18CSC304J

COMPILER DESIGN

INTERMEDIATE CODE GENERATOR

MINOR PROJECT REPORT

Submitted by

T Haarish (RA2011003010632)

S Hariharaan (RA2011003010645)

A Ragunath (RA2011003010681)

Under the guidance of

Dr.G.ABIRAMI

for the course

18CSC304J-Compiler Design

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Kancheepuram

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report "**Intermediate Code Generator**" is the bonafide work of T Haarish (RA2011003010632), S Hariharaan (RA2011003010645) and A Ragunath (RA2011003010681) who carried out the project work under my supervision.

SIGNATURE

Dr.G.Abirami

CD Faculty

CSE

SRM Institute of Science and Technology,

Potheri, SRM Nagar, Kattankulathur,

Tamil Nadu 603203

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy**, for his encouragement.

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V.Gopal**, for bringing out novelty in all executions.

We would like to express my heartfelt thanks to **Chairperson, School of Computing Dr. Revathi Venkataraman**, for imparting confidence to complete

my course project

We wish to express my sincere thanks to **Course Audit Professor** for their constant encouragement and support.

We are highly thankful to our Course project Internal guide **Dr.G.Abirami , Compiler Design Faculty , CSE**, for her assistance, timely suggestion and guidance throughout the duration of this course project.

We extend my gratitude to the **Dr. M. PUSHPALATHA, Ph.D HEAD OF THE DEPARTMENT** and my Departmental colleagues for their Support.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project

TABLE OF CONTENT

ABSTRACT :

An intermediate code generator is a software tool that transforms high-level source code into an intermediate language representation. This intermediate representation is typically a simplified version of the original code, and is designed to be more easily translated into executable machine code. The process of generating intermediate code involves several stages, including lexical analysis, syntax analysis, semantic analysis, and code generation. The resulting intermediate code can be optimized to improve the efficiency and performance of the final executable program.

INTRODUCTION :

An intermediate code generator is a compiler component that transforms source code into an intermediate representation that is easier to optimize and convert into machine code. Intermediate code serves as an intermediate step in the compilation process between source code and machine code.

The intermediate code generated by the intermediate code generator is typically a low-level, machine-independent code that abstracts away from the syntax and semantics of the source language. This intermediate representation typically includes data structures and control flow constructs such as variables, expressions, control statements, and function calls.

The benefits of using an intermediate code generator include easier optimization of the code, better portability across different platforms and architectures, and more efficient code generation. By using an intermediate representation, the compiler can perform a variety of optimizations on the code, such as dead code elimination, loop unrolling, and common subexpression elimination. Additionally, since the intermediate code is machine-independent, the compiler can generate machine code for different platforms without needing to re-implement the compiler for each platform.

Overall, the intermediate code generator plays a critical role in the compilation process, providing a powerful tool for transforming source code into an optimized, machine-independent intermediate representation that can be easily converted into machine code for execution on a wide range of platforms.

METHODOLOGY/TECHNIQUES :

The intermediate code generator plays a crucial role in the overall compilation process, and it is important to choose an appropriate methodology for its implementation. Here are some steps to follow when developing an intermediate code generator:

1. Define the intermediate representation: The first step is to define the intermediate representation that the code generator will use. This representation should be simple and easy to work with, yet powerful enough to capture all of the necessary information from the source code.

2. Parse the source code: The next step is to parse the source code and generate an abstract syntax tree (AST). The AST represents the structure of the code and provides a convenient way to traverse and analyze the code.

3. Generate the intermediate code: Once the AST has been generated, the intermediate code can be generated by traversing the AST and emitting intermediate code for each node. This process should take into account any optimizations that can be performed at this stage.

4. Optimize the intermediate code: After the intermediate code has been generated, it should be optimized to improve performance and reduce code size. There are many optimization techniques that can be applied at this stage, including constant folding, dead code elimination, and loop unrolling.

5. Generate target code: Finally, the optimized intermediate code can be translated into target code for the specific platform and architecture. This may involve further optimizations and transformations to ensure that the generated code is efficient and correct.

When developing an intermediate code generator, it is important to choose a methodology that is flexible and extensible. The generator should be able to handle different programming languages, architectures, and optimization techniques, and should be easy to maintain and debug. Additionally, it should produce code that is both efficient and correct, and should be able to handle complex code structures such as loops and conditionals.

IMPLEMENTATION :

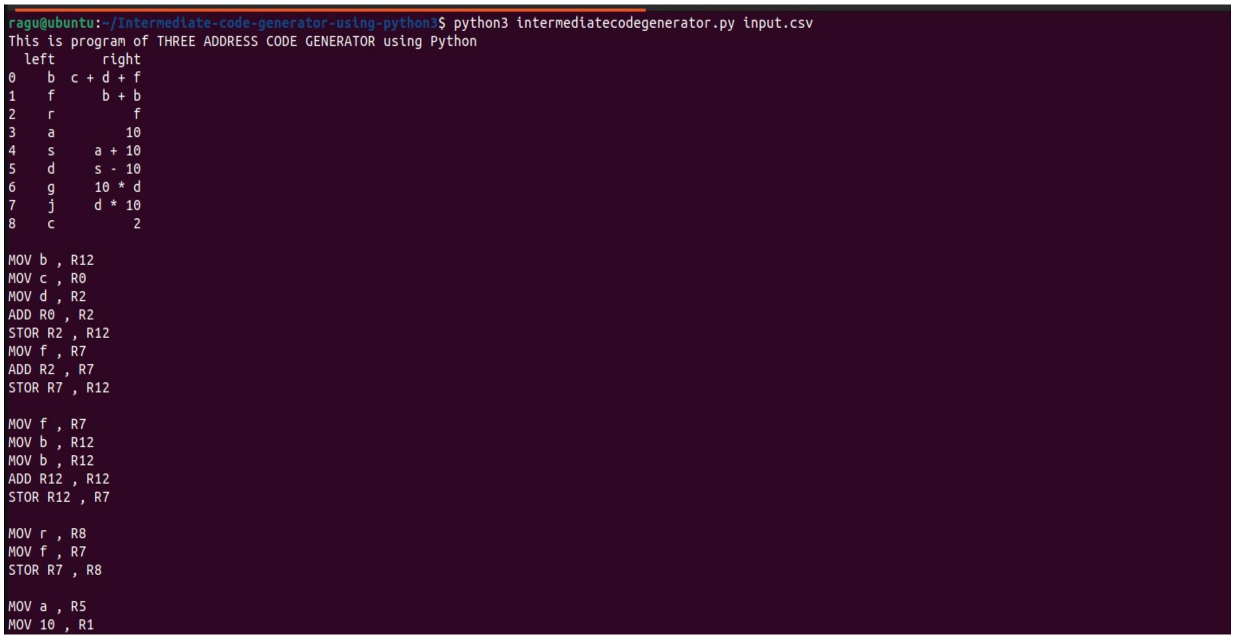
```
print("This is program of THREE ADDRESS CODE GENERATOR using Python")
import pandas as pd
import copy
try:
    a=pd.read_csv("input.csv")
    print(a)
    print('\t')
    c=a.shape# It will gives an tuple of numbers of rows and columns
    #print(c)
    l=[]
    o=list("+-*/")#If you want to add 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.")
```



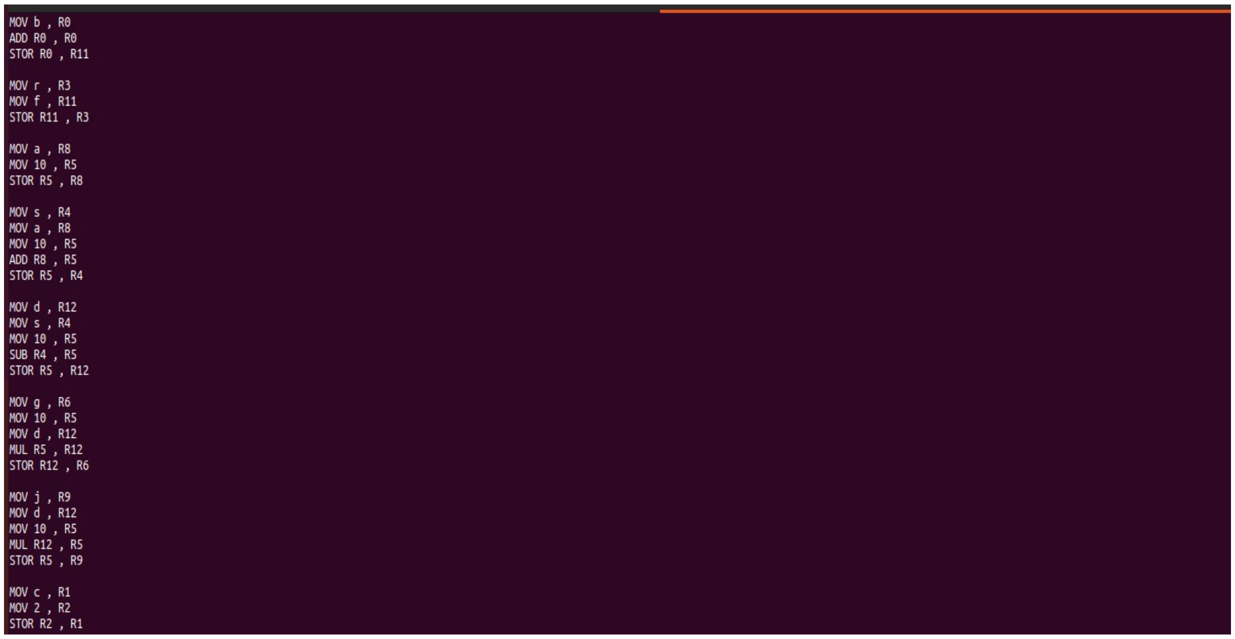
```
ragu@ubuntu:~/Intermediate-code-generator-using-python3$ python3 intermediatecodegenerator.py input.csv
This is program of THREE ADDRESS CODE GENERATOR using Python
  left      right
0    b  c + d + f
1    f      b + b
2    r          f
3    a         10
4    s     a + 10
5    d     s - 10
6    g     10 * d
7    j     d * 10
8    c          2

MOV b , R12
MOV c , R0
MOV d , R2
ADD R0 , R2
STOR R2 , R12
MOV f , R7
ADD R2 , R7
STOR R7 , R12

MOV f , R7
MOV b , R12
MOV b , R12
ADD R12 , R12
STOR R12 , R7

MOV r , R8
MOV f , R7
STOR R7 , R8

MOV a , R5
MOV 10 , R1
```



```
MOV b , R0
ADD R0 , R0
STOR R0 , R11

MOV r , R3
MOV f , R11
STOR R11 , R3

MOV a , R8
MOV 10 , R5
STOR R5 , R8

MOV s , R4
MOV a , R8
MOV 10 , R5
ADD R8 , R5
STOR R5 , R4

MOV d , R12
MOV s , R4
MOV 10 , R5
SUB R4 , R5
STOR R5 , R12

MOV g , R6
MOV 10 , R5
MOV d , R12
MUL R5 , R12
STOR R12 , R6

MOV j , R9
MOV d , R12
MOV 10 , R5
MUL R12 , R5
STOR R5 , R9

MOV c , R1
MOV 2 , R2
STOR R2 , R1
```

RESULT :

Therefore, we have successfully implemented Intermediate code generator.

CONCLUSION :

Intermediate code can translate the source program into the machine program. Intermediate code is generated because the compiler can't generate machine code directly in one pass. It gets its input from the semantic analysis phase and serves its output to the code optimizer phase. The intermediate code generator generates some intermediate representation. And from this intermediate representation, the compiler generates the target code.

REFERENCES :

- https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/
- https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generation.htm
- https://www.javatpoint.com/compiler-design-intermediate-code-generation
- http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/11/Slides11.pdf