1. Intermediate Code Generation

1. Introduction :

The compiler translates the source program (written in high level language) into target program (machine code). The translation is done in several phases. Between the analysis of source program and the generation of target code, an intermediate representation (IR) of the program is produced. This intermediate code is easier to produce and to modify than the target code.

2. What is Intermediate Code ?

Intermediate Code is a low level language-like representation of the source program. It is machine independent and is used as an input to the code optimizer and code generator.

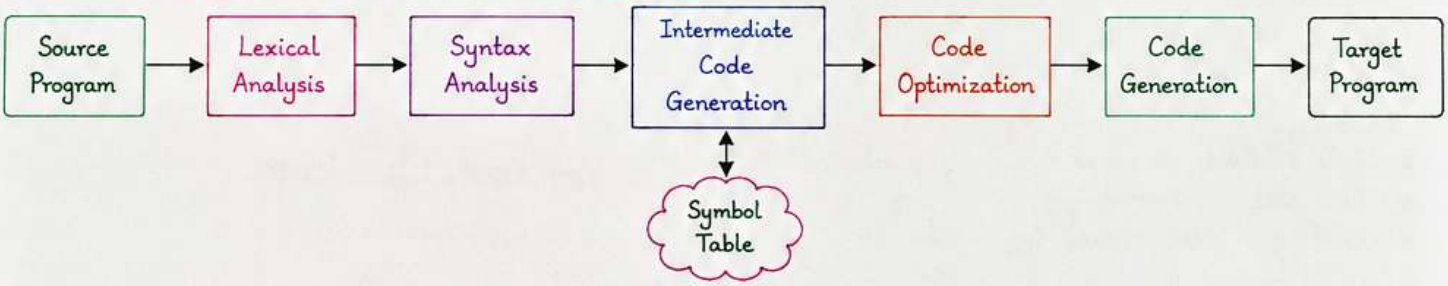
3. Need / Importance of Intermediate Code Generation :

* Separates machine independent optimization from machine dependent code generation.
* Makes the compiler portable to different machines.
* Simplifies code generation.
* Facilitates optimization of code.
* Provides a convenient form for program manipulation.

*** Important Point ***

The intermediate code should be easy to generate, easy to manipulate and easy to translate.

4. Position of Intermediate Code in Compiler :



5. General Structure of Intermediate Code Generation :

Note :

Translation scheme depends on the form of intermediate code used.

6. Methods of Intermediate Code Generation :

1. Syntax Directed Translation (SDT)
2. Using Attribute Grammars
3. Using Translation Schemes

7. Properties of Good Intermediate Code :

* Easy to generate.
* Easy to manipulate.
* Independent of target machine.
* Should preserve the meaning of the source program.
* Should be suitable for code optimization.

8. Example :

Source Program

```
a = b + c * d ;
e = (a - f) / g ;
```

Three Address Code (Intermediate Code)

```
1.  t1 = c * d
2.  t2 = b + t1
3.  a  = t2
4.  t3 = a - f
5.  e  = t3 / g
```

Quadruple Representation

No.	op	arg1	arg2	result
1	*	c	d	t1
2	+	b	t1	t2
3	=	t2	-	a
4	-	a	f	t3
5	/	t3	g	e

Exam Tip

*

Know different forms of intermediate code and their examples.

2. Intermediate Code Forms

1. Introduction :

There are many forms of intermediate code. The choice of a particular form depends on the source language, the target machine and the optimization techniques to be used. Some forms are linear (list based) while others are graphical (tree or graph based).

*** Important Point ***

The intermediate code should be easy to generate, easy to manipulate, machine independent and suitable for optimization.

2. Need for Different Intermediate Code Forms :

* Different forms are suitable for different compiler phases.
* Some forms are easy to generate but hard to optimize.
* Some forms are compact and easy to optimize.
* The choice affects the efficiency of the compiler.
* No single form is best for all situations.

3. Classification of Intermediate Code Forms :

Intermediate code forms are mainly classified as :

1. Linear (List based) forms
2. Graphical (Tree / Graph based) forms

4. Linear (List Based) Intermediate Code Forms :

Code Form	Description
1. Postfix (Reverse Polish Notation)	Operands appear before operator. No parentheses required.
2. Three Address Code (TAC)	Each instruction has at most three addresses. Very popular and used in most compilers.
3. Quadruples	Represented as (op, arg1, arg2, result). Easy to generate and manipulate.
4. Triples	Represented as (op, arg1, arg2). Result is identified by the position (index).
5. Indirect Triples	A table of pointers to triples. Facilitates code motion and optimization.

5. Graphical (Tree Based) Intermediate Code Forms :

Code Form	Description
1. Syntax Trees	Show the syntactic structure of expressions and statements.
2. Expression Trees	Represent expressions in tree form. Useful for optimization.
3. Directed Acyclic Graphs (DAG)	A DAG represents expressions and eliminates redundant computations.

6. Examples of Different Forms :

(a) Source Expression

(a + b) * (c - d)

(b) Postfix

```
a  b  +
c  d  -
*
```

(c) Three Address Code

```
t1 = a + b
t2 = c - d
t3 = t1 * t2
```

(d) Quadruples

No.	op	arg1	arg2	result
1	+	a	b	t1
2	-	c	d	t2
3	*	t1	t2	t3

(e) Triples

No.	op	arg1	arg2
0	+	a	b
1	-	c	d
2	*	(0)	(1)

(f) Indirect Triples

Ptr	→	Triple No.
0	→	0
1	→	1
2	→	2

7. Comparison of Linear and Graphical Forms :

Basis	Linear Forms	Graphical Forms
Representation	List of statements	Tree or Graph structure
Generation	Easy to generate	Relatively complex
Space Requirement	More space (linear)	Less space (compact)
Optimization	Some optimizations easy	Better for optimization
Use	Widely used in practice	Used for advanced optimizations

8. Advantages and Disadvantages :

Advantages	Disadvantages
✔ Provides machine independence.	✖ Extra time required for translation.
✔ Makes optimization easier.	✖ More storage may be needed.
✔ Simplifies code generation.	✖ Complex forms are hard to generate.
✔ Improves portability of compiler.	

9. Summary :

Intermediate code can be represented in different forms. The linear forms like Postfix, Three Address Code, Quadruples, Triples and Indirect Triples are most widely used. The graphical forms like Syntax Trees, Expression Trees and DAGs are used for advanced optimization. The choice of form depends on the requirements of the compiler.

Exam Tip

Learn the examples of all forms. Diagrams and tables are important in exam.

3. Three Address Code (TAC)

1. Introduction :

Three Address Code (TAC) is a linear intermediate code form in which each instruction has at most three addresses. It is simple, easy to generate and easy to optimize. TAC is widely used by modern compilers.

2. Need of TAC :

* Easy to generate from high level language.
* Easy to convert to target code.
* Facilitates code optimization.
* Provides a good balance between readability and efficiency.
* Most machine independent intermediate code.

3. General Form of TAC :

A TAC instruction has one of the following forms –

Form	Meaning
x = y op z	Binary operation
x = op y	Unary operation
x = y	Copy statement
x = y[i]	Array reference
x[i] = y	Array assignment
x = &y	Address of y
x = *y	Indirection (value at address y)
if x relop y goto L	Conditional jump
goto L	Unconditional jump
param x	Parameter passing
call p, n	Procedure call (p = name, n = no. of params)
return x	Return from procedure

4. Example :

Source Expression : a = (b + c) * (d - e) / f

Step	Three Address Code
1.	t1 = b + c
2.	t2 = d - e
3.	t3 = t1 * t2
4.	t4 = t3 / f
5.	a = t4

Note :

t1, t2, t3, ... are temporary variables introduced by the compiler.

5. Example with Control Flow :

Source Code :

```
if (a < b)
   c = c + 1;
else
   c = c - 1;
```

Three Address Code :

No.	TAC Instruction	Meaning
1	if a < b goto L1	If a < b is true, go to L1
2	c = c - 1	Else part (false case)
3	goto L2	Jump to L2 (end)
4	L1: c = c + 1	True part (label L1)
5	L2:	End of if-else

6. Advantages of TAC :

* Machine independent.
* Easy to generate and translate.
* Facilitates optimization.
* Compact and simple representation.
* Supports structured programming constructs.

7. Disadvantages of TAC :

* May require more temporary variables.
* Does not represent high level structures directly.
* Some information may be lost during translation.

8. Common Operators in TAC :

Operator Category	Examples
Arithmetic	+, -, *, /, %
Relational	<, <=, >, >=, ==, !=
Logical	&& (and), || (or), ! (not)
Bitwise	&, |, ^, ~, <<, >>
Assignment	=, +=, -=, *=, /=

9. Complete Example (Expression + If Statement) :

Source Code :

```
if (a > b)
   x = (c + d) * e;
else
   x = (c - d) / e;
```

Three Address Code :

No.	TAC Instruction	Meaning
1	if a > b goto L1	If a > b is true, go to L1
2	t1 = c - d	Else part
3	t2 = t1 / e	Else part
4	x = t2	Else part
5	goto L2	Jump to end
6	L1: t3 = c + d	True part
7	t4 = t3 * e	True part
8	x = t4	True part
9	L2:	End

Important Points

* Each TAC instruction has at most three addresses.
* TAC is easy to optimize.
* It is the most widely used intermediate code form.
* TAC can be directly mapped to machine instructions.

Exam Tip

Always convert given expressions or statements into TAC step by step. Show all intermediate temporaries clearly.

4. Boolean Expressions

1. Introduction :

Boolean expressions are expressions that result in one of the two boolean values - True (1) or False (0). They are used in decision making statements such as if, while, for etc.
The intermediate code generation for boolean expressions is done using conditional jumps.

Important Points

- Boolean expression → Expression evaluated to True or False.
- They are evaluated using relational and logical operators.
- Short-circuit evaluation can be applied for optimization.
- Intermediate code uses conditional jumps for boolean expressions.

2. Need for Boolean Expressions :

- Useful for representing conditions in control flow.
- Helps in generating efficient intermediate code.
- Facilitates the evaluation of complex conditions.
- Used in various constructs like if, while, do-while, for.

3. Types of Operators in Boolean Expressions :

Category	Operators	Meaning	Example
Relational	<, <=, >, >=, ==, !=	Compare two operands	a < b
Logical	&&, ||, !	AND, OR, NOT	a && b
Arithmetic	+, -, *, /, %	Arithmetic operations	a + b
Parentheses	( )	Change precedence	(a>b) && (b>c)

4. Representation of Boolean Expressions :

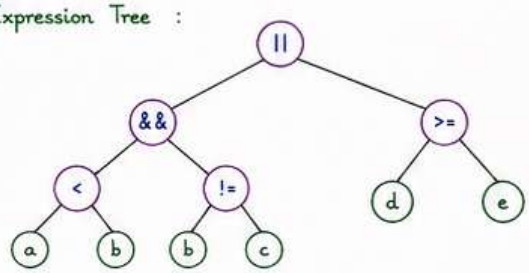
A boolean expression can be represented in different forms such as -

① Infix form (normal mathematical form)
② Prefix form
③ Postfix form
④ Syntax tree / Expression tree

5. Example of Boolean Expressions :

Consider expression : a < b && b != c || d >= e

(a) Infix form : a < b && b != c || d >= e
(b) Postfix form : a b < b c != && d e >= ||
(c) Prefix form : || && < a b != b c >= d e
(d) Expression Tree :



6. Evaluation of Boolean Expressions :

- Boolean expressions are evaluated from leaves to root in the expression tree.
- Result is either True (1) or False (0).

Example : Evaluate a < b && b != c

a	b	c	a < b	b != c	(a < b) && (b != c)
3	5	5	T	F	F
2	4	5	T	T	T
6	4	3	F	T	F
7	8	2	T	T	T

T → True (1), F → False (0)

7. Intermediate Code Generation for Boolean Expressions :

Boolean expressions are usually translated into conditional jumps.

General form :

```
if (E) goto L1      // if E is True, go to L1
  goto L2           // if E is False, go to L2
```

Where,
E → Boolean expression
L1 → Label for True part
L2 → Label for False part

Note :
Short-circuit evaluation can be applied to avoid evaluating unnecessary expressions.

8. Example with Intermediate Code :

Source Code :

```
if (a < b && b != c)
    x = 1;
else
    x = 0;
```

Intermediate Code (Using Labels) :

```
    if  a < b  goto L1      // if a<b True
        goto L2             // if a<b False
L1 : if b != c goto L3      // if b!=c True
        goto L2             // if b!=c False
L3 :  x = 1                 // True part
        goto L4
L2 :  x = 0                 // False part
L4 :                        // End
```

9. Summary :

- Boolean expressions are used in decision making statements.
- They are evaluated using relational and logical operators.
- Expression trees are used to represent and evaluate expressions.
- Short-circuit evaluation improves efficiency.
- Intermediate code for boolean expressions uses conditional jumps.
- These expressions are fundamental in control flow of a program.

5. Case Statements

1. Introduction :

A case statement allows the execution of one block of statements from many alternatives depending on the value of an expression. It is similar to switch statement in C / C++.

General Form :

```
case E of
    L1 : S1
    L2 : S2
    .    .
    .    .
    Ln : Sn
endcase
```

- **E** → expression
- L_i → labels (constant values)
- S_i → sequence of statements

If E matches L_i, then S_i is executed.

2. Need for Case Statement :

- Provides multi-way branch.
- More efficient than multiple if-else statements.
- Improves readability and maintainability.
- Used widely in high level languages.

3. Intermediate Code Generation for Case Statement :

Case statements are translated using conditional jumps.

General Idea :

1. Evaluate the expression E.
2. Compare E with each label L_i.
3. If match found, jump to corresponding statement S_i.
4. If no match, jump to default (if present) else to next statement after case.

4. Example :

Source Code (High Level) :

```
case x of
   1 :  a = b + c ;
   2 :  a = b - c ;
   3 :  a = b * c ;
   4 :  a = b / c ;
 default : a = 0 ;
endcase
```

Assume, expression x is already evaluated and stored in temp variable 't1'.

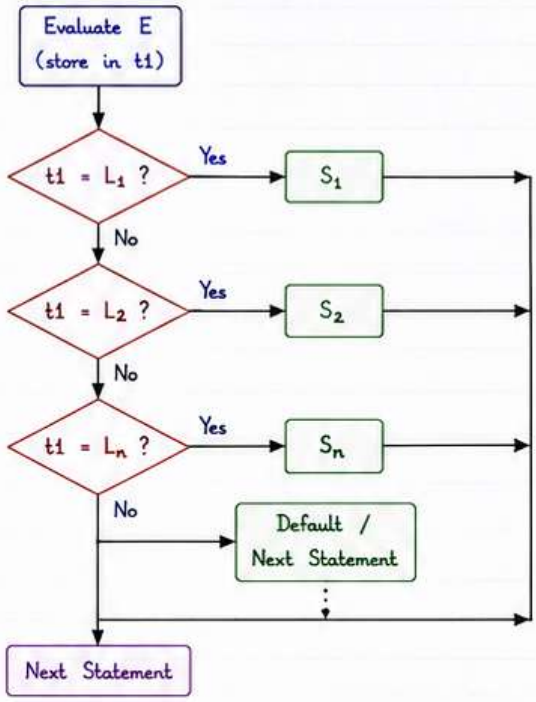
5. Intermediate Code (Using Conditional Jumps) :

Step	Three Address Code
1	if t1 = 1 goto L1
2	if t1 = 2 goto L2
3	if t1 = 3 goto L3
4	if t1 = 4 goto L4
5	goto Ld // default part
L1:	a = b + c
	goto Lend
L2:	a = b - c
	goto Lend
L3:	a = b * c
	goto Lend
L4:	a = b / c
	goto Lend
Ld:	a = 0
Lend:	// next statement

// Case parts

Note : Ld is executed if no label matches the expression.

6. Flow Diagram :



7. Example with Flow Table :

t1 Value	Comparison	Jump Taken	Executed Statement
1	t1 = 1	goto L1	a = b + c
2	t1 = 2	goto L2	a = b - c
3	t1 = 3	goto L3	a = b * c
4	t1 = 4	goto L4	a = b / c
Others	No match	goto Ld	a = 0

8. Case Statement with Range of Values :

Some languages allow range in case (not in C).

Example :

```
case x of
 1..10 : S1
 11..20 : S2
 default : S3
endcase
```

Intermediate Code

```
Check if x in 1..10
Check if x in 11..20
else S3
```

9. Advantages :

- More efficient than nested if-else.
- Easy to read and understand.
- Compiler can generate optimized code.

10. Disadvantages :

- Only constant labels allowed in some languages (like C).
- If number of cases is large, chain of comparisons may increase time.
- Not suitable for range checks in many languages.

11. Summary :

Case statements provide multi-way branching depending on the value of an expression. They are translated into a sequence of conditional jumps using labels and goto statements. If no match is found, control goes to default part or next statement.

6. BACKPATCHING

1. Introduction :

In code generation, the target address of some jump statements may not be known at the time of generating the code. Such jumps are recorded with a placeholder (blank) and later their target address is filled (patched). This technique is called **Backpatching**.

2. Need for Backpatching :

- Target address of jump may not be available during one pass compilation.
- Used in translation of control flow constructs (if, while, do-while, for).
- Improves efficiency by avoiding multiple passes.
- Facilitates incremental code generation.

3. Basic Idea :

Whenever a jump target is not known, put a blank (place holder) in the target field and store the address of that instruction in a list.
When the target address becomes known, go back to that list and fill (patch) the target field.

4. Data Structures Used :

Data Structure	Description
Pointer List	List of addresses of instructions whose target fields are to be patched.
Makelist (i)	Creates a list with single element i.
Merge (p1, p2)	Merges two lists p1 and p2.
Backpatch (p, t)	Fills target field of all instructions in list p with target address t.

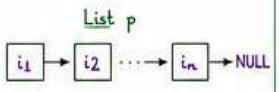
5. Operations on Pointer Lists :

① Makelist (i) :
Create a list with a single element i.

② Merge (p_1, p_2) :
Combine two lists into a single list.

③ Backpatch (p, t) :
Replace all blanks in the target fields of instructions in list p with address t.

Example of List :



Each node contains address of an instruction to be patched.

8. Example : (Translation of while Loop)

Source Code :

```
while (a < b) {
    x = x + 1;
}
```

Intermediate Code using Backpatching :

No.	Instruction	Target
1	L1: if a < b goto _	_
2	goto _	_
3	x = x + 1	–
4	goto L1	1

Backpatching :

- p1 = {1}
- p2 = {2}
- Backpatch(p_1, 3)
- Backpatch(p_2, 5)

Final Code :

No.	Instruction	Target
1	L1: if a < b goto 3	3
2	goto 5	5
3	x = x + 1	–
4	goto 1	1
5	...	–

Note :
- Backpatching is widely used in translation of if, if-else, while, do-while, for and switch statements.

Exam Tip
Always draw tables and flow diagrams in exam.
Write algorithms for Makelist, Merge and Backpatch.
Give examples for if and while statements.

6. Example : (Translation of if Statement)

Source Code :

```
if (a < b)
    x = c + d;
else
    x = c - d;
```

Intermediate Code using Backpatching :

No.	Instruction	Target	Comment
1	if a < b goto _	_	Branch to be patched
2	goto _	_	Skip then-part
3	x = c + d	–	Then-part
4	goto _	_	Jump to end
5	x = c - d	–	Else-part
6	...	...	Next statement

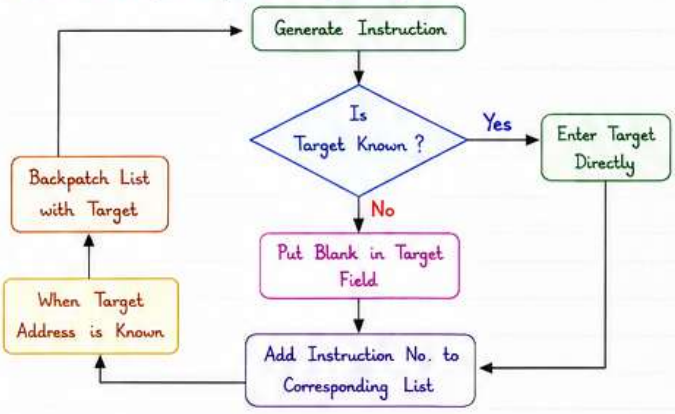
Backpatching Steps :

① After generating instruction 1, create list p1 = {1}.
② After generating instruction 2, create list p2 = {2}.
③ After then-part (instruction 3), create list p3 = {4}.
④ Backpatch(p_1, 3) // Target of if-true = 3
⑤ Backpatch(p_2, 5) // Target of goto = 5
⑥ Backpatch(p_3, 6) // Target of goto after then-part = 6

Final Intermediate Code after Backpatching :

No.	Instruction	Target	Comment
1	if a < b goto 3	3	True → then-part
2	goto 5	5	False → else-part
3	x = c + d	–	Then-part
4	goto 6	6	Jump to end
5	x = c - d	–	Else-part
6	...	–	Next statement

7. Flow Diagram for Backpatching :



9. Summary :

- Backpatching is a technique used to handle forward jumps.
- Blank targets are filled later when the target address is known.
- It uses pointer lists and operations (makelist, merge, backpatch).
- Efficient in one pass code generation.
- Essential for translation of control flow statements.

7. Procedure Calls

1. Introduction :

A procedure call is a request to execute a procedure (function). When a procedure is called, control is transferred to the called procedure and after its completion, control is returned to the calling point.

2. Need for Procedure Calls :

* Supports modular programming.
* Reduces code size and repetition.
* Improves readability and maintainability.
* Allows recursion.
* Facilitates separate compilation.

3. General Form of Procedure Call :

```
call     P, n        // call procedure P with n arguments
     ..............
     ..............
P :  <procedure body>
     return
     ..............
```

4. Phases in Procedure Call Execution :

① Evaluate actual parameters.
② Save return address (address of next instruction).
③ Transfer control to the called procedure.
④ Execute the called procedure.
⑤ Return the result (if any).
⑥ Restore control to the calling point.

5. Example :

Source Code	Three Address Code

`int add (int a, int b)`
`{`
`  int c;`
`  c = a + b;`
`  return c;`
`}`
 	
`void main ()`
`{`
`  int x, y, z;`
`  x = 10;`
`  y = 20;`
`  z = add (x, y);`
`}`
	1. x = 10
	2. y = 20
	3. param x // pass first argument
	4. param y // pass second argument
	5. call add, 2 // call add with 2 args
	6. z = return // get return value
	7. goto L1
	add :
	8. t1 = a + b
	9. return t1
	L1 :
	10. .....

Important Points

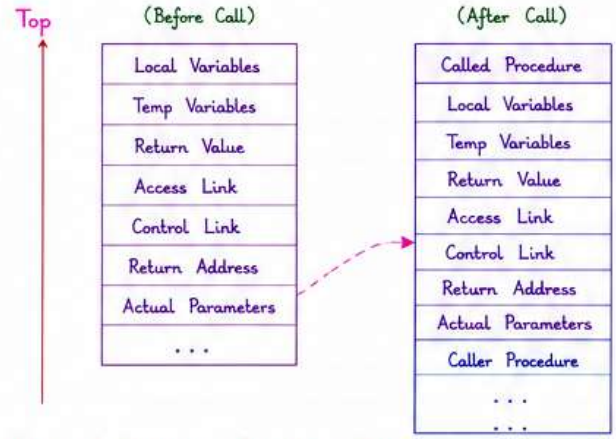
* 'param' is used to pass arguments.
* 'call P, n' calls procedure P with n arguments.
* 'return' brings result back to calling point.
* The return address is automatically saved.
* Stack is used to manage procedure calls.

6. Activation Record (Stack Frame) :

During a procedure call, an activation record is created on the run-time stack.

Field	Description
Actual Parameters	Values of arguments passed to procedure.
Return Address	Address of next instruction to execute after return.
Control Link (Static Link)	Pointer to activation record of calling procedure.
Access Link (Dynamic Link)	Pointer to activation record of caller.
Local Variables	Space for local variables of called procedure.
Temporary Variables	Space for temporaries used in procedure.
Return Value	Space to store the return value.

7. Example of Stack During Procedure Call :



8. Flow of Control in Procedure Call :

9. Nested and Recursive Calls :

* Nested Calls : When procedure P calls Q and Q calls R.
* Recursive Calls : When a procedure calls itself.

Example (Recursive Factorial) :

```
int fact (int n)
{
   if (n <= 1)
       return 1;
   else
       return n * fact(n-1);
}
```

Note :
Recursive calls create multiple activation records on stack until the base case is reached.

10. Summary :

* Procedure calls allow modular and reusable code.
* Parameters are passed using 'param' instruction.
* 'call P, n' transfers control and saves return address.
* Activation records are created on stack.
* Return value is brought back using 'return'.
* Stack helps in supporting nested and recursive calls.

8. Basic Blocks

1. Introduction :

A basic block is a sequence of consecutive statements in which control enters at the beginning and leaves at the end without stopping or branching except at the last statement.

It is a maximal sequence of statements with only one entry point and one exit point.

2. Properties of Basic Blocks :

* The first statement has only one entry (no jumps to it).
* The last statement has only one exit (jump or fall through).
* All other statements have exactly one entry and one exit.
* Control flows through the block in a straight-line manner.
* No branching inside the block except at the last statement.

3. Importance / Need :

* Simplifies the flow of control.
* Useful in code optimization.
* Helps in register allocation.
* Forms the basis for flow graph and optimization techniques.

4. How to Form Basic Blocks :

① Partition the program into basic blocks.
② Each block begins with a leader.
③ Basic block ends at the last statement before the next leader.
④ There is no branch or jump inside the block except last statement.

5. Example :

Consider the following three address code :

No.	Three Address Instructions
1.	x = a + b
2.	if x > y goto (5)
3.	z = x - y
4.	goto (6)
5.	w = x + y
6.	u = w + z
7.	return u

Leaders are :

1 → First statement
3 → Target of goto from (2)
5 → Target of goto from (4)
6 → Statement following goto (4)
7 → Statement following previous leader (6 is leader, next is 7)

Therefore, basic blocks are :

```
B1 : { 1, 2 }      // from leader 1
B2 : { 3, 4 }      // from leader 3
B3 : { 5 }         // from leader 5
B4 : { 6, 7 }      // from leader 6
```

Important Points

* First statement is a leader.
* Target of any jump is a leader.
* Statement following a jump is also a leader.
* Every basic block has only one entry and exit.

6. Representation :

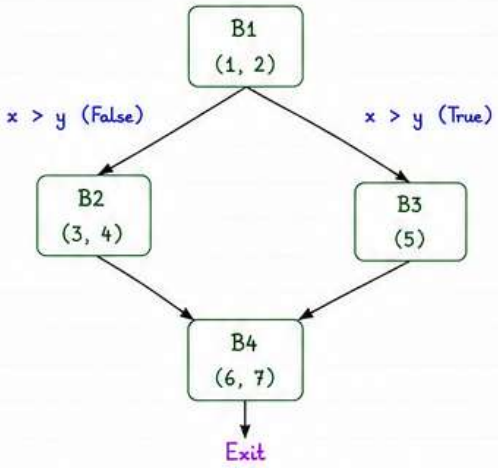
Basic blocks can be represented in a program as follows :

```
B1 :
        Statement 1
        Statement 2
        . . .
B2 :
        Statement p
        Statement q
        . . .
Bn :
        Statement x
        Statement y
        . . .
```

Each block has a unique entry and a unique exit.

7. Flow Graph of Basic Blocks (Example) :

From the above example, flow graph is :



8. Characteristics of Basic Blocks :

Feature	Description
Entry Point	Only the first statement of the block can be the target of a jump.
Exit Point	Only the last statement may have a jump or branch.
Internal Flow	All other statements inside the block have exactly one entry and one exit.
Straight Line	Control flows in a sequential manner inside the block.
Maximal	The block is maximal, i.e., it cannot be extended further.

9. Summary :

* Basic blocks are straight-line sequences with single entry and single exit.
* Leaders help in finding basic blocks.
* Basic blocks are useful in optimization and code generation.
* Flow graph is built using these basic blocks.

Exam Tip

Always identify leaders first.
Draw basic blocks and then construct the flow graph.
Write properties and importance in exam for full marks.

9. Leaders in Basic Blocks

1. Introduction :

* Leaders are statements that can begin basic blocks.
* Identifying leaders helps in constructing basic blocks from a given sequence of statements.
* Once leaders are known, we can partition the program into basic blocks.

2. Definition of Leader :

A statement S is a leader if any of the following is true :

① S is the first statement of the program.

② S is the target of a jump.

③ The statement immediately follows a jump (conditional or unconditional).

3. Algorithm :

Input : Sequence of statements $S_1, S_2, \ldots, S_n$
Output : Set of leaders L
1. $L \leftarrow \{S_1\}$ // first statement is a leader
2. For each statement S_i in the sequence do
   3. If S_i is the target of any jump, then add S_i to L
   4. If S_i is a conditional or unconditional jump, then if $i+1 \leq n$, add S_{i+1} to L
   5. Remove duplicates from L and sort in order
6. End

4. Example :

Consider the following intermediate code :

```
1.  x = a + b
2.  if x < y goto 4
3.  goto 7
4.  y = x * 2
5.  goto 8
6.  z = y + 1
7.  w = z - 3
8.  if a > b goto 2
9.  goto 10
10. print w
```

Targets of jumps :
4, 7, 2

Leaders :
1 (first statement)
2 (target of 8)
4 (target of 2)
7 (target of 6)
9 (follows jump at 8)

Basic Blocks :
(1), (2, 3), (4, 5), (7), (8, 9), (10)

5. Important Points :

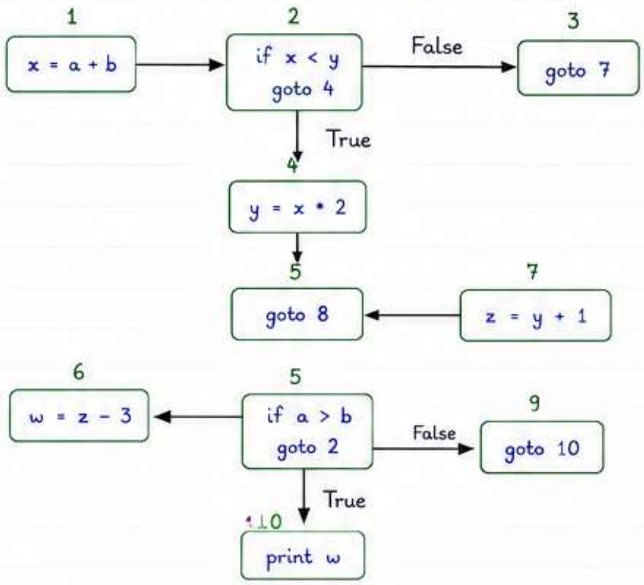
* Leaders help in building basic blocks.
* Basic blocks are the foundation for flow graph construction.
* Correct identification of leaders ensures accurate control flow analysis.

Key Points

✓ Leader = First statement / Jump target / Follows jump
✓ Leaders divide program into basic blocks
✓ Basic blocks have single entry and single exit
✓ Used in flow graph construction and optimization

6. Graphical Representation :

Consider the same code as in example.



7. Summary :

* Leaders are essential for identifying basic blocks.
* They help in control flow analysis and optimization.
* We use leaders to construct basic blocks from a sequence of intermediate code.

Quick Check

Find leaders in the following code :

```
1. a = b + c
2. if a > b goto 4
3. goto 6
4. d = a - b
5. goto 7
6. e = d + 1
7. f = e * 2
8. if f != 0 goto 2
9. print f
```

Answer : Leaders are 1, 2, 4, 6, 8

8. Applications :

* Used in basic block formation
* Used in flow graph construction
* Used in data flow analysis and optimization
* Used in register allocation and code generation

Note :

Correct identification of leaders is the first step towards construction of basic blocks and flow graph.

10. DAG (Directed Acyclic Graph)

1. Introduction :

A DAG (Directed Acyclic Graph) is a directed graph with no cycles. It is used in compiler to represent expressions in an efficient way.
It helps in eliminating common sub-expressions and improves the optimization of code.

2. Need for DAG :

* Eliminates common sub-expressions.
* Reduces the number of computations.
* Saves time and storage.
* Helpful in optimization of code.
* Used for efficient code generation.

3. Basic Terminology :

Term	Meaning
Node	Represents an operator or an operand.
Leaf Node	Represents an operand (identifier or constant).
Interior Node	Represents an operator.
Edge	Directed line from operator to its operands.
Root	Node having no incoming edges.

4. Construction Rules of DAG :

① Create a leaf node for each operand.
② Create an interior node for each operator.
③ Connect operator node to its operand nodes.
④ If a node with same label and same children exists, reuse it (common sub-expression elimination).
⑤ The DAG must be acyclic.

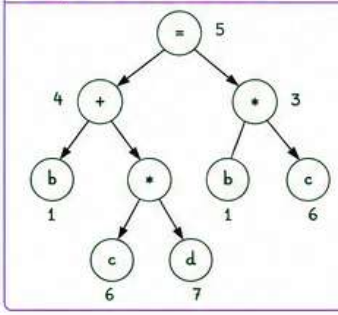
5. Example :

Consider the expression : a = b + c * d + b * c

Step 1 : Identify Sub-expressions

Sub-expressions are :
1. c * d
2. b + (c * d)
3. b * c
4. (b + c * d) + (b * c)
5. a = (above result)

Step 2 : DAG Construction



DAG Table Representation

Node No.	Label	Left Child	Right Child
1	b	-	-
2	*	6	7
3	*	1	6
4	+	1	2
5	=	4	3
6	c	-	-
7	d	-	-

Note :
* Nodes 1, 6 are shared in multiple sub-expressions.
* This reduces redundant computations.

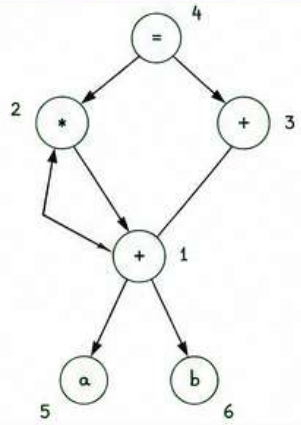
6. Example 2 :

Expression : x = (a + b) * (a + b) + (a + b)

Step 1 : Sub-expressions

1. a + b
2. (a + b) * (a + b)
3. (a + b) * (a + b) + (a + b)
4. x = (above result)

Step 2 : DAG



DAG Table

Node No.	Label	Left Child	Right Child
1	+	5	6
2	*	1	1
3	+	2	2
4	=	3	1
5	a	-	-
6	b	-	-

Observation :
Sub-expression (a+b) is computed only once and reused, hence DAG eliminates redundancy.

7. Advantages of DAG :

* Eliminates common sub-expressions.
* Reduces number of computations.
* Improves code optimization.
* Saves time and memory.
* Leads to efficient intermediate code.

8. Limitations of DAG :

* Complex to construct for large expressions.
* Requires more storage for very large programs.
* Not suitable if expression has side-effects.

9. Applications of DAG :

* Used in optimization phase of compiler.
* Used in expression evaluation.
* Used in common sub-expression elimination.
* Used in code generation for better efficiency.

10. Summary :

* DAG is a directed acyclic graph used to represent expressions.
* It removes redundant computations by eliminating common sub-expressions.
* It consists of nodes (operators/operands) and directed edges.
* It improves the efficiency of generated code.
* It is an important tool in compiler optimization.

Exam Tip

* Always list all sub-expressions before construction.
* Draw DAG carefully and reuse common nodes.
* Make DAG table for full marks.
* Show sharing of nodes clearly.

11. Symbol Table Management

1. Introduction :

A Symbol Table is a data structure used by a compiler to store information about identifiers (variables, constants, functions, arrays, structures, etc.) appearing in a program. It is used in all phases of compilation.

Stores
- ✓ Name
- ✓ Type
- ✓ Scope
- ✓ Value
- ✓ Address

... etc.

2. Need for Symbol Table :

- Stores information about identifiers.
- Helps in semantic analysis.
- Supports type checking.
- Used for memory allocation.
- Detects multiple declarations.
- Helps in code optimization.

3. Information Stored in Symbol Table :

Field	Description
Name	Name of the identifier
Type	Data type (int, float, char, etc.)
Scope Level	Scope in which it is declared
Storage Class	auto, static, register, extern, etc.
Value	Value (for constants)
Address	Memory location
Other Info	Array info, function params, etc.

4. Operations on Symbol Table :

① Insert(id, info) – Insert a new identifier.
② Lookup(id) – Search information about an identifier.
③ Update(id, info) – Update information.
④ Delete(id) – Delete identifier (when scope ends).
⑤ EnterScope() – Create a new scope.
⑥ ExitScope() – Remove symbols of current scope.

5. Structure of Symbol Table Entries :

Each entry of the symbol table contains all the information about an identifier.

Name	Type	Scope Level	Storage Class	Value	Address	Other Info

6. Example :

Consider the following program :

```
int a, b;
float x;
{
  int a;
  a = 10;
  x = 20.5;
}
```

Symbol Table after processing :

Name	Type	Scope Level	Storage Class	Address
a	int	1	auto	1000
b	int	1	auto	1002
x	float	1	auto	1004
a	int	2	auto	2000

Note :
- The second 'a' has different scope level (2) from first 'a' (1).
- Both can exist simultaneously.
- When scope level 2 ends, entry of 'a' *(scope level 2) is removed.

Important Points
- ✓ Symbol table is central to semantic analysis.
- ✓ It helps in maintaining scopes.
- ✓ It supports memory allocation.
- ✓ It is used in intermediate code generation and optimization.
- ✓ Efficient symbol table management improves compiler performance.

7. Implementation of Symbol Table :

① Linear List
② Binary Search Tree
③ Hash Table
④ Trie (Prefix Tree)
⑤ Scope based Organization

Comparison of Implementations :

Method	Insertion	Search	Deletion	Space	Remarks
Linear List	O(1)	O(n)	O(n)	Less	Simple but slow search
Binary Search Tree	O(log n)	O(log n)	O(log n)	More	Efficient, ordered
Hash Table	O(1) avg	O(1) avg	O(1) avg	More	Best for large data
Trie	O(m)	O(m)	O(m)	More	Good for strings
Scope Based	Varies	Varies	Varies	More	Helps manage scopes

8. Hash Table Implementation :

Hash function maps identifier to an index.

Index = h(id) = id % TableSize

Example :

Let TableSize = 7

Identifier	ASCII Sum	Index (mod 7)
a	97	6
b	98	0
x	120	1
sum	115+117+109=351	0
var	118+97+114=329	6

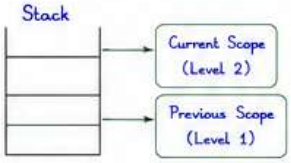
Hash Table (Size = 7)

0 →	b, sum
1 →	x
2 →	—
3 →	—
4 →	—
5 →	—
6 →	a, var

If two identifiers map to same index, we handle collisions using chaining or open addressing.

9. Scope Management :

Scopes are managed using a stack of symbol tables.
When entering a new scope → Push new table.
When exiting scope → Pop table and delete its entries.



Operations
EnterScope() : Push
ExitScope() : Pop

10. Summary :

- Symbol table stores all important information about identifiers.
- It supports various compiler phases.
- It handles scopes and detects multiple declarations.
- It is implemented using different data structures.
- Efficient symbol table management is crucial for compiler design.

Exam Tip
- Always mention information stored in symbol table.
- Write about operations performed on symbol table.
- Diagrams of hash table or scope stack increase marks.

12. Error Handling in Compilers

1. Introduction :

Errors are unavoidable in any source program.

A compiler must detect, report and (if possible) recover from errors so that compilation can continue and more errors can be found in a single run.

Types of Errors
- ✓ Lexical Errors
- ✓ Syntax Errors
- ✓ Semantic Errors
- ✓ Run-time Errors

2. Need for Error Handling :

- Improves reliability of compiler.
- Helps programmer to debug the program easily.
- Detects maximum number of errors in a single compilation.
- Prevents compiler from crashing.
- Provides meaningful messages to the user.

3. Sources of Errors :

Error Type	Description
Lexical Errors	Error in spelling, invalid tokens, illegal characters, etc.
Syntax Errors	Violation of grammar rules.
Semantic Errors	Logically incorrect statements even if syntax is correct.
Run-time Errors	Errors detected during execution (e.g., divide by zero).

4. Phases Where Errors Occur :

Compiler Phase	Errors Detected
Lexical Analysis	Illegal characters, invalid tokens, unrecognized words
Syntax Analysis	Missing symbols, unmatched parentheses, etc.
Semantic Analysis	Type mismatch, undeclared identifiers, etc.
Runtime (Execution)	Division by zero, array out of bound, etc.

5. Error Handling Strategies :

① Error Detection – Identify the error.
② Error Reporting – Inform user with meaningful message.
③ Error Recovery – Resume compilation after error.
④ Error Termination – Stop compilation if recovery not possible.

6. Error Detection & Reporting :

- Detect error as early as possible.
- Explain the error clearly.
- Show the line number and position of error.
- Example of error message :

```
Error : Syntax Error
Line 7 : Missing ';' before '}'
```

7. Error Recovery Techniques :

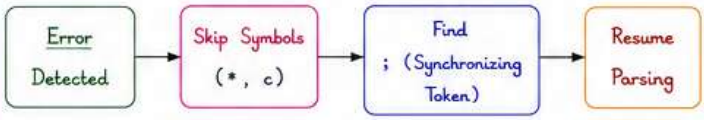
Goal : Continue compilation after detecting an error.

Technique	Description
Panic Mode	Skip input symbols until a synchronizing token is found.
Phrase Level Recovery	Make small changes in the input to correct the error.
Error Tordiuction	Add productions in grammar to handle common errors.
Global Correction	Find the minimum cost set of changes to correct the program.

8. Panic Mode Error Recovery (Example) :

If an error is found, skip symbols until a synchronizing token (like ';', '}', or '$') is found.

Example : Consider input → a = b + * c ; d = e ;



9. Phrase Level Recovery (Example) :

Make local correction to continue parsing.

Example : Input → a = b + * c ;

The parser may assume a missing operand 'x'

Corrected input → a = b + x * c ;

10. Error Productions :

Add error productions in grammar to handle common mistakes.

Example :

```
stmt → if ( E ) stmt
     | if ( E ) stmt else stmt
     | error ) stmt    // to handle missing (
```

11. Semantic Errors (Examples) :

- Use of undeclared variable.
- Type mismatch in expression.
- Wrong number or type of arguments in function call.
- Return type mismatch.

Example :

```
int a ;
a = "hello" ;
Error : Type mismatch
 (int = string)
```

12. Run-time Errors (Examples) :

- Division by zero.
- Array index out of bound.
- Null pointer reference.
- File not found.

Example :

```
int a = 10, b = 0 ;
int c = a / b ;
Error : Division by zero
```

13. Good Error Message Guidelines :

① Be precise and understandable.
② Show line number and position.
③ Indicate type of error.
④ Provide possible reason.
⑤ Suggest possible correction (if possible).

Example Message

```
Error : Undefined Variable
Line 5, Column 12
Variable 'x' is not declared.
Did you mean 'a' ?
```

14. Summary :

- Error handling is an essential part of compiler.
- Errors can occur in lexical, syntax, semantic and run-time phases.
- Compiler must detect, report and recover from errors.
- Different recovery techniques help in finding more errors.
- Clear error messages help programmer to debug easily.
- Good error handling makes compiler user-friendly and robust.

Exam Tip
- Always distinguish between error detection, reporting and recovery.
- Learn panic mode and phrase level recovery with examples.
- Write examples of lexical, syntax, semantic and run-time errors.
- Draw flow of error handling in compiler.

13. Code Optimization

1. Introduction :

Code optimization is the process of improving the intermediate code to make the target code more efficient in terms of time and space.
It does not change the meaning of the program.

Aim :
Better code Quality
Faster Execution

2. Need for Code Optimization :

* Reduces execution time.
* Reduces memory usage.
* Improves program efficiency.
* Essential for resource constrained systems.
* Better performance with same functionality.

3. Properties of Optimizations :

Property	Description
Semantics Preserving	Does not change the meaning of the program.
Safe	Must not alter program behavior.
Profitable	Optimization cost < Benefit gained.
Target Machine Dependent	Some optimizations depend on target machine.
Local / Global	Can be applied on small part or whole code.

4. Levels of Optimization :

* Machine Independent Optimization (Middle-End)
* Machine Dependent Optimization (Back-End)
* Peephole Optimization

5. Machine Independent Optimizations :

(a) Constant Folding :

Evaluate constant expressions at compile time.

Example : `x = 3 * 4 + 2`
Optimized : `x = 12 + 2`
`x = 14`

Benefits
- ✓ Less computation
- ✓ Faster execution
- ✓ Smaller code size
- ✓ Better register allocation

(b) Constant Propagation :

Replace a variable by its constant value.

Example :
```
a = 10
b = a + 5
```
Optimized :
```
a = 10
b = 15
```

(c) Common Subexpression Elimination :

Remove repeated computation of same expression.

Example :
```
t1 = a + b
t2 = a + b
t3 = t1 * c
```
Optimized :
```
t1 = a + b
t2 = t1
t3 = t1 * c
```

Note : Recomputing same expression wastes time and resources.

(d) Dead Code Elimination :

Remove code which does not affect the output.

Example :
```
a = 10      (never used)
b = 20      (never used)
c = a + b
print c
```
Optimized :
```
c = a + b
print c
```

Important Points
* Write definitions with examples.
* Draw flow diagram wherever possible.
* List optimization techniques with examples.
* Write advantages/benefits clearly.

6. Machine Dependent Optimizations :

Optimizations that depend on target machine architecture.

Examples :
1. Register allocation and assignment
2. Instruction scheduling
3. Exploiting addressing modes
4. Loop optimization.

Goal : Generate efficient machine code for better performance.

7. Peephole Optimization :

Optimize code by looking into a small window (peephole) of instructions.

Example :

Before Optimization	After Optimization
`x = x + 0`	No operation
`x = x * 1`	No operation
`x = x * 2`	`x = x + x`
`x = x - 0`	No operation
`goto L1` `L1 :`	Removed

Note : Simple but very effective optimization.

8. Loop Optimization :

Improve efficiency of loops.

Techniques :
* Loop Invariant Code Motion
* Loop Unrolling
* Strength Reduction
* Induction Variable Optimization

Example :
```
for (i = 1; i <= n; i++)
    x = x + 0;   // useless
```
Can be removed.

9. Code Motion :

Move code to a better position without changing semantics.
* Loop Invariant Code Motion : Move code outside loop if it does not change within loop.
* Code Hoisting : Move computations upward.
* Code Sinking : Move computations downward.

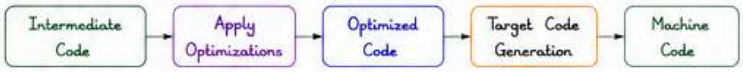
10. Instruction Scheduling :

Reorder instructions to reduce stalls and improve pipeline performance.
Goal : Better utilization of CPU resources.

11. Benefits of Code Optimization :

* Faster execution
* Reduced memory usage
* Better CPU performance
* Smaller code size
* Lower power consumption

12. Optimization Process :



Important Steps
* Analyze the code
* Apply suitable optimization
* Check correctness
* Generate optimized code

13. Summary :

* Code optimization improves performance.
* It is semantics preserving.
* Applied at different levels.
* Uses various techniques to reduce time and space.
* Essential part of compiler design.

* Write definitions with examples.
* Draw flow diagram wherever possible.
* List optimizations techniques with examples.

14. Code Generation

1. Introduction :

Code Generation is the phase of compiler that takes intermediate code as input and produces equivalent target code in the target language (Assembly / Machine code).

Goal :
- ✓ Correct code
- ✓ Efficient code
- ✓ Fast execution
- ✓ Less memory

2. Need for Code Generation :

- Converts intermediate code into target language.
- Makes program ready to execute on target machine.
- Utilizes registers and memory efficiently.
- Improves execution speed.
- Reduces code size.

3. Input to Code Generator :

- Intermediate Code (Three Address Code)
- Symbol Table
- Machine Description
- Literal Table

Output : Target Code (Assembly / Machine Code)

4. Steps in Code Generation :

Input Intermediate Code → Select Instruction Sequence → Register Allocation & Assignment → Generate Target Code → Code Optimization

5. Issues in Code Generation :

- Efficient use of registers.
- Optimal instruction selection.
- Memory allocation.
- Evaluation order of expressions.
- Control flow and jumps.

6. Instruction Selection :

Select a sequence of target instructions that perform the same operation as intermediate code.

Intermediate Code	Possible Instruction Sequences	Choice

x = y + z	ADD x, y, z
	LOAD x, y
	ADD x, z
		ADD x, y, z

x = y * z	MUL x, y, z
	LOAD x, y
	MUL x, z
		MUL x, y, z

x = y	MOV x, y	MOV x, y

7. Register Allocation & Assignment :

Assign variables to machine registers to reduce memory access.

Variable	Location	Notes
a	R1	In register R1
b	R2	In register R2
c	Memory	Spilled to memory

8. Memory Management :

- Allocate memory for variables and temporaries.
- Reuse memory locations when values are no longer needed.
- Helps in reducing memory usage.

Important Points :
- ✓ Good code generation improves execution speed.
- ✓ Efficient register use reduces memory access.
- ✓ Instruction selection and optimization reduce code size.
- ✓ Code generator uses machine description for target code.

9. Example :

Intermediate Code (Three Address Code) :

1. t1 = a + b
2. t2 = t1 * c
3. d = t2 - e

a, b, c, d, e → Variables
t1, t2 → Temporaries

Register Allocation :

a → R1
b → R2
c → R3
e → R4
t1 → R5
t2 → R6

Generated Target Code (Assembly)

Step	Instruction	Explanation
1	ADD R5, R1, R2	t1 = a + b
2	MUL R6, R5, R3	t2 = t1 * c
3	SUB R7, R6, R4	d = t2 - e
4	MOV d, R7	Store result in d

10. Code Optimization Techniques in Code Generation :

- Peephole Optimization
- Common Subexpression Elimination
- Dead Code Elimination
- Constant Folding
- Strength Reduction
- Loop Optimization

Example (Peephole)
```
MOV R1, R1
ADD R2, R1, #0
```
→ Removes useless operations.

11. Types of Target Code :

Type	Description
Assembly Code	Human readable code using mnemonics.
Machine Code	Binary code (0s and 1s) executed by CPU.
Relocatable Code	Code that can be loaded at any memory location.
Absolute Code	Code with fixed memory location.

12. Factors Affecting Code Generation :

- Machine architecture
- Instruction set
- Number of registers
- Addressing modes
- Memory organization

Note : Code generation is highly dependent on hardware architecture.

13. Code Generation Process Flow :

Intermediate Code → Instruction Selection → Register Allocation → Target Code Generation → Optimized Target Code

14. Summary :

- Code generation converts intermediate code into target code.
- It selects efficient instruction sequences.
- Registers are allocated to reduce memory access.
- Various optimization techniques improve the quality of code.
- Good code generation results in fast and efficient programs.

Exam Tip:
- Write definition with diagram wherever possible.
- List steps and techniques with examples.
- Draw flow of code generation process.
- Write advantages of code generation clearly.

15. Bootstrap Compiler

1. Introduction :

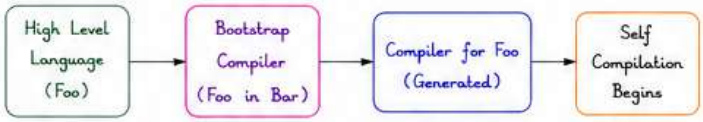
A Bootstrap Compiler is a compiler written in the same high level language for which it is intended to compile programs. It is used to build a compiler for a new language using some existing compiler.

Goal :
- ✓ Independent
- ✓ Self-hosting
- ✓ Portable
- ✓ Reusable
- ✓ Efficient

2. Need for Bootstrap Compiler :

- Eliminates dependency on assembly language.
- Easier to develop and maintain.
- Portability across different machines.
- Improves reliability and readability.
- Once written, it can be used to compile itself.

3. Bootstrap Compiler Concept :



Initially, a small compiler (bootstrap compiler) is written in another language (Bar). This compiler generates a full compiler for language Foo.

4. Characteristics :

- Written in high level language.
- Used to generate compiler for same language.
- After few stages, it can compile itself.
- No need of assembly level programming.

Example :
C compiler written in C language.

5. Phases in Bootstrap Compilation :

① Write a small bootstrap compiler in language L, using language M.
② Use it to compile the full compiler written in L.
③ Use the generated compiler to compile itself.
④ Repeat the process to get improved version.
⑤ Final compiler is completely written in L.

Note : L Language for which compiler is written
M = Language in which bootstrap is written

6. Example - Building a C Compiler Using Bootstrap :

Stage	Compiler Used	Description
1	Small compiler (C in Assembly)	To compile full C compiler
2	Generated C compiler	Compiles itself
3	New C compiler	More efficient code
4	Optimized C compiler	Final production compiler

7. Advantages :

- Easy to implement.
- Easier maintenance and modification.
- Portability to other machines.
- Better readability and reliability.
- No need to know assembly level details.

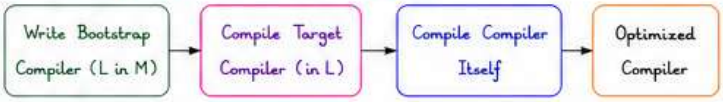
Example :
Many modern compilers like C, C++, Java compilers use bootstrap techniques.

8. Limitations :

- Initial bootstrap compiler must be written in some other language.
- More compilation stages are required initially.
- Performance may be low in initial stages.
- Needs more time and resources in starting.

Important Points:
- ✓ Bootstrap compiler reduces dependency on machine specific code.
- ✓ Self-hosting property makes compiler more powerful.
- ✓ It improves portability and maintainability.
- ✓ Modern compilers are mainly built using bootstrap technique.

9. Bootstrap Compilation Process Diagram :



Legend :

L = Target Language

M = Machine Language / Another Language

10. Self Compilation Example (C Compiler) :

Step	Action
1	Small C compiler is written in Assembly.
2	This compiler compiles full C compiler (written in C).
3	The generated compiler compiles itself.
4	A new improved compiler is generated.
5	This process repeats till final optimized compiler is ready.

Bootstrap Code (Pseudo)

```
void main(){
   read_source();
   analyze();
   generate_code();
   write_target();
}
```

Compiles Itself →

Generated Compiler (Same Language)

```
void main(){
   read_source();
   analyze();
   optimize();
   generate_code();
   write_target();
}
```

11. Applications :

- Construction of compilers for new high level languages.
- Used in system software development.
- Used in cross compilers and portable compilers.
- Helpful in educational compiler projects.

12. Comparison Table :

Feature	Bootstrap Compiler	Traditional Compiler
Language Used	High Level Language	Assembly / Machine Level
Portability	High	Low
Development Time	More (Initial)	Less (Initial)
Maintenance	Easy	Difficult
Readability	High	Low
Self Compilation	Possible	Not Possible

13. Summary :

- Bootstrap compiler is written in same high level language.
- It is used to generate full compiler for that language.
- It eliminates dependency on assembly language.
- It improves portability, maintainability and reliability.
- Modern compilers are built using bootstrap technique.
- It is an important concept in compiler construction.

Exam Tip:
- Define bootstrap compiler with a neat example.
- Draw bootstrap compilation process diagram.
- Write advantages and limitations clearly.
- Self compilation stages are important.

16. Compiler Construction Tools

1. Introduction :

Compiler construction tools are software packages which help in the automatic construction of compilers. They generate lexical analyzer, parser and other components automatically from high level specifications.

Goal :
- ✓ Save time
- ✓ Reduce errors
- ✓ Portable
- ✓ Easy to use
- ✓ Efficient

2. Need of Compiler Construction Tools :

- Manual compiler construction is time consuming.
- Difficult to debug and maintain.
- Tools generate code automatically.
- Help in designing reliable and efficient compilers.
- Support for lexical analysis, parsing and code generation.

3. Commonly Used Compiler Construction Tools :

Tool	Purpose
LEX / Flex	Lexical analyzer generator.
YACC / Bison	Parser generator (syntax analyzer).
ANTLR	Parser generator for multiple languages.
COBOL Tools	For COBOL language compiler construction.
LISP Tools	For LISP language compiler construction.
LLVM	Modern compiler infrastructure.

4. LEX / Flex (Lexical Analyzer Generator) :

- Used to generate lexical analyzer from regular expressions.
- Input : Specification of tokens using regular expressions.
- Output : C program (lexical analyzer).
- Generates functions : yylex(), yywrap().

LEX Input (Regular Expressions) → LEX → C Program (Lexical Analyzer)

Example (LEX Specification) :

```
digit       [0-9]
id          [a-zA-Z_] [a-zA-Z0-9_]*
%%
{digit}+    { printf("NUM\n"); }
{id}        { printf("ID\n"); }
[ \t\n]+    ;
.           { printf("Other\n"); }
%%
```

5. YACC / Bison (Parser Generator) :

- Used to generate parser (syntax analyzer) from context free grammar.
- Input : Grammar rules with actions.
- Output : C program (parser).
- Generates functions : yyparse(), yylex().

YACC Input (Grammar Rules + Actions) → YACC → C Program (Parser)

Example (YACC Specification) :

```
%{ #include <stdio.h>  %}
%token ID NUM
%%
E : E '+' T    { printf("E -> E + T\n"); }
  | T          { printf("E -> T\n"); } ;
T : T '*' F    { printf("T -> T * F\n"); }
  | F          { printf("T -> F\n"); } ;
F : '(' E ')' |  ID  |  NUM ;
%%
```

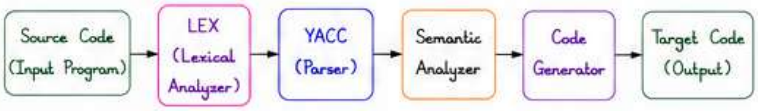
6. ANTLR (Another Tool for Language Recognition) :

- Generates lexer, parser and tree walker.
- Supports multiple programming languages.
- Works with LL(*) parsing techniques.
- Used for language processing tools.

Example :
ANTLR is used in modern IDEs and tools like IntelliJ, Eclipse, etc.

Important Points :
- ✓ Tools reduce manual effort.
- ✓ LEX handles lexical analysis.
- ✓ YACC handles syntax analysis.
- ✓ Together they build compiler front end.
- ✓ Modern tools provide complete compiler support.

7. Role of LEX and YACC Together :



Working :
1. LEX reads input and generates tokens.
2. Tokens are sent to YACC.
3. YACC checks syntax using grammar rules.
4. If correct, semantic analysis and code generation are done.
5. Target code is produced.

8. Comparsion of LEX and YACC :

Feature	LEX / Flex	YACC / Bison
Purpose	Lexical Analyzer Generator	Parser Generator
Input	Regular Expressions	Context Free Grammar
Output	C Program (yylex())	C Program (yyparse())
Handles	Tokens (Lexemes)	Syntax (Structure)
Language	Regular Language	Context Free Language
Stage	Lexical Analysis	Syntax Analysis

9. Other Modern Tools :

- LLVM : Infrastructure for modern compiler design.
- COBOL Tools : Used for COBOL compiler development.
- LISP Tools : Used for LISP compiler development.
- IR Tools : Generate and optimize intermediate code.

Note :
These tools are essential for building modern compilers quickly and efficiently.

10. Advantages of Compiler Construction Tools :

- Save development time.
- Reduce human errors.
- Portable across platforms.
- Easy to maintain and modify.
- Support powerful features.
- Better reliability.

Example :
Using LEX and YACC, we can build a complete compiler front and in less time.

11. Limitations :

- Tools are language dependent.
- Limited for complex language features.
- Need knowledge of tool specifications.
- Generated code may not be optimized.

12. Summary :

- Compiler construction tools automate the process of compiler building.
- LEX is used for lexical analyzer generation.
- YACC is used for parser generation.
- Together they form the front end of a compiler.
- Other tools provide support for modern compiler development.
- These tools improve productivity and reduce errors.

Exam Tip
- Write about LEX, YACC with examples.
- Draw role of LEX and YACC together.
- List advantages and limitations.
- Write applications of compiler construction tools.