Academic Year: 2024-25
Semester: VI
Class : TE (Div-A) Subject: SPCCL

Name: Krushnali Biradar
ID:22102039
Roll No.: 15

Experiment6

AIM: To study and implement Code generation phase of compiler

THEORY: 1) Definition: Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.

2) 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.

Code:

```
#include <stdio.h>
#include <string.h>

struct code {
   char nemo[4];
   char op1[3];
   char op2[3];
};

int getreg() {
   static int r = 48; // ASCII '0'
   return r++;
}

int main() {
   char stmt[4][6] = {"T=A-B", "U=A-C", "V=T+U", "W=V+U"};
   struct code c[7];
   char add_dis[10][4], op; // Increased column size to 4
   int i, cp = 0, reg, j = 0, flag;

   for (i = 0; i < 4; i++) {
      printf("\n%s", stmt[i]);
```

```
op = stmt[i][3];
flag = 0;

switch (op) {
   case '-':
      reg = getreg();
      strcpy(c[cp].nemo, "MOV");
      c[cp].op1[0] = stmt[i][2];
      c[cp].op1[1] = '\0';
      c[cp].op2[0] = 'R';
      c[cp].op2[1] = reg;
      c[cp].op2[2] = '\0';
      printf("\n%s\t%s\t%s", c[cp].nemo, c[cp].op1, c[cp].op2);
      cp++;

      strcpy(c[cp].nemo, "SUB");
      c[cp].op1[0] = stmt[i][4];
      c[cp].op1[1] = '\0';
      c[cp].op2[0] = 'R';
      c[cp].op2[1] = reg;
      c[cp].op2[2] = '\0';
      printf("\n%s\t%s\t%s", c[cp].nemo, c[cp].op1, c[cp].op2);

      // Assign Address Descriptor
      add_dis[j][0] = stmt[i][0];
      add_dis[j][1] = 'R';
      add_dis[j][2] = reg;
      add_dis[j][3] = '\0';
      printf("\nAddress Descriptor of %c is %c%c", add_dis[j][0], add_dis[j][1], add_dis[j][2]);
      j++;
      cp++;
      break;

   case '+':
      strcpy(c[cp].nemo, "ADD");

      for (j = 0; add_dis[j][0] != stmt[i][4]; j++);
      c[cp].op1[0] = 'R';
      c[cp].op1[1] = add_dis[j][2];
      c[cp].op1[2] = '\0';

      for (j = 0; add_dis[j][0] != stmt[i][2]; j++);
      c[cp].op2[0] = 'R';
      c[cp].op2[1] = add_dis[j][2];
      c[cp].op2[2] = '\0';

      printf("\n%s\t%s\t%s", c[cp].nemo, c[cp].op1, c[cp].op2);

      // Assign Address Descriptor
      add_dis[j][0] = stmt[i][0];
      add_dis[j][1] = 'R';
      add_dis[j][2] = reg;
      add_dis[j][3] = '\0';
      printf("\nAddress Descriptor of %c is %c%c", add_dis[j][0], add_dis[j][1], add_dis[j][2]);
```

```
            cp++;

            if (i == 3) {
               strcpy(c[cp].nemo, "MOV");
               c[cp].op1[0] = 'R';
               c[cp].op1[1] = add_dis[j][2];
               c[cp].op1[2] = '\0';
               c[cp].op2[0] = stmt[i][0];
               c[cp].op2[1] = '\0';
               printf("\n%s\t%s\t%s", c[cp].nemo, c[cp].op1, c[cp].op2);
            }
            break;
      }
   }
   return 0;
}
```

Output: