SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

INTERMEDIATE CODE GENERATION

Guided by:

Dr.Ramkumar

Submitted By:

Rohit S (RA2011003010913)

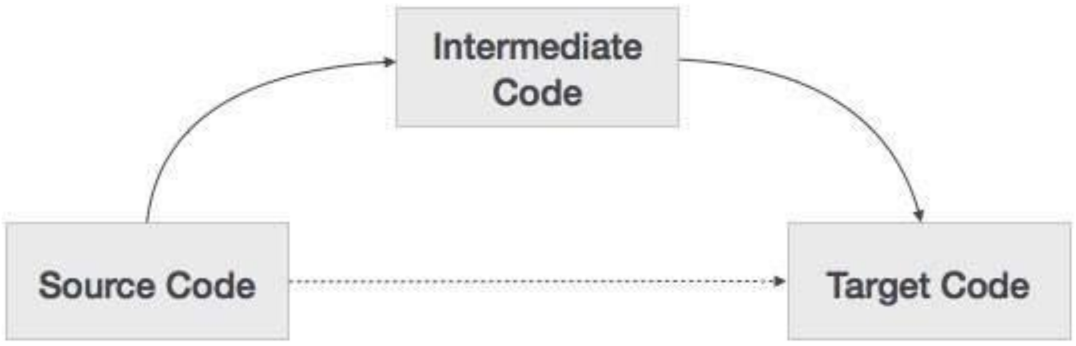
Udhay kumar K(RA2011003010935)

Aim:-

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

ABSTRACT:-

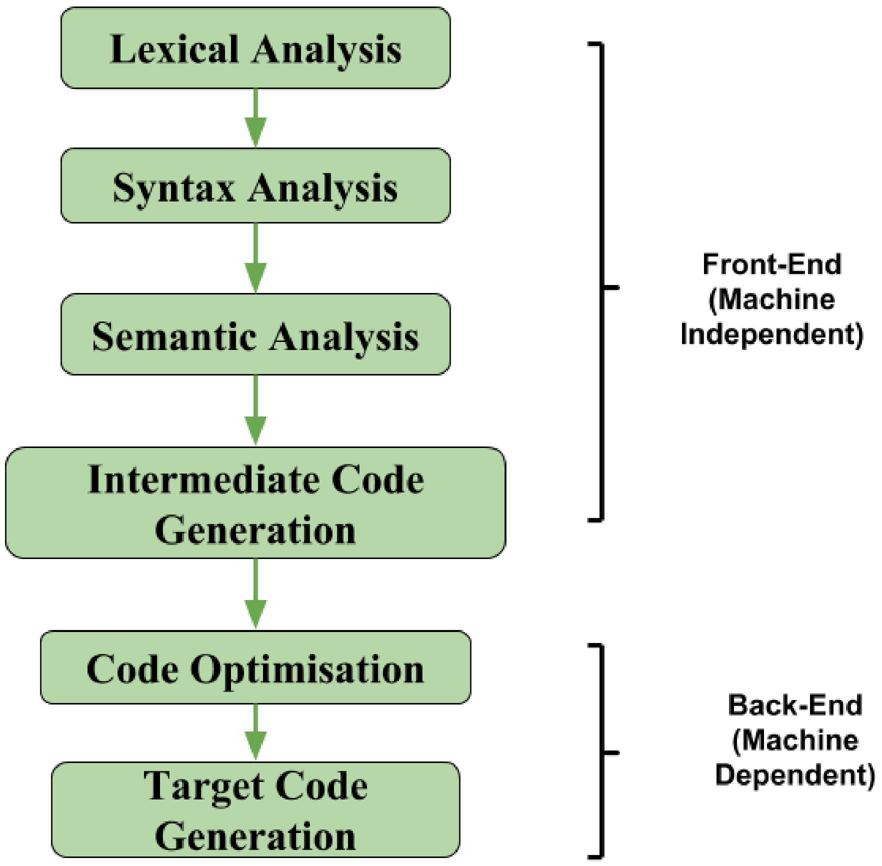
In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine-independent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


Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

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

Quadruples:

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples

format:

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples:

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg1	arg2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples:

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations:

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;
float b;
Allocation process:
{offset = 0}
   int a;
   id.type = int
   id.width = 2

offset = offset + id.width
{offset = 2}
```

float b;
id.type = float
id.width = 4

offset = offset + id.width
{offset = 6}

To enter this detail in a symbol table, a procedure enter can be used. This method may have the following structure:

enter(name, type, offset)

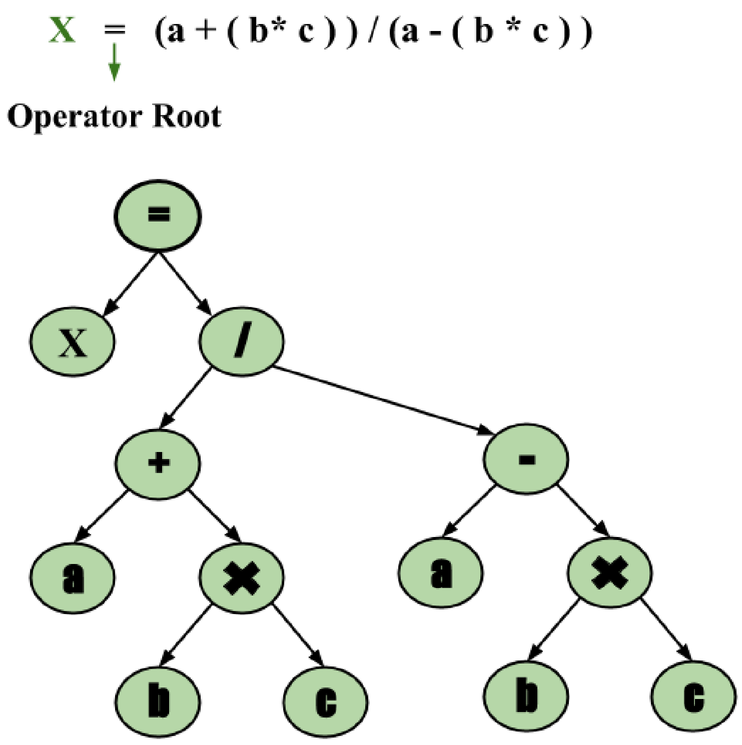
This procedure should create an entry in the symbol table, for variable name, having its type set to type and relative address offset in its data area.

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)



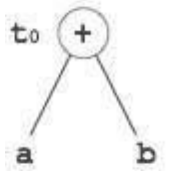
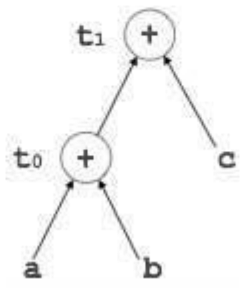
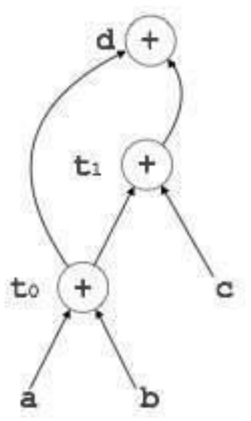
Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

$t_0 = a + b$
$t_1 = t_0 + c$
$d = t_0 + t_1$

 [t_0 = a + b]	 [t_1 = t_0 + c]	 [d = t_0 + t_1]

Code Generator

A code generator is expected to have an understanding of the target machine’s runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

Target language:

The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.

IR Type:

Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.

Selection of instruction:

The code generator takes Intermediate Representation as input and converts (maps) it into target machine’s instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

Register allocation:

A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

Ordering of instructions:

At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Note :

If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg :

Code generator uses getReg function to determine the status of available registers and the location of name values. getReg works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y OP z, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of y OP z is to be saved:

- Call function getReg, to decide the location of L.
- Determine the present location (register or memory) of y by consulting the Address Descriptor of y. If y is not presently in register L, then generate the following instruction to copy the value of y to L:
  MOV y', L
  where y' represents the copied value of y.

- Determine the present location of z using the same method used in step 2 for y and generate the following instruction:
  OP z’, L
  where z’ represents the copied value of z.
- Now L contains the value of y OP z, that is intended to be assigned to x. So, if L is a register, update its descriptor to indicate that it contains the value of x. Update the descriptor of x to indicate that it is stored at location L.
- If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.

Descriptors:

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

Register descriptor :

Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

Address descriptor :

Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

updates the Register Descriptor R1 that has value of x and

updates the Address Descriptor (x) to show that one instance of x is in R1.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

High Level IR –

High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

Low Level IR –

This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

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:

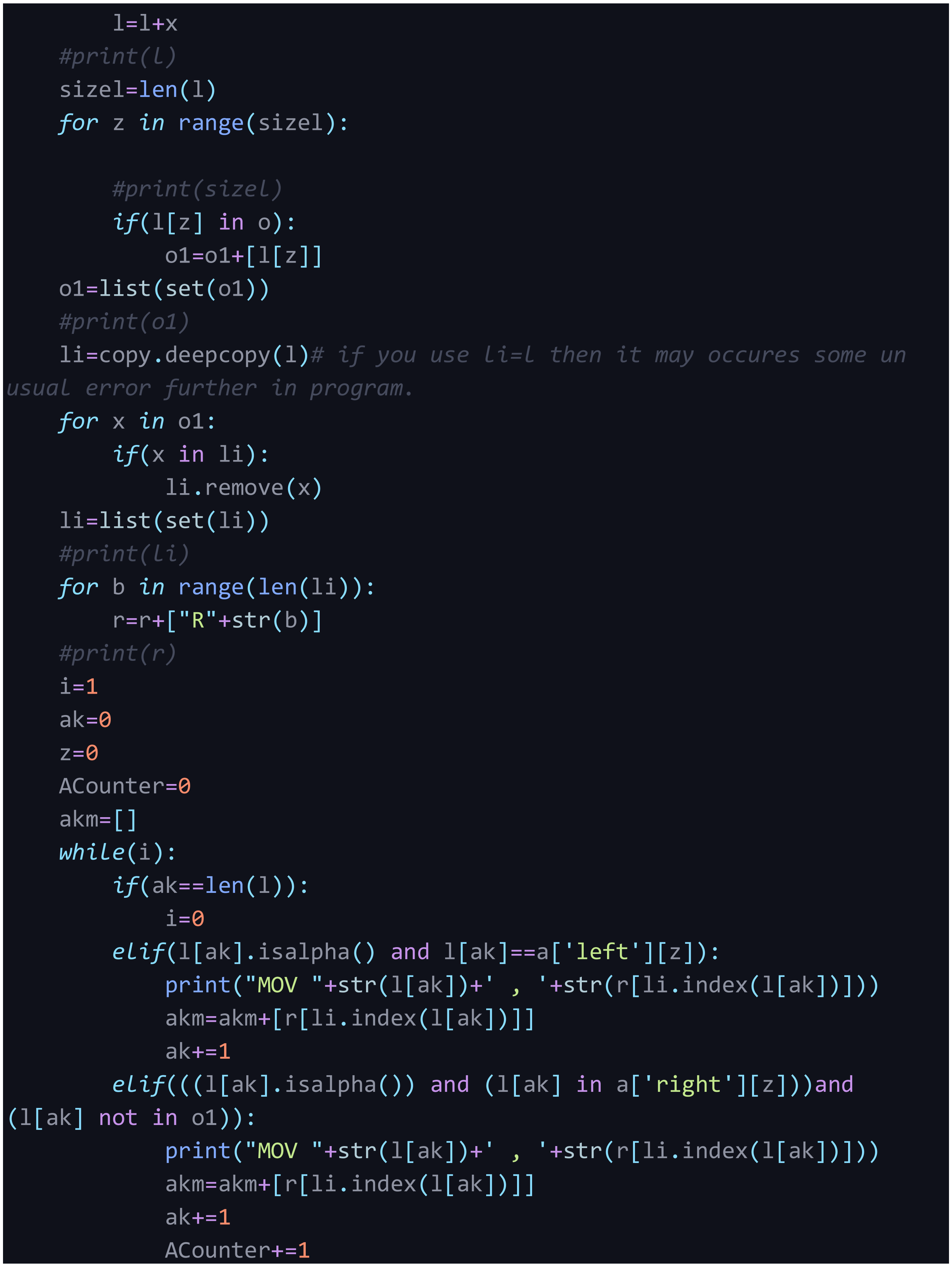
The process of generating intermediate code can result in code that executes slower than code generated directly from the source code.

Requirements to run the script:

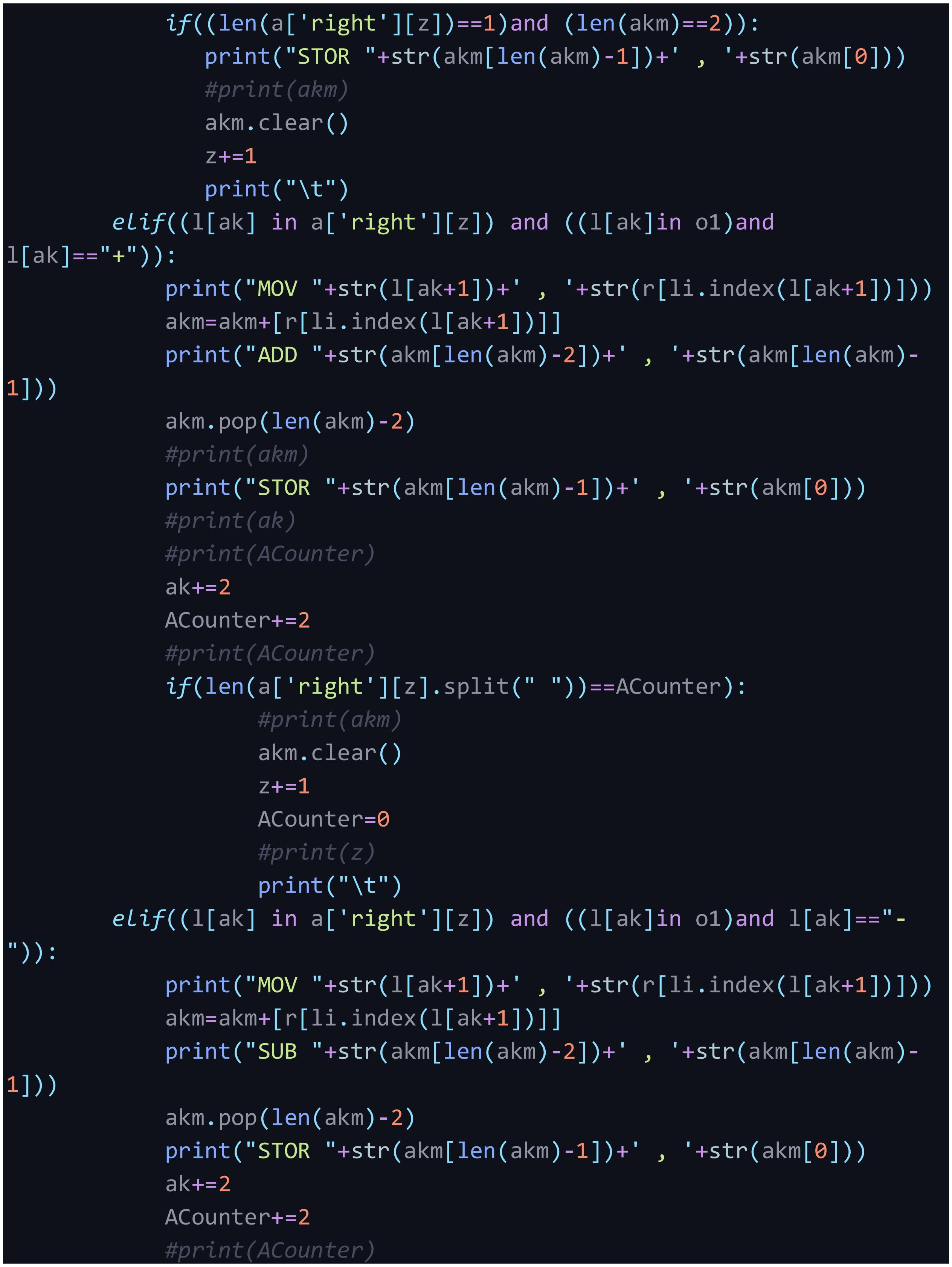
1. Environment to compile and run the code.
2. Import pandas
3. Import copy

Code:

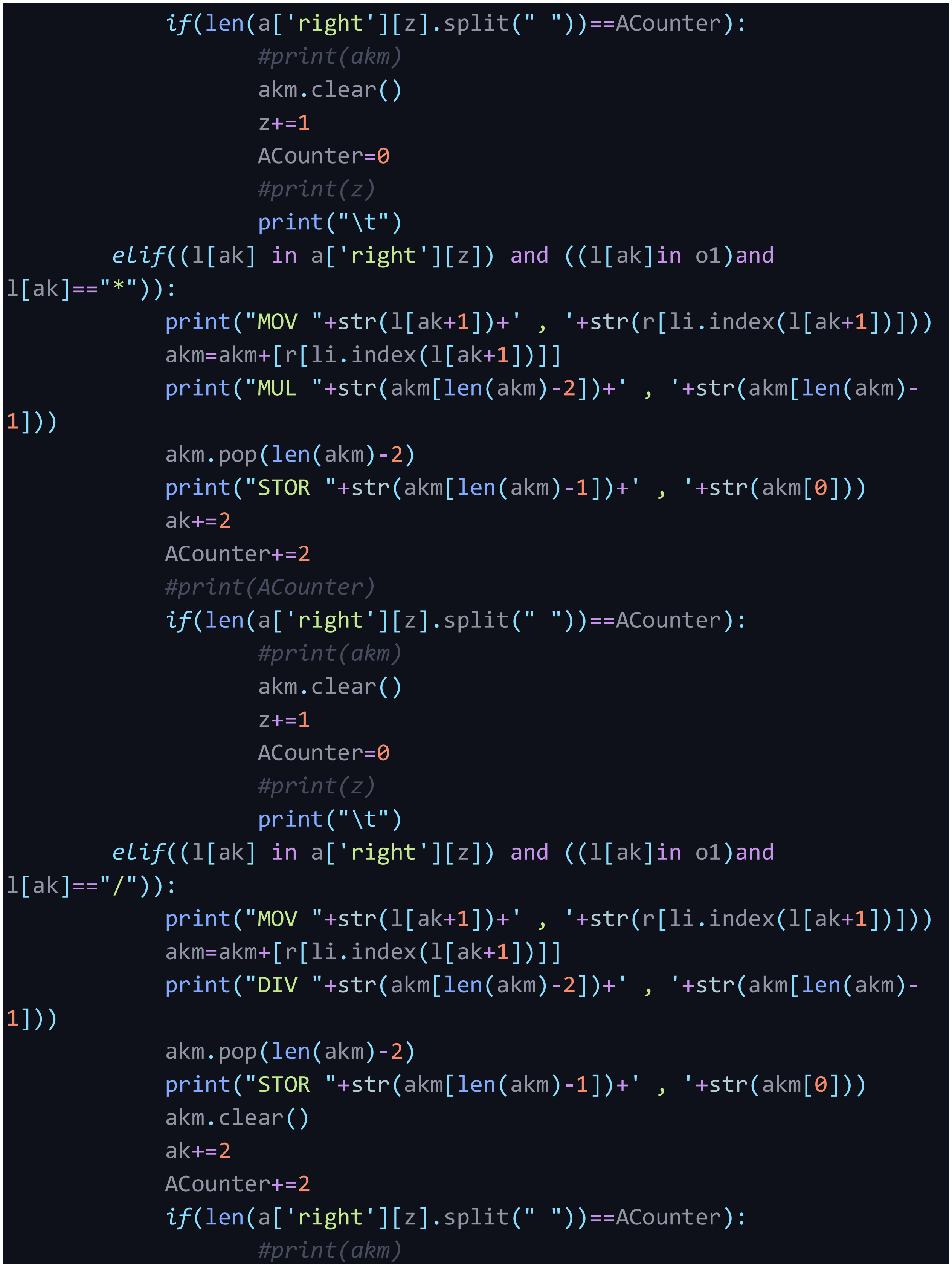
```
     print("This is program of THREE ADDRESS CODE GENERATOR using 
Python.\n\f MADE BY:- Prabhu M, S Kongarasan")
import pandas as pd
import copy
try:
    a=pd.read_csv("input.csv")
    print("\a One thing in this program is that it takes an input of 
csv file.\n\aThe formate of the csv file is in 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
```

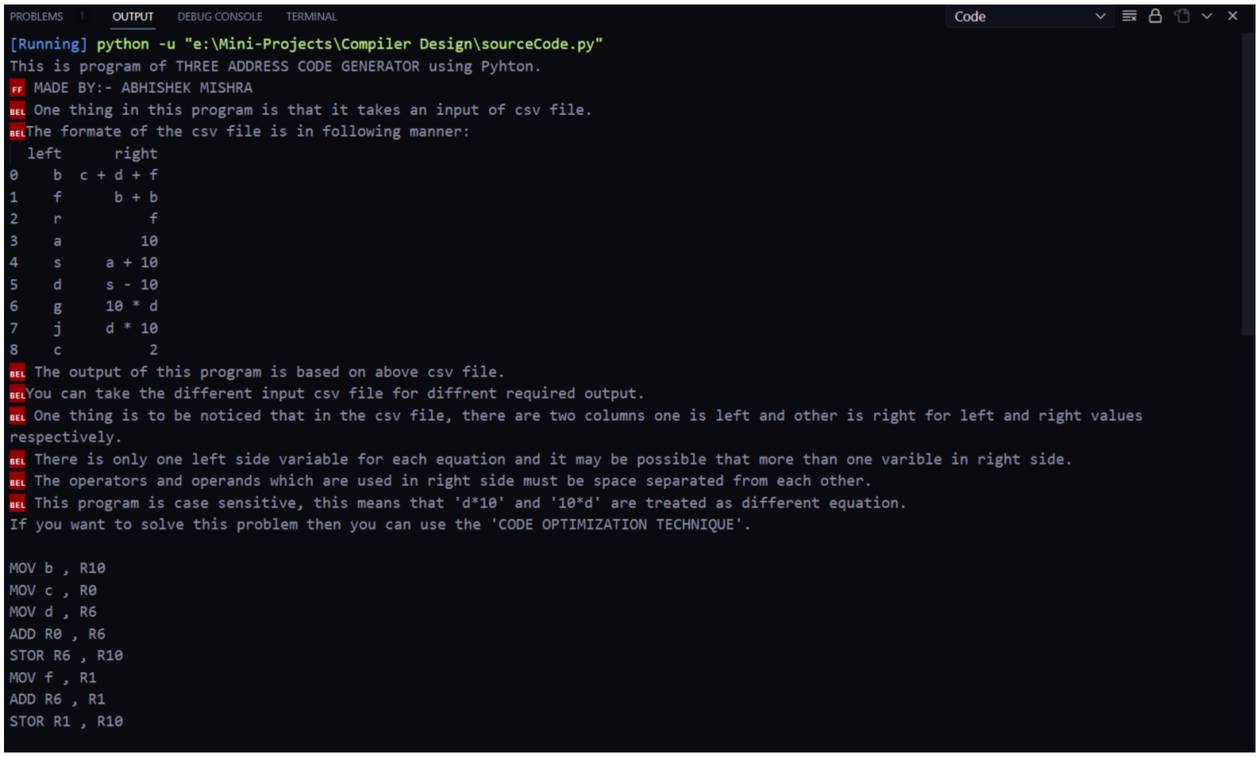


```
            if((len(a['right'][z])==1)and (len(akm)==2)):
               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
               #print(akm)
               akm.clear()
               z+=1
               print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and
l[ak]=="+")):
            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)
```

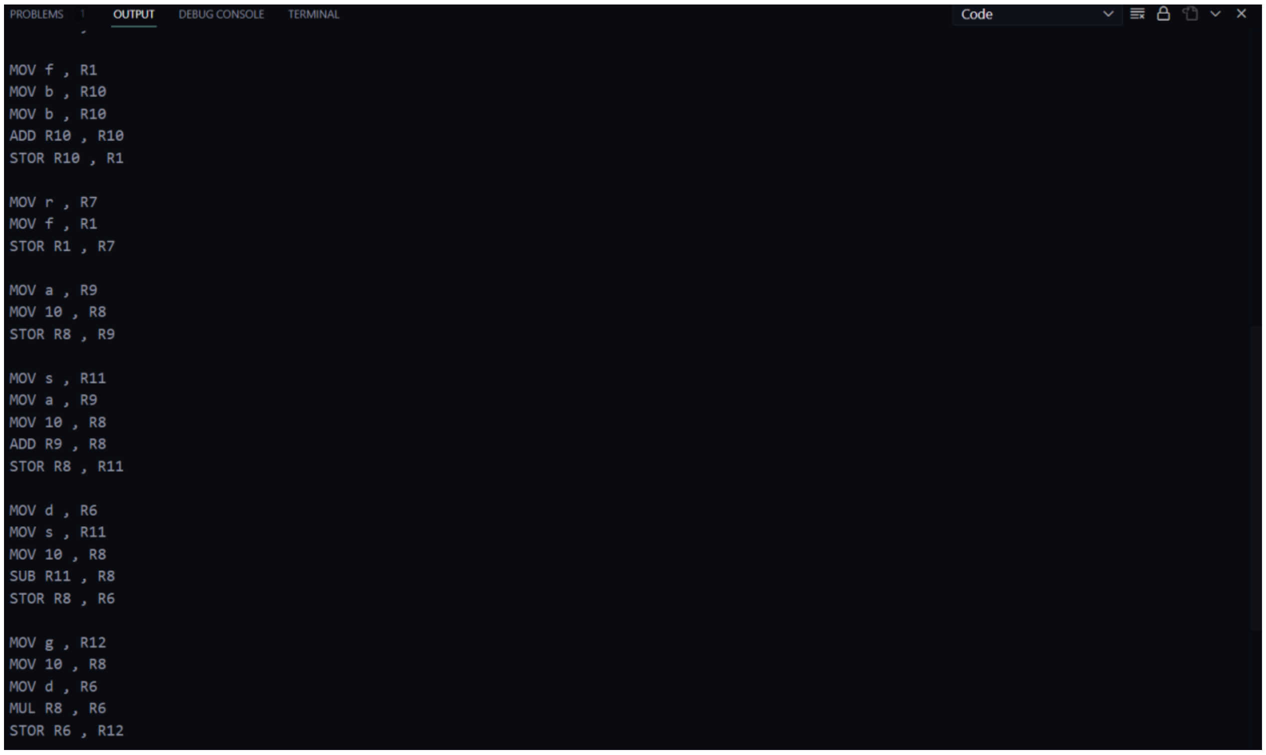


```
            if(len(a['right'][z].split(" "))==ACounter):
                   #print(akm)
                   akm.clear()
                   z+=1
                   ACounter=0
                   #print(z)
                   print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and
l[ak]=="*")):
            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.")
```

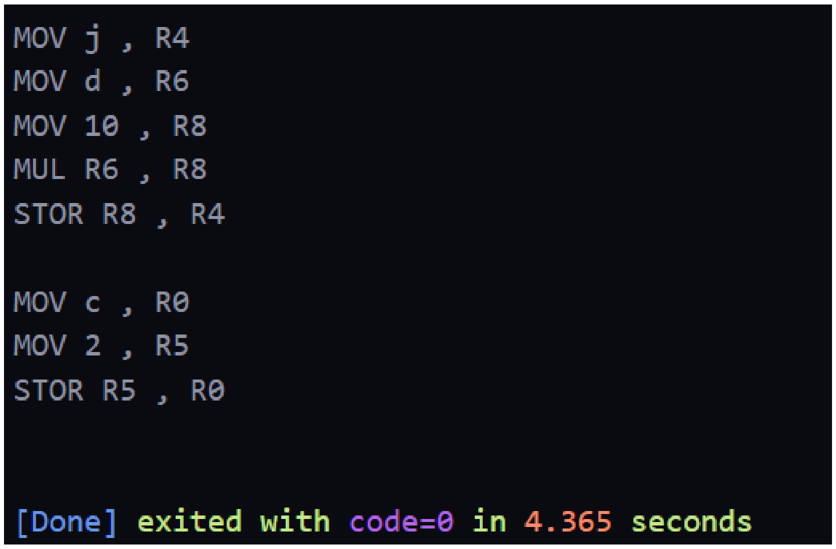
Output:



```
[Running] python -u "e:\Mini-Projects\Compiler Design\sourceCode.py"
This is program of THREE ADDRESS CODE GENERATOR using Pyhton.
 MADE BY:- ABHISHEK MISHRA
 One thing in this program is that it takes an input of csv file.
The formate of the csv file is in following manner:
  left      right
0    b  c + d + f
1    f      b + b
2    r          f
3    a         10
4    s     a + 10
5    d     s - 10
6    g     10 * d
7    j     d * 10
8    c          2
 The output of this program is based on above csv file.
You can take the different input csv file for diffrent required output.
 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.
 There is only one left side variable for each equation and it may be possible that more than one varible in right side.
 The operators and operands which are used in right side must be space separated from each other.
 This program is case sensitive, this means that 'd*10' and '10*d' are treated as different equation.
If you want to solve this problem then you can use the 'CODE OPTIMIZATION TECHNIQUE'.

MOV b , R10
MOV c , R0
MOV d , R6
ADD R0 , R6
STOR R6 , R10
MOV f , R1
ADD R6 , R1
STOR R1 , R10
```



```
MOV f , R1
MOV b , R10
MOV b , R10
ADD R10 , R10
STOR R10 , R1

MOV r , R7
MOV f , R1
STOR R1 , R7

MOV a , R9
MOV 10 , R8
STOR R8 , R9

MOV s , R11
MOV a , R9
MOV 10 , R8
ADD R9 , R8
STOR R8 , R11

MOV d , R6
MOV s , R11
MOV 10 , R8
SUB R11 , R8
STOR R8 , R6

MOV g , R12
MOV 10 , R8
MOV d , R6
MUL R8 , R6
STOR R6 , R12
```



```
MOV j , R4
MOV d , R6
MOV 10 , R8
MUL R6 , R8
STOR R8 , R4

MOV c , R0
MOV 2 , R5
STOR R5 , R0


[Done] exited with code=0 in 4.365 seconds
```

Result:

We have successfully implemented the intermediate code generation using python in Visual Studio Code (Environment).