Subject - Compiler Design [VII th sem].

1. (i) Left factoring is the process of factoring out common:-
   (c) prefixes of alternative

(ii) The three address code contain:
   (a) Exactly three address.

(iii) Which of the following is not true of an LL(1) Parser?
   (c) It is also called a shift-reduce parser

(iv) Which of the following statement is true:-
   (c) CLR parser is more powerful than LALR.

(v) Lexemes are the smallest possible logical units of a program.
   (a) True

(vi) Compiler is preferred than interpreter because:
   (a) It take less time to execute.

(vii) A quadruple is a record structure with:
   (a) Four field

(viii) Following techniques are based on function preserving transformations:-
   (a) Common sub expression elimination.

(ix) Shift reduce parser is a top down parser.

~~a~~ (b) False

(x) ...... is an interface between sources program and Compiler.

(a) Lexical Analysis.

SECTION - B.

UNIT-I

2(a) Lexeme :- Lexemes are the smallest possible logical units of a program.

Defination :- " Lexeme is a sequence of characters that form a token."

example :- sum := a+b;

lexemes are :- Sum, a, + b ;

Token :- A lexical token is a sequence of charact that can be treated as a unit in the gramme of the programming language.

→ A token can be defined as a :

"a logical group or unit to which lexeme belongs to."

Token and lexeme :- A token is a categorized blo of text. The block of text corresponding to the token is known as lexeme.

→ A lexeme is an interface of token.

" A lexeme is a sequence of characters in a source program that is matched by that the pattern for a token."

Example:- sum := a+b

Sr. No.	Lexeme	~~Pattern~~ Token
1.	sum	identifier
2.	=	assign-operator.
3.	a	identifier
4.	+	add-operator
5.	b	identifier
6.	~~b~~ ;	delimiter.

Pattern:- "A pattern is a rule describing the set of lexemes that can represent a particular token in source language."

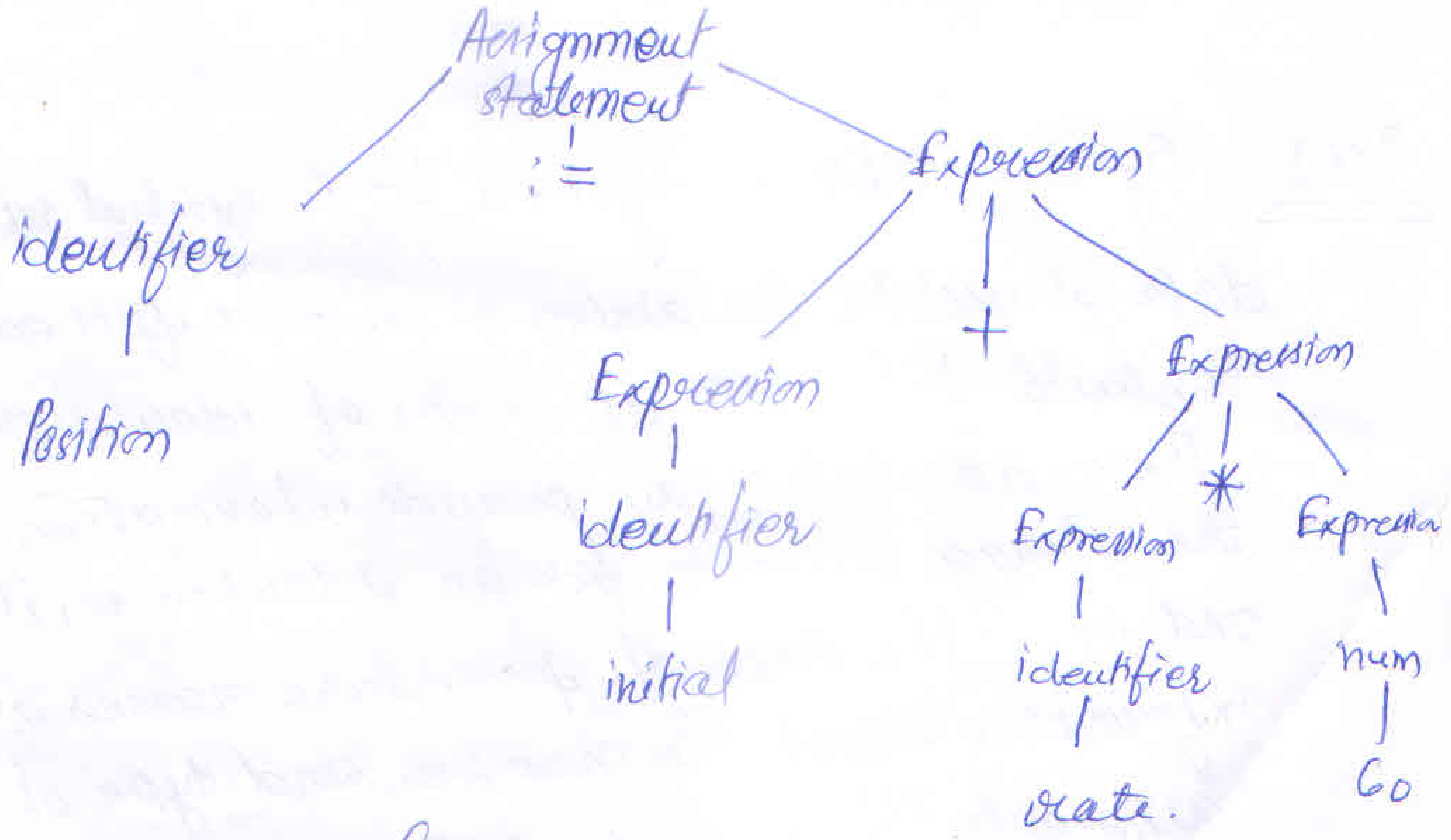
→ Pattern is a rule.

2(b) Syntax Analysis :-

→ Hierarchical analysis is called parsing or syntax analysis, in which characters or tokens are grouped hierarchically into nested collections, with corrective meaning.

→ Syntax analysis involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesized output, usually the grammatical phases of the source program are represented by a parse tree.

example :- Position := initial + rate * 60.



→ Manually in this phase syntax will be checked.

Intermediate phase :- ⇒ After syntax and semantic analysis, Compilers generates an explicit intermediate representⁿ of the source program.

→ The intermediate representation should have two important properties:-

(i) It should be easy to produce.

(ii) It should be easy to translate into the target program.

→ It change integer to real also.

→ Three address code is code.

example:-

```
Temp1 := int to real (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
 id1  := temp3
```

$\boxed{id1 := id2 + id_3 * 60}$

2.(C) Symbol table Management :- A symbol table is a data structure containing a record for each identifier, with fields for attributes of identifier.

→ These attributes may provide information about the storage allocated for an identifier, it's scope and in the case of procedure names it stores information about the number and type of its arguments, the method of passing argument and the type returned if any.

→ When an identifier in the source program is detected by lexical analyzer, the identifier is entere in the symbol table, however the attributes of an identifier Cannot normally be determined during lexical analysis.

Error detection and reporting :- A good Compiler must determine the line number of program exactly, where the error have occurred.

Following types of error may occur at different levels of Compilation.

→ Lexical Analysis phase Can detect errors Caused by illegal or unrecognized characters in the input that donot form any token of the language and another type of error may be due to unterminated character or string.

→ Symbol table management module encounters errors, when there exists an identifier that has multiple declarations with Contradictory attributes.

3(a) Predictive Parsing table

$S \rightarrow A$

$A \rightarrow Ab \mid Ad$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

<u>Solution</u> :- Since the grammar contains left recursion in A-productions, which may cause the parser to stack into an infinite loop, therefore it is required to eliminate left recursion before constructing the predictive parsing table. After elimination of left recursion, grammar becomes

~~$S \rightarrow A$~~
~~$A \rightarrow aBX$~~
~~$X \rightarrow dX \mid \epsilon$~~
~~$B \rightarrow bBC \mid f$~~
~~$C \rightarrow g$~~

$S \rightarrow A$

$A \rightarrow A'$

$A' \rightarrow bA' \mid dA' \mid \epsilon$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

$FIRST(A') = FIRST(bA') \cup FIRST(dA') \cup \{\epsilon\}$
$= \{b, d, \epsilon\}$

$FIRST(A) = FIRST(A') = \{b, d, \epsilon\}$

$FIRST(S) = FIRST(A) = \{b, d, \epsilon\}$

$FIRST(B) = FIRST(bBC) \cup FIRST(f)$
$= \{b, f\}$

$FOLLOW(S) = \{\$\}$
$FOLLOW(A') = \{\$\}$
$FOLLOW(A) = \{\$\}$

	b	d	f	g	$
S	$S \to A$	$S \to A$			$S \to \varepsilon$
A'	$A' \to bA'$	$A' \to dA'$			$A' \to \varepsilon$
A	$A \to A'$	$A \to A'$			$A \to \varepsilon$
B	$B \to bBc$		$B \to f$		
C				$C \to g$	

3(b)

SLR	CLR	LALR
(i) It has no lookahead	(i) It uses lookahead symbol.	(i) It also uses 1 lookahead symbol.
(ii) It supports very less categories of language	(ii) It supports wide category of language	(ii) It supports more languag than SLR but less than CLR.
(iii) It uses follow for reduction process.	(iii) It ~~ite~~ uses 1 lookahead symbol for reduction process.	(iii) It uses 1 lookahead symbo for reduction process.
(iv) It is easiest to implement.	(iv) It is complex to implement.	(iv) It is easy in comparision to CLR but complex in comparision to SLR.
(v) It is least powerful parser.	(v) It is powerful than LALR and SLR.	(v) It is powerful than SLR but less powerful than CLR.

S.No.	SLR	CLR	LALR
(vi)	SLR Parser is more efficient with respect to time and space	(vi) CLR is least efficient with respect to time and space.	(vi) LALR is more efficient than CLR but less efficient than SLR.
(vii)	In this shift-reduce and reduce-reduce Conflict arises	(vii) In CLR also shift-reduce & reduce-reduce Conflict arises.	(vii) In LALR shift-reduce Conflict do not arise only reduce-reduce Conflict arises.
(viii)	every SLR lang. is CLR and LALR	(viii) Every CLR lang. Cannot be SLR or LALR.	(viii) Every LALR lang. Can be SLR but cannot be CLR.
(ix)	It Contains 100 of states.	(ix) It Contains 1000 of states.	(ix) It Contains 100 of states.

(c) ~~a(~~ $-(a+b)*(c+d)+(a+b+c)$

Three Address Code

$t_1 := a+b$

$t_2 := c+d$

$t_3 := t_1+c$

$t_4 := -t_1$

$t_5 := t_1 * t_2$

$t_6 := t_5 + t_3$

~~t~~

Triple

	OP	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	+	(0)	c
(3)	Uminus	(0)	
(4)	*	(3)	(1)
(5)	+	(4)	(2)

UNIT - III

UNIT-III

4.(a) (i) This is the simplest form of allocation.

(ii) In this allocation scheme, names are bound to storage as the program is compiled, so there is no need for a run time support package.

(iii) Since, the bindings do not change at run time, every time a procedure is activated. Its names are bound to the same storage location.

(iv) Early programming languages, such as fortran has static storage, the amount of which was known at compile time static storage allocation is efficient because no time or space is expended for storage management during execution.

(v) It is incomplete with recursive sub programs calls, with data structures whose size is dependent on computed or input data & with many other desirable language features.

at run time

4.(b) :- Exactly what ~~type o~~ information is stored in the symbol table depends on many things. The prog-ramming language will determine much of the information that is stored, but the target archite-chure will also influence what data is stored.

Five classes of information stored in symbol table.

① Constants:- Constants are identifiers that represent a fixed value - one that can never be changed.

② Variables:- Variables are identifiers, whose value may change between executions & during a single execution of program.

③ User defined types:- A user defined type is typically a conglomeration of one or more existing types (some languages allow us to define recursive structure.)

④ Subprograms:- Procedures, functions & methods are named segments of code. Naturally, the symbol table should record a procedure's name.

Classes :- Classes are abstract data types, that restrict access to its members and encapsulation provides convenient language level polymorphism.

4.(C) Storage Allocation

Storage allocation is basically depends on programming language implementation. According to programming language implementation following storage allocation strategies are applied for activation records.

① Static allocation lays out storage for all data objects at compile time.

② Stack allocation manages run time storage as a stack.

③ Heap allocation allocates and deallocates storage as needed at run time from a data area known as heap.

Static allocation :-

① This is the simplest form of allocation

(ii) It lays out storage for all data object at compile time.

(iii) ~~a~~ Since, the bindings do not change

4.(b) :- Exactly what ~~type o~~ information is stored in the symbol table depends on many things. The prog-ramming language will determine much of the information that is stored, but the target archite-chure will also influence what data is stored.

Five classes of information stored in symbol table.

① Constants:- Constants are identifiers that represents a fixed value - one that can never be changed.

② Variables:- Variables are identifiers, whose value may change between executions & during a single execution of program.

③ User defined types:- A user defined type is typically a conglomeration of one or more existing types (some languages allow us to define recursive structure.)

④ Subprograms:- Procedures, functions & methods are named segments of code. Naturally, the symbol table should record a procedure's name.

Classes :- Classes are abstract data types, that restrict access to its members and encapsulation provides convenient language level polymorphism.

4.(c) Storage Allocation

Storage allocation is basically depends on programming language implementation. According to programming language implementation following storage allocation strategies are applied for activation records.

① Static allocation lays out storage for all data objects at compile time.

② Stack allocation manages run time storage as a stack.

③ Heap allocation allocates and deallocates storage as needed at run time from a data area known as heap.

Static allocation :-

① This is the simplest form of allocation

(ii) It lays out storage for all data object at compile time.

(iii) ~~S~~ Since, the bindings do not change

at run time, every time a procedure is activated.
(4) In this allocation scheme, names are bound to storage as the program is compiled, so there is no need for a run time support package.
(2) Stack based :-
(i) This is the simplest runtime storage management technique.
(ii) Storage allocation begins at one end.
(iii) Storage must be freed in the reverse order of allocation, so block allocated space recently must free the space ~~free~~ first.
(iv) Only a single stack pointer is all that is needed to control the storage management.
(3) Heap Based Storage Management :-
(1) Heap is used for storage of values which may require to be accessible from the time the storage is allocated until the program terminates.
(2) The major problem arises in these type of allocation is that, how to reallocate or reuse the allocated space.
(3) Two methods (Garbage Collection & Reference counters method) are available to manage the heap.
(4) These allocation allocates and deallocates storage as needed at run time from a data area known as heap.

UNIT-IV

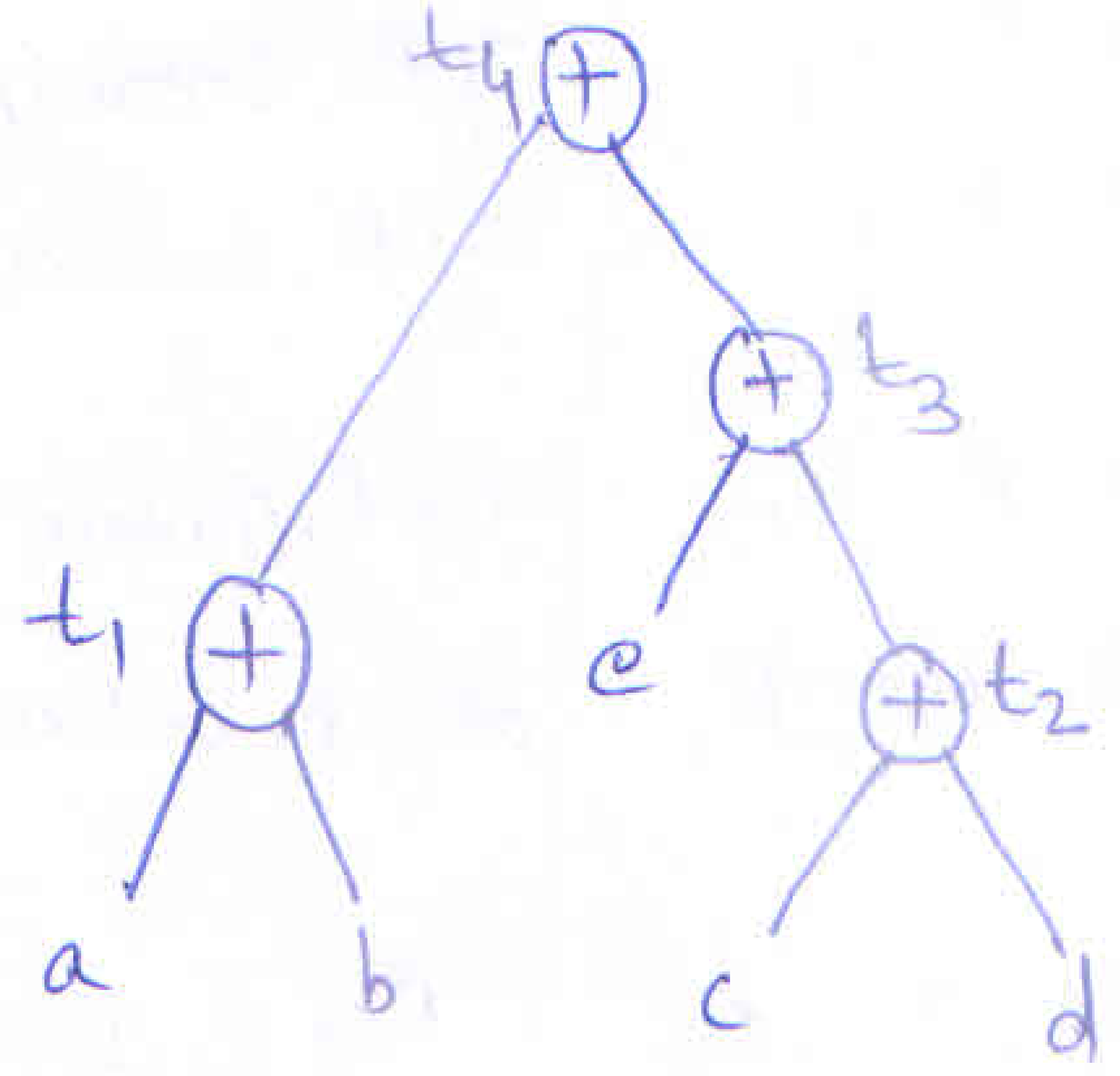
5.(a) DAG

$(a+b)+(e+(c+d)$

Three Address Code of the given statement (expression) is :-

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

DAG :-



(b) Code for

$$w := (A-B)+(A-C)+(A-C)$$

Three address code will be:-

$t_1 := A-B$

$t_2 := A-c$

$t_3 := t_1+t_2$

$t_4 := t_3+t_2$

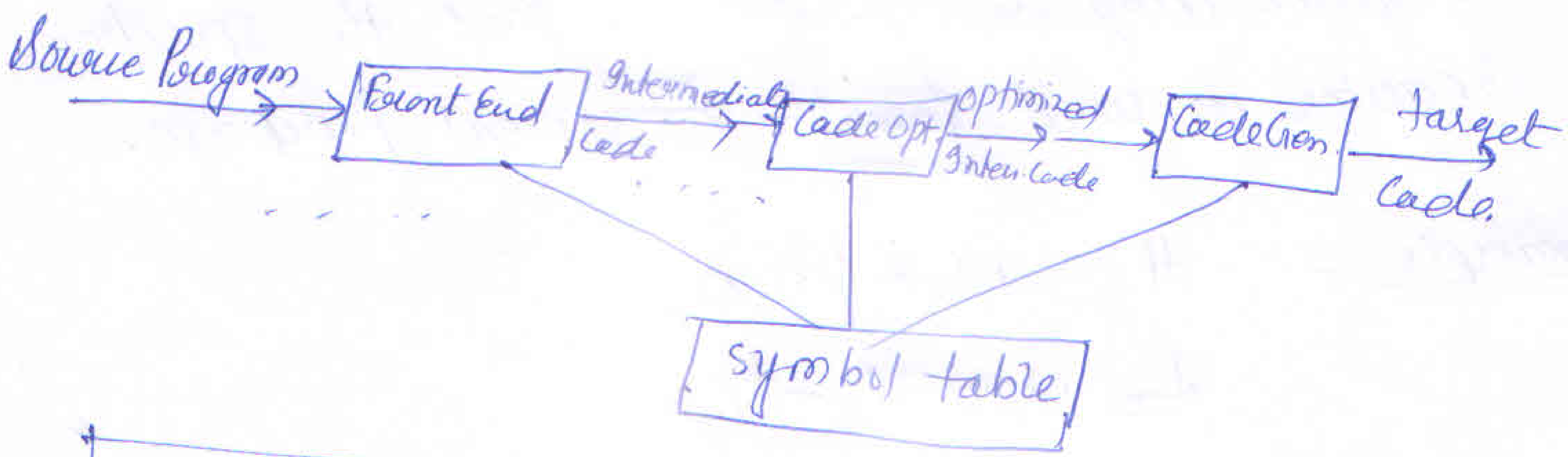
$w := t_4$

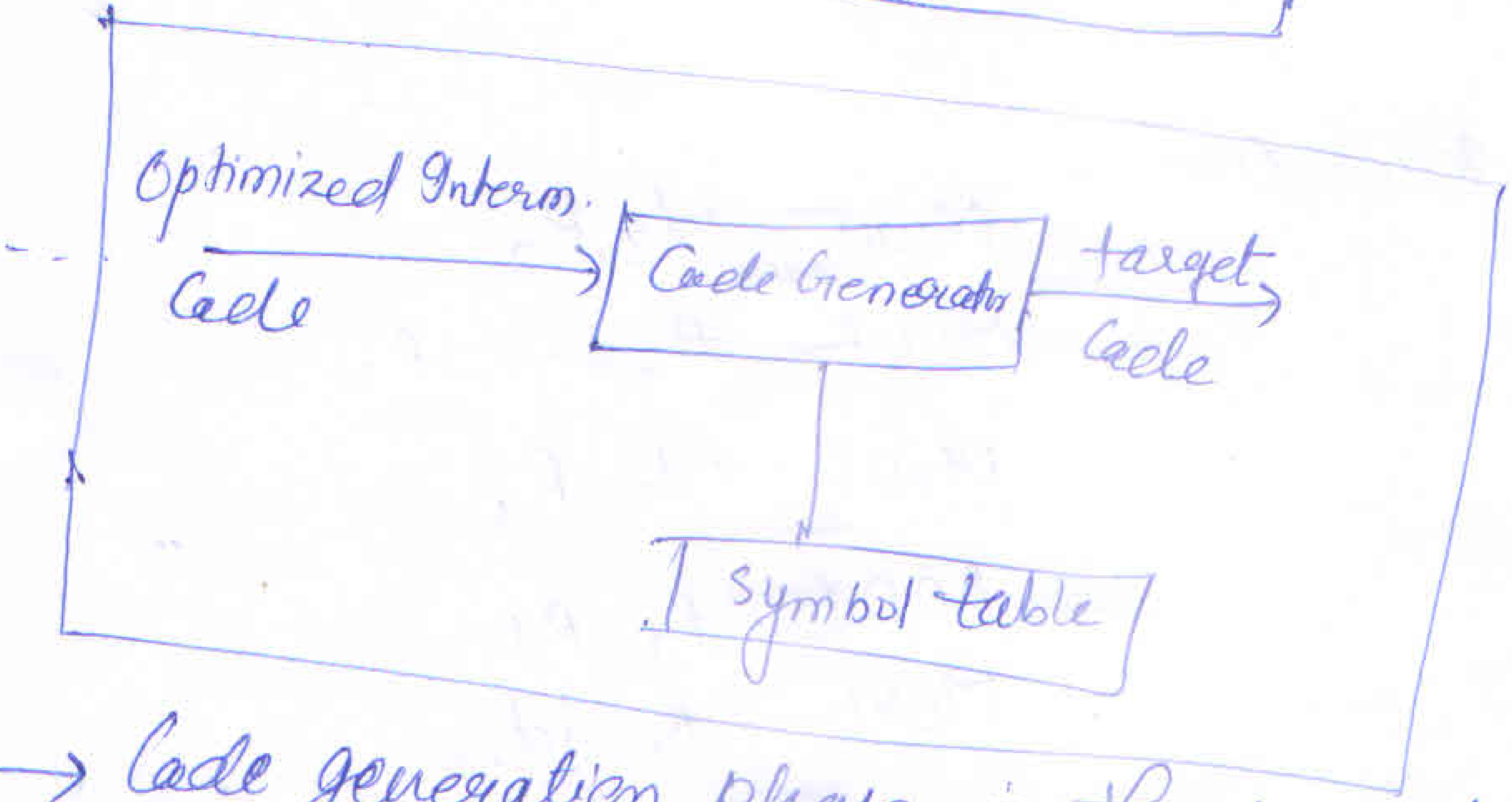
Three Add. Code. (Statement)	Instruction Register	Reg. Descriptor	Address Descriptor
$t_1 := A-B$	MOV A, R_0	reg R_0, Contains A	A is in R_0
~~$t_2 :=$~~	SUB B, R_0	reg R_0, Contains t_1	t_1 is in R_0
$t_2 := A-C$	MOV A, R_1	reg R_1, Contain A	A is in R1 Reg.
	SUB C, R_1	reg. R1 Contain t_2	t_2 is in R1 Reg.
$t_3 := t_1+t_2$	ADD R_1, R_0	Reg. R_0 Contains t_3	t_3 is in R_0
$t_4 := t_3+t_2$	ADD R_1, R_0	reg. R_0 Contains t_4	t_4 is in R_0 reg.

t_4 is in Memory.

5.(c)

Block diagram:-





→ Code generation phase is the last phase of Compilation process.

→ Input of the code generation phase is optimized Intermediate Code.

→ ~~op~~ Output of the code generation phase is the target Code.

→ Code generation phase takes the help from symbol table.

→ Register allocation and Register assignment is the important phase (task) of code generation phase.

→ Register allocation :- The names (code), ~~are~~ which are stored in the register is focused.

→ Register assignment :- In this part, the specific register is used for the assignment purpose.

Example :- $t_1 := id_3 * 60.0$

~~t_2~~ $id_1 := id_2 + t_1$.

after code generation :-

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id.
```

UNIT-V

6. (a) Code optimization phase attempts to improve the intermediate Code, so that the resulting machine Code executes faster and requires less amount of storage space, in order to save machine resources.

→ The Compilers that apply Code improving transformation are called optimizing Compiler. ~~There~~

→ There are basically two types of optimization.

(i) Machine independent optimizations

→ Program transformation that improve the target Code without taking into consideration any properties of target machine.

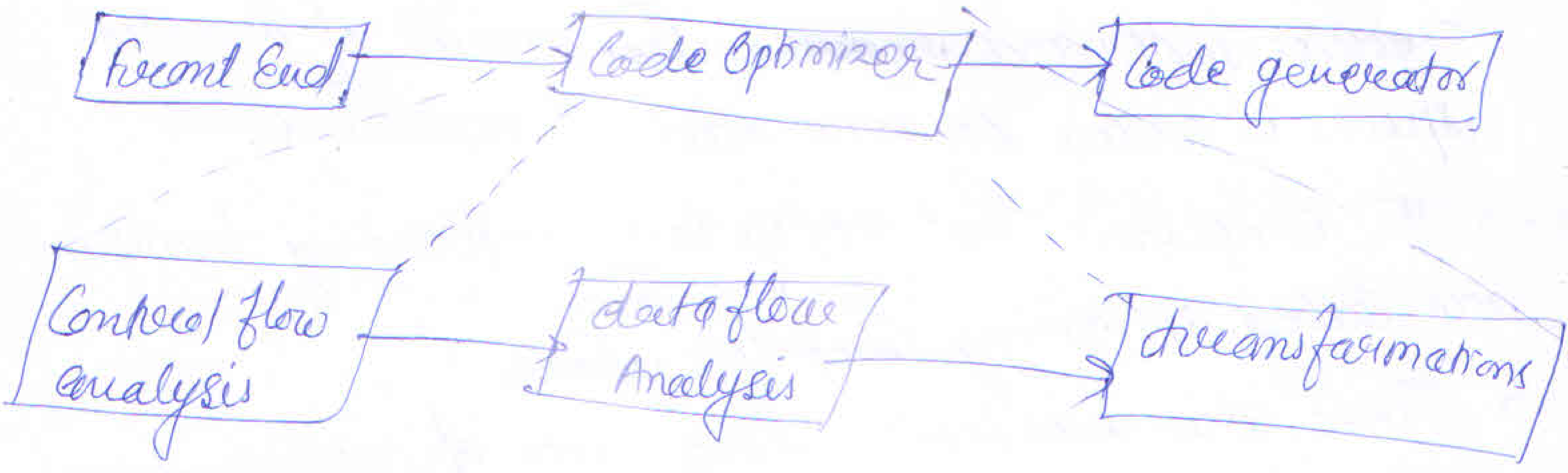
(ii) Machine dependent optimization:- e.g. register allocation and collination of special machine instruction following are the Criteria for Code improving transformati

→ 1. transformation must preserve the meaning of program, i.e. optimization must not change the o/p produced by a program for a given input.

2. → On the average ~~of~~ the transformation must speed up the program by a measurable amount.

3 → A tranformation must be worth the effort.

The transformation of intermediate code are done using the following organization.



Advantages :-

① operation needed to implement high level Construc (such as address Calculation) are made explicit in intermediate code, so it is possible to optimize them.

② Intermediate code can be relatively independen of target machine.

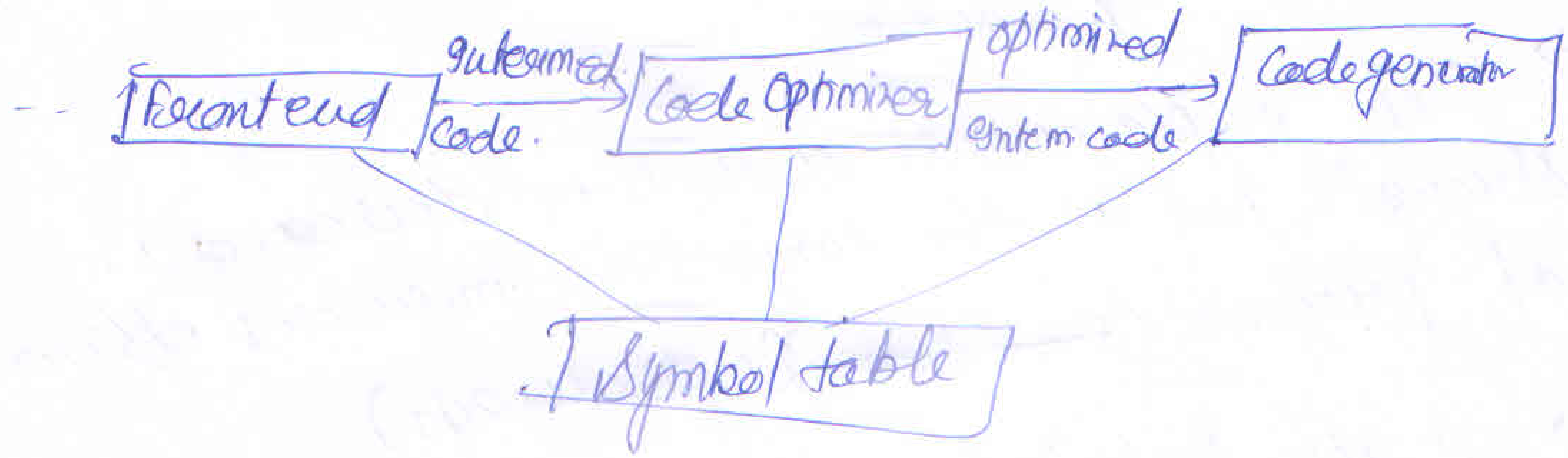
6.ⓑ Reduction in strength ~~measure~~ :- replaces the expensive operations by equivalent cheaper ones on target m/c.

→ In multiplication of two integers numbers requires shift and addition operation where as addition operation can be expressed simply for ex.

6.(C) Basic block:- It is a sequence of ~~basic block~~ Consecutive statement in which, flow of Control enters at the begining and leaves at the end without halt or possibility of branching except at the end.

→ A name in a basic block is said to be live at a given point if, its value is used after that point in the program, perhaps in another block.

blockdiagram



Code optimization is reffered to as the process of optiming the intermediate Code.

→ The semantic equivalance of the program should not change.

→ Input [front end] ⇒ The input of the Code optimizer is the output of the front end. The input of the Code optimization phase is the Intermediate Code.

→ Code Optimization:- Code optimization is the phase where the Code is optimized.

→ In the Code optimization phase semantics of equivalence of the program should not change.

→ The next block is ~~target~~ $\boxed{\text{Code generator}}$ → In Code generator phase, the final (target) code is produced. Code generation is the last phase of compilation process. In this phase register allocation and Register assignment is an important task.

→ Symbol table :- A symbol table is a data structure used by a compiler to keep track of scope, life and binding information about names. These names are used in the source program to identify the various program elements, like variables constants, procedures and the labels of statem