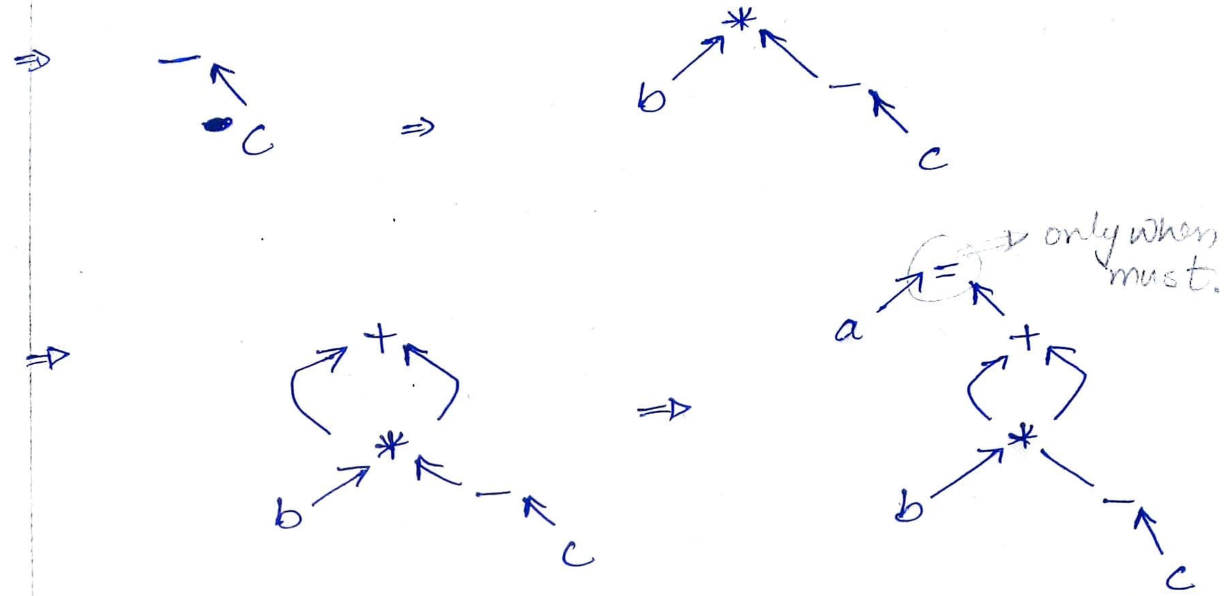
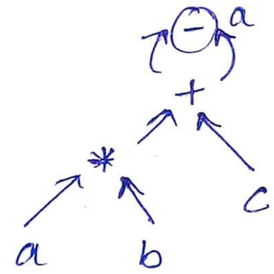
5. $a = b * -c + b * -c$

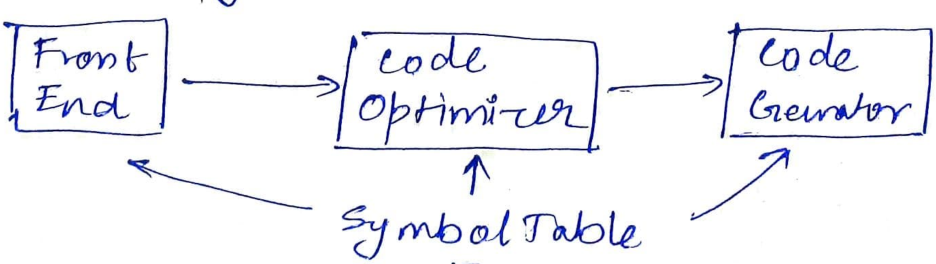


6. $a = (a * b + c) - (a * b - c)$



CODE GENERATION

- This is the final phase of the compiler.
- It takes as input the intermediate representation of the source program & produces as an output equalent traged program



~ We'll look at mainly 3 topics

1. Issues in the design of a Code Generator
2. Target Machine ~~and~~
3. Simple Code Generator

and finally : Peephole Optimization

1. Issues in the design of a Code Generator

~ While designing a code genrator we'd have to keep certain issues in mind they are.

1. Input to the code generator
2. Target Program
3. Memory Management
4. Instruction Selection
5. Register Allocation.
6. Evaluation Order.

#1. Input to the Code Generator

~ Code generator accepts the input from the code optimizer - which is the intemediate optimized code

~ Intemediate code is rep. using:

1. Postfix Notation

2 Three Address Code
- ↳ Quadraples
- ↳ Triples
- ↳ Indirect triples

3. DAG or Syntax directed Tree

~ The assumption is that the input is free of errors.

#2. Target Program

~ The output of the code generator is the target program

~ This can be in the form
- Absolute machine language
- Relocatable machine language
- Assemby Language.

* Absolute Machine Langugge

- The absolute machine program is placed in a fixed memory location in RAM(main memory)
- Suitable when the program is very small.
- ~~Com~~ Program is compiled & executed in a faster manner.

~~This Not So~~

* Relocatable machine Language

- It is nothing but object code.
- We need linker and loader. Linker ~~takes~~ links several program(modules) to a single program.
- Once linking is over it generates an executable file and then the loader loads this to the main memory.
- Whenever we get a free space, we use that to store a program.
- Suitable for both small & large programs.

* Assembly language

- It is better, because it is easier & efficient to generate assembly language.
- Further since most machine supports assembly language, they ~~are~~ can be extended.

- The most comom target machine architure are RISC, CISC & Stack Based (eg: JVMs for java byte code)

#3. Management of Memory

- Symbol table is used to mange the memory
- Mapping of variablenams to addrus is done co-opeativly by the front end & code generator.
- Hence all the labels should be done propery - (back jumps is ~~better~~ easier than front jumps)

#4. Instruction Selection

- Instuction selection and the speed of the instruction is very ~~importang~~ important.

eg:-

a = a+1

can be written as

MOV R_0, a
MOV R_0, #1
MOV R_0, a

now these three instruction can be replaced with single instruction : $\boxed{\text{INC a}}$ - this is. nomemory efficient & faster.

eg:- $a = b + c$

$d = a + e$

MOV	R_0, b	
ADD	R_0, c	
MOV	a, R_0	instead
MOV	R_0, a	of performing Redundant operations,
ADD	e, R_0	we can do
MOV	R_0, d	

$\Rightarrow$

MOV	R_0, b
ADD	R_0, c
ADD	R_0, c
MOV	d, R_0

~ The nature of instruction set of the target machine has a strong effect on the difficulty of instruction selection.

~ Uniformity & completeness of the instruction set are important factors.

~ Instruction speed & machine idioms are other important factors.

#5. Register Allocation

~ The key problem in code generation is what to hold in what registers.

~ Registers are fastes computational units for the target machine but we usuall do no have enough of them.

~ This performed in two ways, register allocation & register assignment.

(register is allocated)

~ Register Allocation specifies which register ~~variable~~ contains which variable.

eg:- R_0 contain a variable A
R_1 contain a variable B

(register is assigned to)

Register Assignment specifies which variable is contained in which register.

eg:- A will be contained by R_0

~ The diffrence is when the no. of variables either exeeds or is less than the no. of avaiable registers.

~ Finding optimal solution to assignment of variables to register's is difficult (even with single-register machine).

~ It is an NP-Complete problem (one of the plausible approach is heuristic optimization)

~ It becoms even more complecated if the target machine has certain conventions.

#6. Evaluation Order

- The order in which the instructions are executed as well as the operations are perfomed will decide the efficiency of the target code.

- Picking the best order is an NP-complete problem
- This can be solved to an exted by code optimization where the order of instuctions get changed.

- Other design goals of the target code generated include correctness, ease of impleuntation, testing and maintaintainability.

[2] Target Machine

- Familarity with the target machine & its instruction set is a pre-requisite for designing a good code generator.

1. Our target computer is a byte-addasable maine withe 4 byts to a word & with n-general purpose registers. $R_0 - R_{n-1}$.

2. The instruction set is in the format

$\boxed{\text{op destination, source}}$

op = opcode

3. It has the following op-codes : MOV, ADD, SUB.

4. The source and destination fields are not long enough to hold memory address.

- Hence certain bit pattern in these field specify that words following an instruction contain operands and/or addresses.

5. The source & destination of an instruction are specified by combining registers and memory locations with address mode.

6. The address modes together with their assembly-language forms & associated cost are as follows.

Mode	Form	Address	Added Cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + contents(R)	1.
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents(c + contents(R))	1
literal	#c	64. c	1

eg:- MOV M,R_0 - moves the contents of ~~register~~ R_0 to register R_0

MOV M,4(R0) - move contents of ~~contents~~ (4 + contents (R_0)) to M

MOV M, *4(R_0) - move contents (~~4~~ contents (4+ contents (R_0))) to M.

MOV R_0, #1 - load constant 1 to register R_0

Instruction Cost

- Cost of an instruction is one plus the cost associated with the source & destination address modes, indicated by address cost in the above table.

- This cost corresponds to the length of the instruction

- Address mode with only length of the instruction ~~is stored~~ has zero cost.

eg:- MOV R_0 , b

ADD R_0 , c   cost = 6

MOV R1, R_0

MOV *R1, *R0   cost = 2

ADD *R2, *R0

A simple code generator

- It generates target code for a sequence of 3-address code instructions.

~ We assume that the computed result is stored in registers as long as possible.

~ It is removed from the register if ~~the regi~~

(i) The register is needed for another computation

(ii) just before a procedure call or jump statement.

Register and Address Descriptors (Data Structures)

~ The code generator uses descriptors to keep track of registers contents & address for names.

1. A Register Descriptor : Keeps track of what is currently in each register.

   ~ It is consulted whenever a new generator is needed. (initially all registers are empty)

2. An Address Descriptor : Keeps track of the location where the current value of the name (space) (variable) can be found at runtime. (provides variable info)

   ~ The location might be register, a stack location or a memory address, etc.

Algorithm for simple code generator

~ It uses a function getReg() to assign registers to variables.

~ The following actions are performed by code generatore for an instruction $\boxed{x = y \text{ op } z}$.

~ It assumesthat L is the location where the output of y op z ~~istobe~~ is to be saved.

Steps

1. Call the function "get Reg()" to get the location of L.

2. Determine the present location of "y" by consulting address descriptor of y.

~ If y is no present in (any) location L, then generate the instruction $\boxed{\text{mov } L, y'}$ to copy the value of y to L.

y prime or address descriptor of y - where value of the 'y' is stored

3. The present location z is determined using step 2 and the instruction is generated as $\boxed{\text{op } L, z'}$

address descriptor of z' ⇒ value of z is perform op with value in L.

4. Now L contains the value of $\boxed{y \text{ op } z}$.

~ If L is a register then update its register descriptor ~~such~~ that it contains the value of x.

~ Likewise update the address descriptor of x to indicate that it is stored in 'L'.

5. If y & z have no future use, then we can update the register descriptors of both y & z to remove them.

Eg:- Let the expression be: $d = (a-b) + (a-c) + (a-c)$

$t_1 = a - b$
$t_2 = a - c$
$t_3 = t1 + t2$
$d = t3 + t_2$

– is the corresponding 3 address code.

	Statements	Code Generator	Register Descriptor	Address Descriptor
1.	$t_1 = a - b$	MOV R_0, a	R_0 contains t_1	t_1 in R_0
		SUB R_0, b	~~R_0 contains t_1~~	~~t_1 in R_0~~
2.	$t2 = a - c$	MOV R_1, a	R_0 contains t_1	t_1 in R_0
		SUB R_1, c	R_1 contains t_2	t_2 in R_1
3.	$t_3 = t_1 + t_2$	ADD R_0, R_1	R_0 contains t_3	t_3 in R_0
			R_1 contain t_2	t_2 in R_0
4.	$d = t_3 + t_2$	ADD R_1, R_0	R_1 contains d	d in R_1

III Peephole Optimization

- This technique is applied to improve the performance of the program.
- It is done by examining a short sequence of instructions through a window (or peephole) and replace the instructions by a faster (or short sequence of instructions)

> Peep hole is a small ~~short~~ moving window on the taget program.

Peep hole Optimization Techniques (or characteristics)

1. Redundant instruction Elemination
2. Removal of unreachable code
3. Flow of control optimizations
4. Algebraic simplifications & Reduction in Strength
5. Machine idioms.

#1. Redundant Instruction Elemination

~ Eg:- MOV R_0, a
MOV A, R_0

~ Here the value of a can be directly access from R_0 and it is not required to move it back to a.

~ If such a statemet is in a loop it will take ~~cause~~ more ~~exe~~ unnecessary exeation time.

~ Hene the instuction MOV a, R_0 is redundant.

#2. Removal of Unreachable Code

~ If certain statems are never executed ~~excluded~~ during the life time of the program, they are unreachable.

~ Hence is can be safely removed.

eg:-

```
def sum(a, b):
    return (a+b)
    print(a+b)
```

unreachable and canbe removed

3. Flow of Control Optimization

~ Using peep-hole optimization unnecessary jumps can be eliminated

eg:: goto L1
. . . .
L1 : goto L2
. . . .
L2 : goto L3
. . . .
~~L3 : goto~~
L3 MOV R_0, a

can be ⇒ optimized as.

goto L3
. . . .
L1 : goto L3
. . . .
L2 : goto L3
. . . .
L3 : mov R_0, a

4. Algebraic Simplifications & Reduction in Strength

~ Some expressions can be made simple

eg:. $x = x * 1$ ⇒ x
$x = x + 0$

$x = 3 + 4$ ⇒ $x = 7$

$x = x \wedge 2$
⇒ $x = x * x$

$b = y/3$
⇒ $b = y >> 3$

is reduced in strength by using "cheaper" operators.

5. Use of Machine Idioms

~ It is process of using powerful features of CPU instructions - which peforms the operation in faster manner

eg:- $a = a + 1$ ⇒ INC a ; $a = a - 1$ ⇒ DEC a.