Transformation on Basic Blocks :-

A basic block computes a set of expressions. Two basic blocks are said to be equivalent if they compute the same set of expressions.

There are two important classes of local transformations that can be applied to basic blocks

1. Structure-Preserving Transformation
2. Algebraic Transformations

(1) Structure-Preserving Transformation :-

The primary SPT on basic blocks are:

(a) Common subexpression Elimination –

Consider the basic block

```
a := b+c
b := a-d        —(1)
c := b+c
d := a-d
```

The second and fourth stmts. compute the same expression, namely b+c-d. so we can transform this block into following equivalent block

```
a := b+c
b := a-d        —(2)
c := b+c
d := b
```

The first and third stmt in (1) and (2) appear to have the same expression on the right, the second stmt. redefines 'b'. so the value of 'b' in the third stmt. is different from the value of 'b' in the first stmt.

(b) Dead-code Elimination –

Suppose there is a stmt. $x := y+z$ in the basic block and x is never subsequently used. Then this stmt. may be safely removed without changing the value of the basic block.

(c) Renaming Temporary variables :- Suppose we have a stmt. $t := b + c$. If we change this stmt. to $u := b + c$ where u is a new temporary variable and change all uses of this instance of t to u, then the value of basic block is not changed. Such a basic block is called normal-form block.

(d) Interchange of stmts :- Let us consider a block with the two adjacent stmts -

$$t_1 := b + c$$
$$t_2 := x + y$$

Then we can interchange the two stmts without affecting the value of the block iff neither x nor y is t_1 and neither b nor c is t_2.

(e) Algebraic Transformations :- Countless algebraic transformations can be used to change the set of expression computed by a basic block into an equivalent set. For example stmts $x := x + 0$ or $x := x * 1$ can be eliminated from a basic block.

Similarly, the exponential operator in the stmt $x := y ** 2$ can be replaced by the cheaper, but equivalent stmt.

$$x := y * y$$

Flow Graphs :- Graphical representation of basic block is called flow graph. The nodes of the flow graph are the basic blocks. One node is distinguished as initial; it is the block whose leader is the first stmt. There is a directed edge from block B_1 to B_2 if B_2 can immediately follow B_1 in some execution sequence; i.e. if

(1) there is a conditional or unconditional jump from the last stmt. of B_1 to the first stmt. of B_2, or

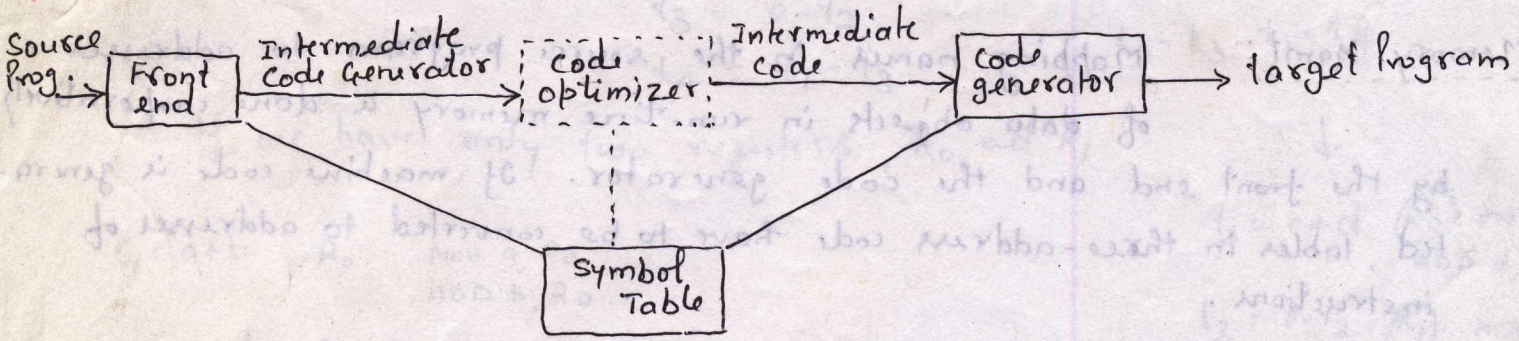
(2) B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

Loops :- A loop is a collection of nodes in a flow graph such that

① All nodes in the collection are strongly connected; that is, from any node in the loop to any other, there is path of length one or more, wholly within the loop, and

② The collection of nodes has a unique entry, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

Code Generation



Issues in the Design of a Code Generator :-

(1) **Input to the Code Generator:** The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with info? in the symbol table that is used to determine the run-time address of the data objects denoted by the names in the intermediate representation. We assume that prior to code generation the front end has scanned, parsed, and translated the source program into intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, pointers, integers etc.). We also assume that necessary type-checking has taken place.

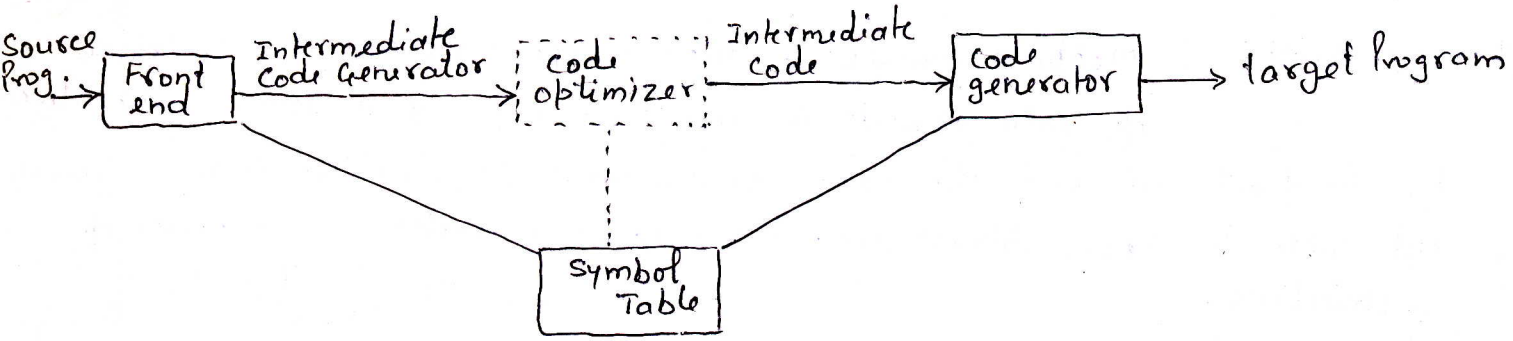
(2) **Target Program:-** Like the intermediate code, the o/p may take on a variety of forms; absolute machine code, relocatable machine code, or assembly language.

The advantage of producing absolute machine code as o/p is that it can be placed in a fixed location in memory and immediately executed.

Producing a relocatable machine code as o/p allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

In assembly language program, we can

Code Generation



Issues in the Design of a code Generator :-

(1) Input to the Code Generator: The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with info? in the symbol table that is used to determine the run-time address of the data objects denoted by the names in the intermediate representation. We assume that prior to code generation the front end has scanned, parsed, and translated the source program into intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, pointers, integers etc.). We also assume that necessary type-checking has taken place.

(2) Target Program :- Like the intermediate code, the o/p may take on a variety of forms: absolute machine code, relocatable machine code, or assembly language.

The advantage of producing absolute machine code as o/p is that it can be placed in a fixed location in memory and immediately executed.

Producing a relocatable machine code as o/p allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

In assembly language program, we can generate symbolic instructions and use the macro facilities of the assembler to help generate code. Producing assembly code doesn't

Because

duplicate the entire task of assembler, this choice is reasonable native, especially for a machine with a small memory, where a co must use several passes. Relocatable code will be preferred.

(3) Memory Mgmt: Mapping names in the source program to addresses of data objects in run-time memory is done cooperatively by the front end and the code generator. If machine code is generated, lables in three-address code have to be converted to addresses of instructions.

(4) Instruction selection :- for fast and efficient code generation choose appropriate instructions.

suppose $x := y + z$

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

will produce poor code.

$a := b + c$
$d := a + e$

would be translated into

```
MOV b, R0
ADD c, R0
MOV R0, a      → if a is not subsequently used
MOV a, R0      → redundant stmt
ADD e, R0
MOV R0, d
```

or

for increment instr^{n} $a := a + 1$

```
MOV a, R0
ADD #1, R0      can by replaced by INC
MOV R0, a       instr$^{n}$.
```

(5) Register Allocation :- Instructions involving register operands are usually shorter and faster than those involving operands in memory.

for exaple $a = b + c$

if b and c are in memory

```
MOV b, a
ADD c, a        six word cost
```

if a, b, c are in R_0, R_1, R_2

```
MOV *R1, *R0
ADD *R2, *R0    ~~three~~ two word cost
```

Choice of Evaluation order :-

$t_1 = a + b$
$t_2 = c + d$
$t_3 = e - t_2$
$t_4 = t_1 - t_3$

will require more registers than the order $t_2 - t_3 - t_1 - t_4$

if we have only two registers, R_0 and R_1

$t_1 = a+b$	R_0	MOV a, R_0 ADD b, R_0
$t_2 = c+d$	R_1	MOV c, R_1 ADD d, R_1
$t_3 = e - t_2$		MOV R_0, t_1
	R_0	MOV e, R_0 SUB R_1, R_0
$x = t_1 - t_3$	R_1	MOV t_1, R_1 SUB R_0, R_1

↓

$t_2 = c+d$	R_0	MOV c, R_0 ADD d, R_0
$t_3 = e - t_2$	R_1	MOV e, R_1 SUB R_0, R_1
$t_1 = a+b$	R_0	MOV a, R_0 ADD b, R_0
$x = t_1 - t_3$	R_1	SUB R_1, R_0
		MOV R_0, x

(7) Approaches to code generation :- choose better one either labelling and codegen or straightforward

Methods for evaluating semantic Rules :-

① **Parse-tree methods :-** At compile time, these methods obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input. These methods will fail only if the dependency graph for the particular parse tree under consideration has a cycle.

② **Rule-based methods :-** At compiler-construction time the semantic rules associated with productions are analyzed, either by hand, or by a specialized tool. For each production, the order in which the attributes associated with that production are evaluated is predetermined at compiler-construction time.

③ **oblivious Methods :-** An evaluation order is chosen without considering the semantic rules. For example, if translation takes place during parsing, then the order of evaluation is forced by the parsing method, independent of the semantic rules.

Rule-based and oblivious methods need not explicitly construct the dependency graph at compile time, so they can be more efficient in their use of compile time and space.

Evaluation order :- Any topological sort of a dependency graph gives a valid evaluation order. A topological sort of a directed acyclic graph is any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that edges go from nodes, $m_i \rightarrow m_j$ then m_i appears before m_j in the ordering.

Error-Recovery strategies :- (syntax)

① **Panic-mode recovery :-** The simplest method to implement and can be used by most parsing methods. On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. While panic-mode correction often skips a considerable amount of i/p without checking it for additional errors, it has the advantage of simplicity and it is guaranteed not to go into an infinite loop.

② **Phrase-level recovery :-** On discovering an error, a parser may perform local correction on the remaining input, i.e., it may replace a prefix of the remaining i/p by some string that allows the parser to continue. A typical local correction would be to replace a comma by a semicolon, delete an extraneous semicolon. We must be careful to choose replacements that do not lead to infinite loops. Its major drawback is the difficulty it has in coping with situations in which actual error has occurred before detection.

③ **Error productions :-** If we have a good idea of the common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. We then use the grammar augmented by these error productions to construct a parser.

④ **Global Correction :-** Ideally, we would like a compiler to make as few changes as possible in processing an incorrect i/p string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect i/p string 'x' and grammar 'G', these algo. will find a parse tree for a related string 'y', such that the no. of insertions, deletions and changes of tokens required to transform 'x' into 'y' is as small as possible. But these methods are too costly to implement in terms of time and space, so are only of theoretical interest.