Name: Gaurav Thakkar

Class: CSE 1

Batch: B

Roll No : 2183505

Experiment Number: 9

TITLE: Code generation using DAG / labeled tree.0

PROBLEM STATEMENT:

Implement Code generation using DAG / labeled tree.

OBJECTIVE:

1. To understand working of Code Generation Phase of Compiler.

THEORY:

Issues in Code Generation Phase :

Following generic issues are concerned while designing code generator.

(1) Input to code generator
(2) Target program
(3) Memory management
(4) Instruction selection
(5) Register allocation
(6) Choice of evaluation order
(7) Approaches to code generation.

Let us see one by one how these issues are concerned with design of code generation phase of the compiler.

Input to Code Generation Phase (or Input to Code Generator) :

We know that input to code generator is the output of 'Intermediate Code Generator'

Now output of intermediate code generator phase is:

(a) Intermediate Representation (Intermediate code) :

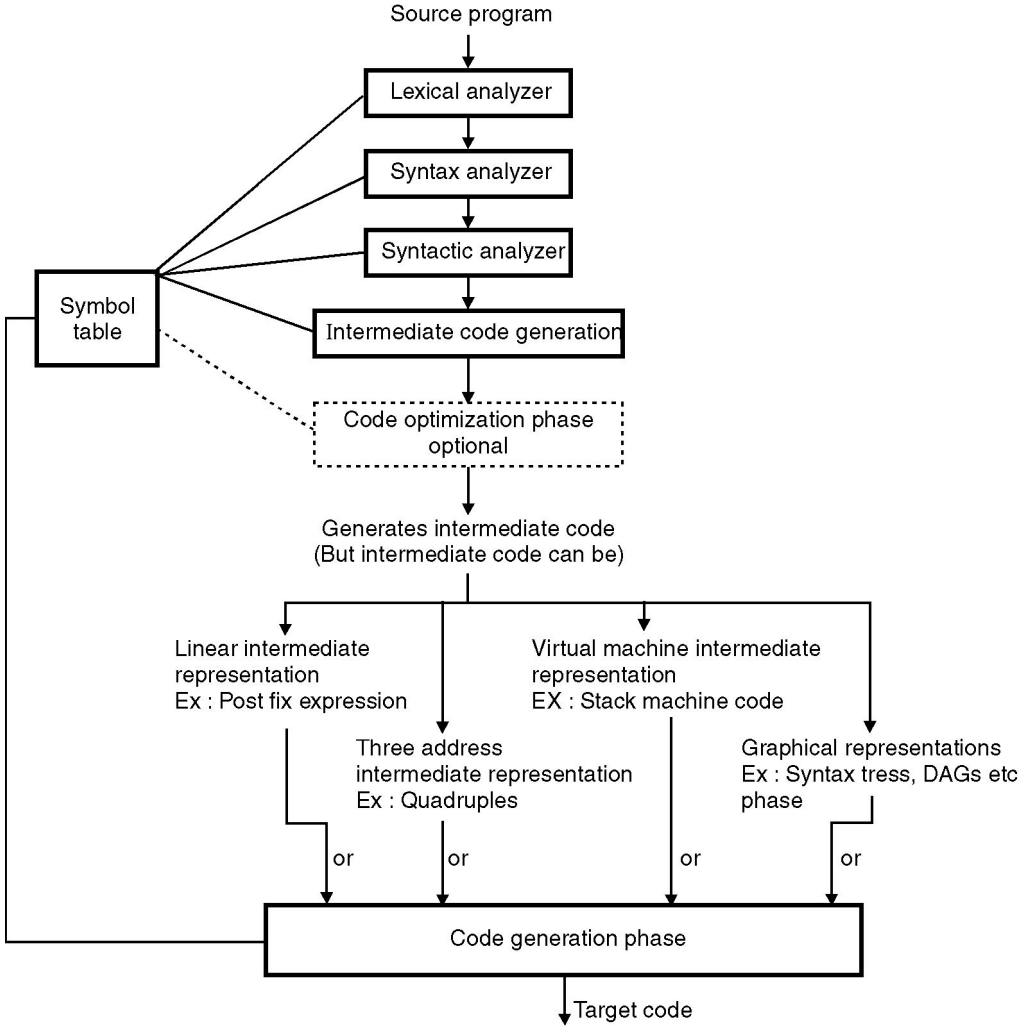
This intermediate code is generated by front end of compiler.

Intermediate code produced by intermediate phase can be any of the following as

- Qudruples
- Triples

(b) Information in Symbol Table:

This information is used to determine runtime address of all data objects which are specified by their names in the input of code generation phase i.e. Intermediate code.



One of possible Intermediate Representations.

Fig.: Input to code generator

Thus, before code generation phase, source program is processed by Lexical, Syntax, Semantic Analyzers, Intermediate code generator and may or may not be Code Optimizer. Because code optimization can be implemented before and/or after and/or during code generation phase of the compiler.

Thus, while designing code generator it is assumed that

(i) Source program is already analysed, parsed and some equivalent intermediate code is generated.
(ii) Type checking is already performed.
(iii) Intermediate code has some additional type conversion functions.
(iv) All syntactic and semantic errors are detected.
(v) Intermediate code given as input to Code Generation phase is error free.

As shown in Fig.. intermediate code generated by intermediate code generation phase can be either of a possible types of intermediate representations as

(a) Linear representation as postfix expressions.
(b) Three address intermediate representation as quadruples
(c) Virtual machine representation as stack machine code.
(d) Graphical intermediate representation as parse tree pr DAGS.

Now code generator algorithms are different for different intermediate representation of source program i.e. code generator algorithms has to be designed according to intermediate representation generated by intermediate code generator.

Algorithm discussed can be used for three address code, parse or syntax tees. Other algorithms can be used for other intermediate representations.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Pseudo Code:-

LEX:

Execute the program with the following commands.

```
$ yacc –d Codegen.y
$ lex Codegen.l
$cc lex.yy.c y.tab.c –ll –ly -lm
 $.\a.out
```

CODE

LEX:-

```
%{
#include<stdio.h>
#include<string.h>
#include"y.tab.h"
%}
```

```
%%
[a-z][a-zA-Z0-9]* |
[0-9]+                  {
                            strcpy(yylval.vname,yytext);
                            return NAME;
                        }
[ |\t]                  ;
.|\n                    { return yytext[0]; }
%%
```

YACC:-

```
%{
#include<stdlib.h>
#include<stdio.h>
FILE *fpOut,*yyout,*yyin;
%}
%union
{
        char vname[10];
}
%left '+' '-'
%left '*' '/'
%token<vname>NAME
%type<vname>expression
%type<vname>line
%%
input   : line '\n' input
        | '\n' input
        | ;
line    : NAME '=' expression  {
                                fprintf(fpOut,"MOV %s, AX\n",$1);
                               }
        ;
expression: NAME '+' NAME       {
                                fprintf(fpOut,"MOV AX, %s\n",$1);
                                fprintf(fpOut,"ADD AX, %s\n",$3);
                               }
        | NAME '-' NAME       {
                                fprintf(fpOut,"MOV AX, %s\n",$1);
                                fprintf(fpOut,"SUB AX, %s\n",$3);
                               }
        | NAME '*' NAME       {
                                fprintf(fpOut,"MOV AX, %s\n",$1);
                                fprintf(fpOut,"MUL AX, %s\n",$3);
```

```
                        }

        | NAME '/' NAME     {
                            fprintf(fpOut,"MOV AX, %s\n",$1);
                            fprintf(fpOut,"DIV AX, %s\n",$3);
                        }
        | NAME              {
                            fprintf(fpOut,"MOV AX, %s\n",$1);
                            //strcpy($$, $1);
                        }
        ;
%%
FILE *yyin;
main()
{
        FILE *fpInput;
        fpInput = fopen("input1.txt","r");
        if(fpInput=='\0')
        {
                yyerror("file read error");
                exit(0);
        }
        fpOut = fopen("output1.txt","w");
        if(fpOut=='\0')
        {
                yyerror("file creation error");
                exit(0);
        }
        yyin = fpInput;
        yyout=fpOut;
        yyparse();
        fcloseall();
}
yyerror(char *msg)
{
        fprintf(stderr,"%s",msg);
}
```

OUTPUT

```
[gtxtreme@archlinux Assignment9]$ echo "x = 20
> y = b + 40
> t1 = b * c
> t2 = t1 / d
> t3 = a + t2
> t4 = t3 - e
> f = t4
> g = 10">input1.txt
[gtxtreme@archlinux Assignment9]$ flex CodeGeneration.l
[gtxtreme@archlinux Assignment9]$ yacc -d CodeGeneration.y
[gtxtreme@archlinux Assignment9]$ gcc lex.yy.c y.tab.c -lfl -w
[gtxtreme@archlinux Assignment9]$ ./a.out
[gtxtreme@archlinux Assignment9]$ cat output1.txt
MOV AX, 20
MOV x, AX
MOV AX, b
ADD AX, 40
MOV y, AX
MOV AX, b
MUL AX, c
MOV t1, AX
MOV AX, t1
DIV AX, d
MOV t2, AX
MOV AX, a
ADD AX, t2
MOV t3, AX
MOV AX, t3
SUB AX, e
MOV t4, AX
MOV AX, t4
MOV f, AX
MOV AX, 10
MOV g, AX
[gtxtreme@archlinux Assignment9]$
```

FAQ's

Design Issues in Code Generation.

The following are the design issues in Code Generation

(1) Input to code generator

(2) Target program

(3) Memory management
(4) Instruction selection
(5) Register allocation
(6) Choice of evaluation order
(7) Approaches to code generation.

Types of Target Codes that can be generated by Code Generation Phase.

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.

Flow Graph and Basic Block, along with Partition Algorithm.

Basic Blocks-

Basic block is a set of statements that always executes in a sequence one after the other.The characteristics of basic blocks are-They do not contain any kind of jump statements in them.There is no possibility of branching or getting halt in the middle.All the statements execute in the same order they appear.They do not lose lose the flow control of the program.

Flow Graphs-

A flow graph is a directed graph with flow control information added to the basic blocks.

The basic blocks serve as nodes of the flow graph.

There is a directed edge from block B1 to block B2 if B2 appears immediately after B1 in the code.

Three Address Code for the expression a = b + c + d is-

T1= b+c

T2= T1 +d

a=T2

Partition Algorithm for Basic

Blocks

Input

: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement

in exactly one block

1. Determine the set of leaders, the first statements if basic blocks

a) The first statement is the leader

b) Any statement that is the target of a goto is a leader

c) Any statement that immediately follows a goto is a leader

2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Input to the Code Generation like Instruction Selection, Memory Management, Choice of Evaluation Order Instruction Cycle, Instruction Cost etc.

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

Target program –
The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language. Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.
Memory Management –
Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.
Instruction selection –

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

Evaluation order –

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

Conclusion:

In this experiment we learn how to perform Code Generation from an Intermediate Code using Lex and Yacc