Unit - 5

Q1. Issues in code generation

The following issues arise during the code generation phase →

1. Input to code generator
2. Target program
3. Memory ~~alloca~~ management
4. Instruction selection
5. Register allocation
6. Evaluation order.

1. Input to code generator →

The input to code generation consists of intermediate representation of the source program produced by front end, together with information in the symbol table.

Intermediate representation can be:
i) Linear representation (eg. postfix notation)
ii) Graphical representation (eg. syntax trees and dags)
iii) Three address representation (eg. Quadruples)
iv) Virtual machine representation (eg. stack machine code)

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking.

Therefore, input to the code generator is assumed to be error free.

2. Target program →

The output of the code generator is is the target program

The output may be:

i) Absolute machine language: It can be placed in a fixed memory location and can be executed immediately.

ii) Relocatable machine language: It allows subprograms to be compiled separately.

iii) Assembly language: Code generation is made easier

3. Memory management →

Names in the source program are mapped to the addresses of data objects in run-time memory by the front end and code generator

It makes use of symbol table

A name in the three address statement refers to a symbol table entry for the name.

4. Instruction selection →

The instructions of the target machine should be complete and uniform.

Instruction speeds and machine idioms are important factors in the efficiency of the target program.

The quality of the generated target code may be determined by speed and size.

eg. $x = y + z$ ⟹ three address statement form

Code sequence generated ⟹

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
```

Unfortunately this kind of statement by statement code generation often produces poor code

eg.

```
a = b+c
d = a+e
```

↓

```
MOV  b, R0
ADD  c, R0
MOV  R0, a
MOV  a, R0   → this can be eliminated.
ADD  e, R0
MOV  R0, d
```

A target machine with a rich instruction set can provide several ways to implement a given operation.

eg. If a target machine has "increment" instruction INC, then for

```
a = a+1
```

instead of

```
MOV  a, R0
ADD  #1, R0
MOV  R0, a
```

we can implement it more efficiently by

```
INC a
```

5. Register allocation →

Instructions involving register operands are shorter and faster than instructions involving operands in memory.

Register allocation → The set of variables in the program that will reside in the registers is chosen.

Register assignment → The specific registers that the variables will reside in is chosen.

6. Evaluation order →

At last, the code generator decides which order the instructions will be ~~evaluated~~ executed in.

The order in which computations are performed can affect the efficiency of the target program.

eg. $a+b-(c+d)*e$

Three address code	Code
$t1 = a+b$	MOV a, R_0
$t2 = c+d$	ADD b, R_0
$t3 = t2*e$	MOV R_0, t1
$t4 = t1-t3$	MOV c, R1
	ADD d, R1
	MOV e, R_0
	MUL R1, R_0
	MOV t1, R1
	SUB t3, R1
	MOV R1, t4

Reordered 3 address code	Code
$t2 = c+d$	MOV c, R_0
$t3 = t2*e$	ADD d, R_0
$t1 = a+b$	MOV e, R1
$t4 = t1-t3$	MUL R_0, R1
	MOV a, R_0
	ADD b, R_0
	SUB R1, R_0
	MOV R_0, t4

Reduces the number of final code by 2, thus saves cost.

Q2. A Simple Code Generator

A simple code generator generates the target code for a sequence of three address code statements and effectively uses registers to store operands of the statement.

eg. $a = b + c$

This can have the following sequence of codes →

ADD R_j , R_i if R_j contains c and R_i contains b

Cost = 1.

OR.

ADD c , R_i if a is in a memory location

Cost = 2

OR.

MOV c , R_j moving c from memory to register
ADD R_j , R_i

Cost = 3

Register descriptor → tracks what is currently in each registers

Address descriptor → stores ~~current~~ location where current value of a name can be found.

Generating code for assignment statement

Code generation algorithm →

for each 3 address statement of the form $x = y \text{ op } z$, perform the following actions-

1. Invoke function getreg to find the Location L where the result of y op z should be stored.
2. Consult address descriptor of y to find y', (the current location of y), then MOV y', L
3. Generate instruction OP z', L, where z' is the current location of z
4. Update the address descriptor of x to indicate that x is in Location L

Generating code for Assignment Statements →

the assignment $d = (a-b) + (a-c) + (a-c)$ can be translated into the following three address code →

$t = a - b$
$u = a - c$
$v = t + u$
$d = v + u$

Code sequence for the example is →

Statements	Code gen	Register descriptor	Address descriptor
t=a-b	MOV a, R_0 SUB b, R_0	R_0 contains t	t in R_0
u=a-c	MOV a, R_1 SUB c, R_1	R_0 contains t R_1 contains u	t in R_0 u in R_1
v=t+u	ADD R_1, R_0	R_0 contains v R_1 contains u	v in R_0 u in R_1
d=v+u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating code for indexed assignment →

Statements	code gen	cost
b = a[i]	MOV a[i], b	2
a[i] = b	MOV b, a[i]	

Statements	code gen	cost
a = b[i]	MOV b[R_i], R	2
a[i] = b	MOV b, a[R_i]	3

Generating code for Pointer assignments →

Statements	Code gen	cost
a = *p	MOV *Rp, a	2
*p = a	MOV a, *Rp	2

Generating code for conditional statements →

eg. x = y + z
if x < 0 , goto z

target code ⇒

```
MOV  y, R0
ADD  z, R0
MOV  R0, x
CJ   <z
```