram[0]  
 elif len(gram) == 1:  
 return [0]  
 elif gram[1] not in terms and gram[-1] is not start:  
 for i in rules[gram[-1]]:  
 s += leading(i, rules, gram[-1], start)  
 s += [gram[1]]  
 return s

def trailing(gram, rules, term, start):  
 s = []  
 if gram[-1] not in terms:  
 return gram[-1]  
 elif len(gram) == 1:  
 return [0]  
 elif gram[-2] not in terms and gram[-1] is not start:  
  
 for i in rules[gram[-1]]:  
 s += trailing(i, rules, gram[-1], start)  
 s += [gram[-2]]  
 return s  
  
leads = {}  
trails = {}  
for i in terms:  
 s = [0]  
 for j in rules[i]:  
 s += leading(j, rules, i, i)  
 s = set(s)  
 s.remove(0)  
 leads[i] = s  
 s = [0]  
 for j in rules[i]:  
 s += trailing(j, rules, i, i)  
 s = set(s)  
 s.remove(0)  
 trails[i] = s  
  
for i in terms:  
 print("LEADING(" + i + "):", leads[i])  
for i in terms:  
 print("TRAILING(" + i + "):", trails[i])

**OUTPUT:**

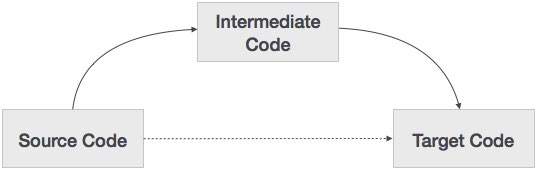


**PROGRAM 10**

# AIM: WAP to generate machine code from the given intermediate code.

**THEORY:**

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



* If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
* Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
* The second part of compiler, synthesis, is changed according to the target machine.
* It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

**PROGRAM:**

fp1\_read = open('intermediate\_txt', 'r')  
lines = fp1\_read.readline()  
while lines:  
 w = lines.split(' ')  
 op = w[0]  
 arg1 = w[1]  
 arg2 = w[2]  
 result = w[3]  
 if op == '+':  
 print("MOV R0,", arg1)  
 print("\nADD R0,", arg2)  
 print("\nMOV", result, ",R0")  
 if op == "\*":  
 print("\nMOV R0,", arg1)  
 print("\nMUL R0, ", arg2)  
 print("\nMOV", result, ",R0")  
 if op == "-":  
 print("\nMOV R0, ", arg1)  
 print("\nSUB R0, ", arg2)  
 print("\nMOV ", result, ",R0")  
 if op == "/":  
 print("\nMOV R0, ", arg1)  
 print("\nDIV R0, ", arg2)  
 print("\nMOV ", result, ",R0")  
 if op == "=":  
 print("\nMOV R0, ", arg1)  
 print("\nMOV", result, ",R0")  
 lines = fp1\_read.readline()  
  
fp1\_read.close()

**INPUT FILE**: intermediate\_code.txt

+ a b t1  
\* c d t2  
- t1 t2 t  
= t ? x

**OUTPUT :**

