Name – Bhargav Shamuvel Gurav | **PRN** – 2041009

Class – L.Y. B-Tech (Computer) | **Batch** – B1

Course Code – CO406U | **Course Name** - CDL

Practical no. B (2)

Aim: Write a C program to generate machine code from abstract syntax tree generated by the parser.

Theory :

Generating machine code from an abstract syntax tree (AST) is a key step in the compilation process of a programming language. The AST represents the structure and semantics of the source code, and translating it into machine code involves multiple steps. Here's a high-level overview of the process:

1. Traverse the AST:

The first step is to traverse the AST in a bottom-up or top-down manner, depending on the specific implementation. During traversal, you visit each node in the tree and generate code for that part of the program. The traversal typically follows the order of operations defined in the source code.

2. Type Checking and Semantic Analysis:

Before generating machine code, you'll need to perform type checking and semantic analysis. This ensures that the program is semantically valid and adheres to the rules of the programming language. Type checking involves verifying that the operations and data types are compatible.

3. Symbol Resolution:

Resolve references to variables, functions, and other symbols. The symbol table is used to map variable names to memory addresses, and function names to their entry points in memory.

4. Code Generation:

At each node of the AST, you generate the corresponding machine code. This involves mapping high-level language constructs to machine instructions. For example, if you have an assignment statement in the AST, you generate code to

store a value in a memory location. The specific machine code instructions will depend on the target architecture (e.g., x86, ARM, MIPS, etc.).

For example, consider the following C code:
int x = 5;
The corresponding AST node for the assignment might generate machine code like:

mov [memory_location_of_x], 5

This is a simplified example, and in reality, code generation can be much more complex, involving optimizations, stack management, register allocation, and handling control flow constructs.

5. **Optimizations:**
After generating the initial code, there are often optimization passes that can improve the performance and size of the resulting machine code. Common optimization techniques include constant folding, dead code elimination, and register allocation.

6. Output:
The generated machine code is typically written to an object file or an executable file, depending on the target platform and the build process.
It's important to note that the exact process and the instructions generated depend on the compiler, the source language, the target architecture, and the desired level of optimization. Building a full-fledged compiler is a complex task, but this overview gives you a sense of the major steps involved in generating machine code from an AST.

Program Code:
```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct quadraple
{
 int pos;
 char op;
```

```
 char arg1[5];
 char arg2[5];
 char result[5];
}quad[15];
int n=0;
void assignment(int);
void uminus(int );
void explore();
void codegen(char op[5],int);
char tuple[15][15];
int main(void)
{
 FILE *src;
 int nRetInd,i;
 char str[15];
 src=fopen("code.txt","r");
 fscanf(src,"%s",str);
 while(!feof(src))
 {
 strcpy(tuple[n++],str);
 fscanf(src,"%s",str);
 }
 printf("INPUT:\nIntermediate codes:\n");
 for(i=0;i<n;i++)
 printf("%s\n",tuple[i]);
 explore();
 getch();
 printf("OUTPUT:\n");
 printf("Quadruple: \n");
 printf("pos\topr\targ1\targ2\tresult\n");
 for(i=0;i<n;i++)

printf("\n%d\t%c\t%s\t%s\t%s",quad[i].pos,quad[i].op,quad[i].arg1,quad[i].arg2,qu
ad
[i].result);
 i=0;
 printf("\n\ncode generated :\n");
```

```
 while(i<n)
 {
 if(quad[i].op=='+')
 codegen("ADD",i);
 if(quad[i].op=='=')
 assignment(i);
 if(quad[i].op=='-')
 if(!strcmp(quad[i].arg2,"\0"))
 uminus(i);
 else
 codegen("SUB",i);
 if(quad[i].op=='*')
 codegen("MUL",i);
 if(quad[i].op=='/')
 codegen("DIV",i);
 i++;
 }
 getch();
 return 0;
}
void codegen(char op[5],int t)
{
 char str[25];
 printf("MOV %s,R0\n",quad[t].arg1);
 printf("%s %s,R0\n",op,quad[t].arg2);
 printf("MOV R0,%s\n",quad[t].result);
}
void assignment(int t)
{
 char str[25];
 printf("MOV %s,%s\n",quad[t].arg1,quad[t].result);
}
void uminus(int t)
{
 char str[25];
 printf("MOV R0,0\n");
 printf("SUB %s,R0\n",quad[t].arg1);
```

```
 printf("MOV R0,%s\n",quad[t].result);
}
void explore()
{
 int i,j,t,t1,t2;
 for(i=0;i<n;i++)
 {
 quad[i].pos=i;
 for(j=0,t=0;j<strlen(tuple[i])&&tuple[i][j]!='=';j++)
 {
 quad[i].result[t++]=tuple[i][j];
 }
 t1=j;
 quad[i].result[t]='\0';
 if(tuple[i][j]=='=')
 {
 quad[i].op='=';
 }
 if(tuple[i][j+1]=='+'||tuple[i][j+1]=='-'||tuple[i][j+1]=='*'||tuple[i][j+1]=='/')
 {
 quad[i].op=tuple[i][j+1];
 t1=j+1;
 }
 for(j=t1+1,t=0;j<strlen(tuple[i])&&tuple[i][j]!='+'&&tuple[i][j]!='-
'&&tuple[i][j]!='*'&&tuple[i][j]!='/';j++)
 {
 quad[i].arg1[t++]=tuple[i][j];
 }
 t2=j;
 quad[i].arg1[t]='\0';
 if(tuple[i][j]=='+'||tuple[i][j]=='-'||tuple[i][j]=='*'||tuple[i][j]=='/')
 {
 quad[i].op=tuple[i][j];
 }
 for(j=t2+1,t=0;j<strlen(tuple[i]);j++)
 {
 quad[i].arg2[t++]=tuple[i][j];
```

```
 }
 quad[i].arg2[t]='\0';
 }
}
```

code.txt

Intermiate codes:
t0=c*d
t1=t0
c=a/v

Output:

Conclusion : In this practical we learnt how the machine code is generated from a syntax tree by the parser.