**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**Kattankulathur, Chengalpattu District - 603203**



**18CSC304J/ COMPLIER DESIGN**

**MINI PROJECT REPORT**

**INTERMEDIATE CODE GENERATION**

***Guided by:***

*Dr. M. Basker*

**Submitted By:**

Prabhu M (RA2011003010827)

Kongarasan S (RA2011003010856)

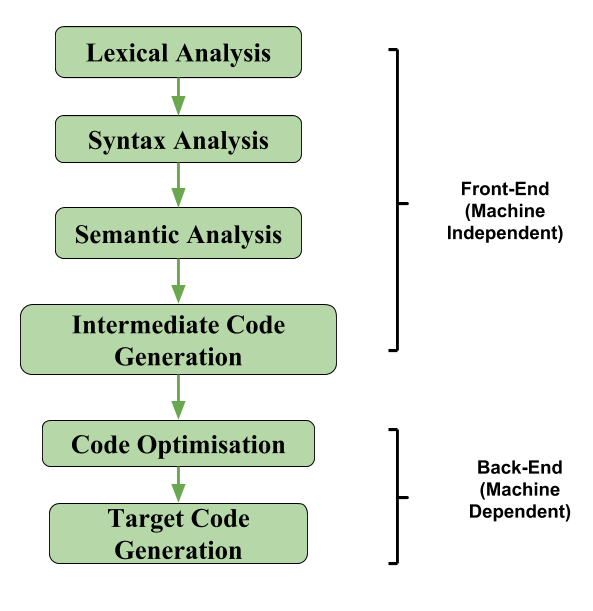
***Aim:-***

To perform the “**Intermediate code Generation”** in compiler design.

***ABSTRACT:-***

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine-independent intermediate code are:

1. Because of the machine-independent intermediate code, portability will be enhanced. For ex, suppose, 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. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
2. Retargeting is facilitated.
3. It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.



If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine-independent intermediate code, we will have only one optimizer. Intermediate code can be either language-specific (e.g., Bytecode for Java) or language. independent (three-address code). The following are commonly used intermediate code representations:

**1. Postfix Notation:**

Also known as reverse Polish notation or suffix notation. The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation, the operator follows the operand.

**Example 1:**

The postfix representation of the expression (a + b) \* c is : ab + c \*

**Example 2:**

The postfix representation of the expression (a – b) \* (c + d) + (a – b) is : ab – cd + \*ab -+

**2.Three-Address Code:**

A statement involving no more than three references(two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as a three address code. Three address statement is of form x = y op z, where x, y, and z will have address (memory location). Sometimes a statement might contain less than three references but it is still called a three address statement.

**Example:**

The three address code for the expression a + b \* c + d : T 1 = b \* c T 2 = a + T 1 T 3 = T 2 + d T 1 , T 2 , T 3 are temporary variables.

There are 3 ways to represent a Three-Address Code in compiler design:

i) Quadruples

ii) Triples

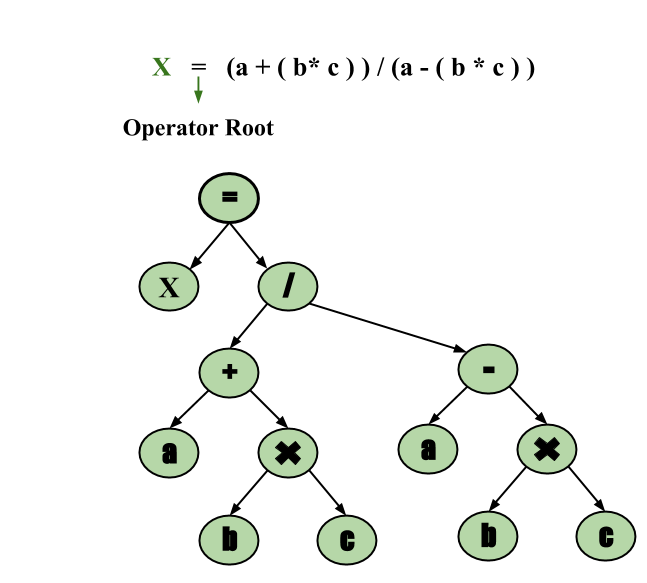
iii) Indirect Triples

**Syntax Tree:**

A syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by the single link in the syntax tree the internal nodes are operators and child nodes are operands. To form a syntax tree put parentheses in the expression, this way it’s easy to recognize which operand should come first.

**Example:**

x = (a + b \* c) / (a – b \* c)



***Advantages of Intermediate Code Generation:***

**Easier to implement:**

Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.

**Facilitates code optimization:**

Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance and efficiency of the generated code.

**Platform independence:**

Intermediate code is platform-independent, meaning that it can be translated into machine code or bytecode for any platform.

**Code reuse:**

Intermediate code can be reused in the future to generate code for other platforms or languages.

**Easier debugging:**

Intermediate code can be easier to debug than machine code or bytecode, as it is closer to the original source code.

***Disadvantages of Intermediate Code Generation:***

**Increased compilation time:**

Intermediate code generation can significantly increase the compilation time, making it less suitable for real-time or time-critical applications.

**Additional memory usage:**

Intermediate code generation requires additional memory to store the intermediate representation, which can be a concern for memory-limited systems.

**Increased complexity:**

Intermediate code generation can increase the complexity of the compiler design, making it harder to implement and maintain.

**Reduced performance:**

The process of generating intermediate code can result in code that executes slower than code generated directly from the source code.

***Requirements to run the script:***

1. Environment to compile and run the code.
2. Import pandas
3. Import copy

***Code:***

print("This is program of THREE ADDRESS CODE GENERATOR using Python.\n\f MADE BY:- Prabhu M, S Kongarasan")

*import* pandas *as* pd

*import* copy

*try*:

    a=pd.read\_csv("input.csv")

    print("\a One thing in this program is that it takes an input of csv file.\n\aThe formate of the csv file is in following manner: ")

    print(a)

    print("\a The output of this program is based on above csv file.\n\aYou can take the different input csv file for diffrent required output.")

    print("\a One thing is to be noticed that in the csv file, there are two columns one is left and other is right for left and right values respectively.")

    print("\a There is only one left side variable for each equation and it may be possible that more than one varible in right side.")

    print("\a The operators and operands which are used in right side must be space separated from each other.")

    print("\a This program is case sensitive, this means that 'd\*10' and '10\*d' are treated as different equation.\nIf you want to solve this problem then you can use the 'CODE OPTIMIZATION TECHNIQUE'.")

    print('\t')

    c=a.shape*# It will gives an tuple of numbers of rows and columns*

*#print(c)*

    l=[]

    o=list("+-\*/")*#If you want to add more operator youn can use that as well*

    o1=[]

    r=[]

*for* i *in* range(c[0]):*# Here c[0] is 0th element of tuple c, which is a.shape (c=a.shape)*

        l=l+[a['left'][i]]

        d=a['right'][i]

        x=d.split()

        l=l+x

*#print(l)*

    sizel=len(l)

*for* z *in* range(sizel):

*#print(sizel)*

*if*(l[z] in o):

            o1=o1+[l[z]]

    o1=list(set(o1))

*#print(o1)*

    li=copy.deepcopy(l)*# if you use li=l then it may occures some un usual error further in program.*

*for* x *in* o1:

*if*(x in li):

            li.remove(x)

    li=list(set(li))

*#print(li)*

*for* b *in* range(len(li)):

        r=r+["R"+str(b)]

*#print(r)*

    i=1

    ak=0

    z=0

    ACounter=0

    akm=[]

*while*(i):

*if*(ak==len(l)):

            i=0

*elif*(l[ak].isalpha() and l[ak]==a['left'][z]):

            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))

            akm=akm+[r[li.index(l[ak])]]

            ak+=1

*elif*(((l[ak].isalpha()) and (l[ak] in a['right'][z]))and (l[ak] not in o1)):

            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))

            akm=akm+[r[li.index(l[ak])]]

            ak+=1

            ACounter+=1

*if*((len(a['right'][z])==1)and (len(akm)==2)):

               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

*#print(akm)*

               akm.clear()

               z+=1

               print("\t")

*elif*((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="+")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("ADD "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

*#print(akm)*

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

*#print(ak)*

*#print(ACounter)*

            ak+=2

            ACounter+=2

*#print(ACounter)*

*if*(len(a['right'][z].split(" "))==ACounter):

*#print(akm)*

                   akm.clear()

                   z+=1

                   ACounter=0

*#print(z)*

                   print("\t")

*elif*((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="-")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("SUB "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            ak+=2

            ACounter+=2

*#print(ACounter)*

*if*(len(a['right'][z].split(" "))==ACounter):

*#print(akm)*

                   akm.clear()

                   z+=1

                   ACounter=0

*#print(z)*

                   print("\t")

*elif*((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="\*")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("MUL "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            ak+=2

            ACounter+=2

*#print(ACounter)*

*if*(len(a['right'][z].split(" "))==ACounter):

*#print(akm)*

                   akm.clear()

                   z+=1

                   ACounter=0

*#print(z)*

                   print("\t")

*elif*((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="/")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("DIV "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            akm.clear()

            ak+=2

            ACounter+=2

*if*(len(a['right'][z].split(" "))==ACounter):

*#print(akm)*

                   akm.clear()

                   z+=1

                   ACounter=0

*#print(z)*

                   print("\t")

*elif*((l[ak].isnumeric())and(l[ak] in a['right'][z])):

            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))

            akm=akm+[r[li.index(l[ak])]]

            ak+=1

            ACounter+=1

*if*((len(akm)==2)and (a['right'][z]==l[ak-1])):

               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

               akm.clear()

               z+=1

               ACounter=0

*#print(z)*

               print("\t")

*elif*((l[ak] not in o1)or (l[ak] not in string.ascii\_lowercase)):

            print("\f Error!\n\f Please enter valid syntax for three address code.\n\f Check your csv file...")

            print(f"\f Error description...\nError in line number {z} and place number {ak}.")

            print(f"\f Error element is {a['right'][z]}.")

*break*

*except* (FileNotFoundError):

    print("Please check you input file. It may possible that file doesn't exist.")

    print("Also check the file name that is given in input section at the starting place.")

*except*(ArithmeticError):

    print("An arithmetic error is caused due to which program is not proceed futher.Please check for the solution.")

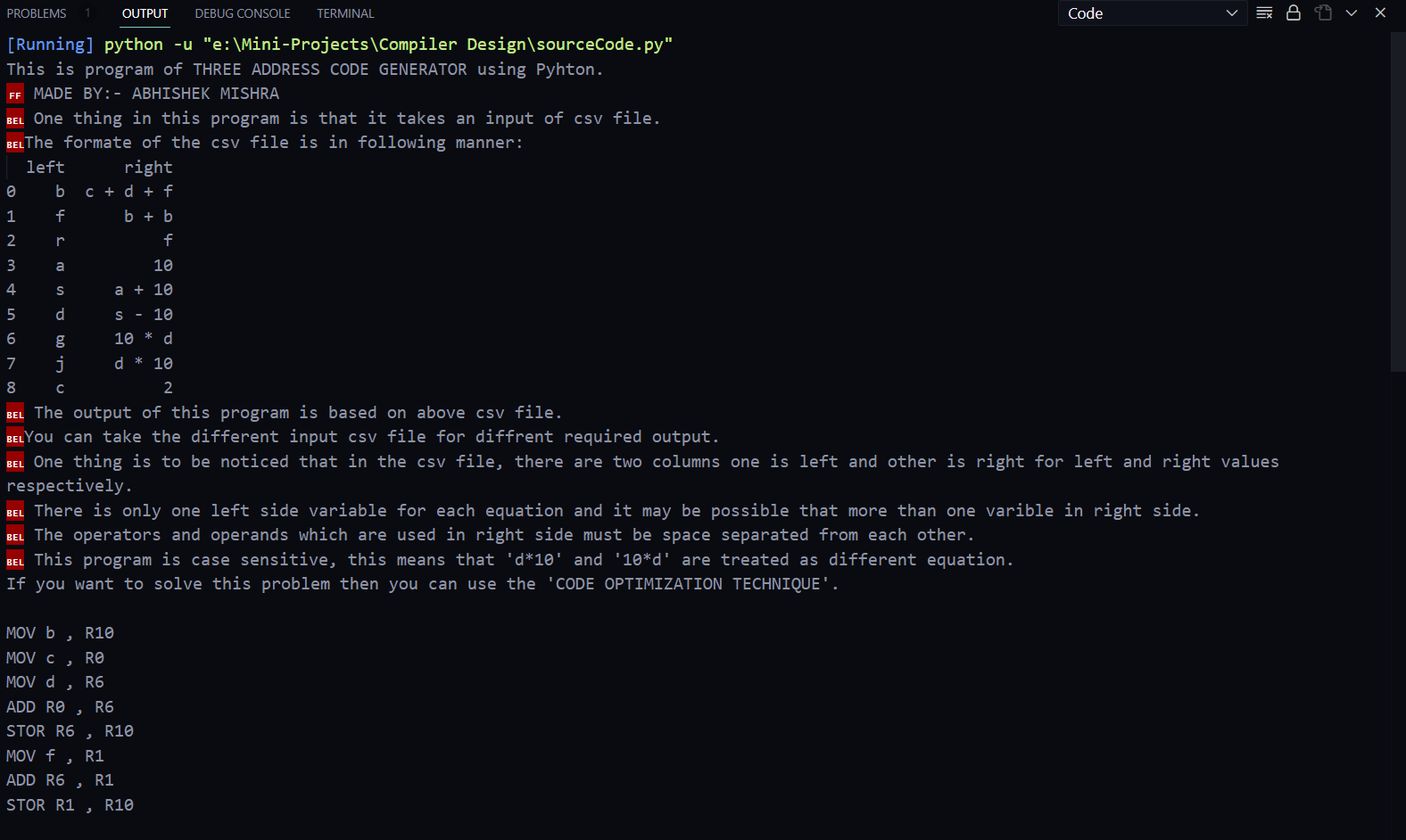
*except*(IndexError):

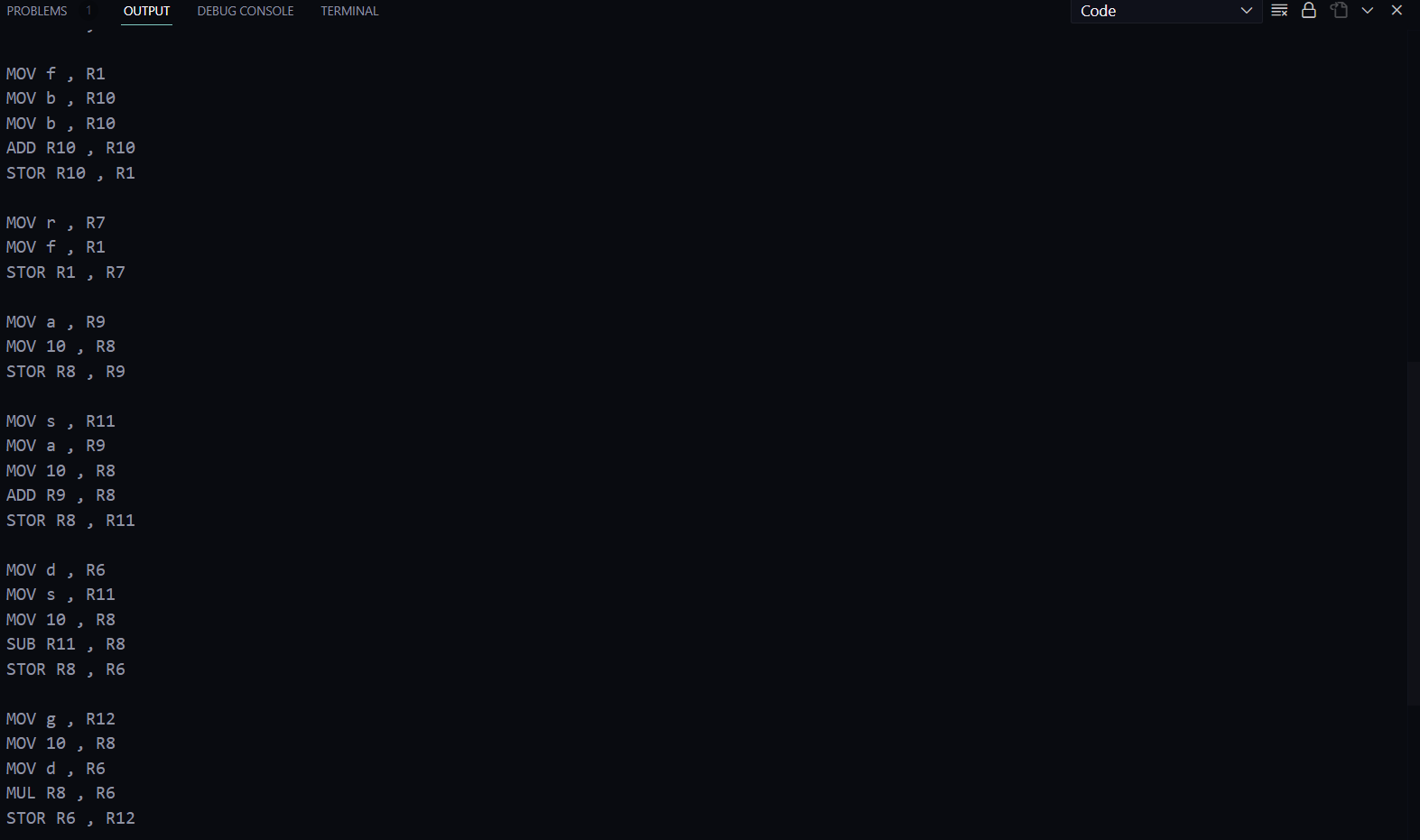
    print("List index out of range.")

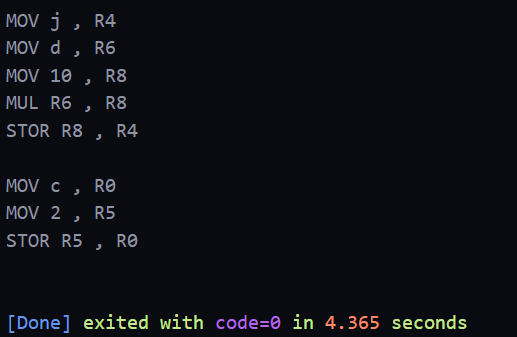
*except*:

    print("An exceptions occurred.")

**Output:**



****



***Result:***

We have successfully implemented the intermediate code generation using python in VSCode (Environment) .