COMPILER DESIGN STEP MATERIAL

UNIT-I

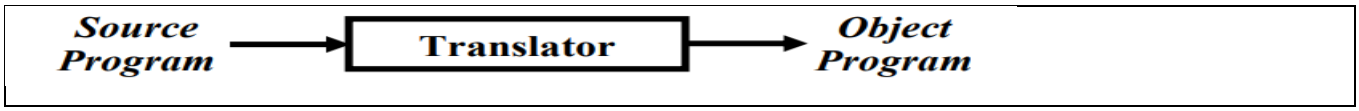
Short Answer Questions:

1. What is translator? Give examples of translators.

Ans:

Translator is a kind of program that takes one kind of program as input and converts it into another form.

The input program is called source program and output program is called target (Object) program.



There are four types of translators according to the source language as well as target language:

1.Compiler

2. Interpreter

3. Assembler

4. Pre-processor

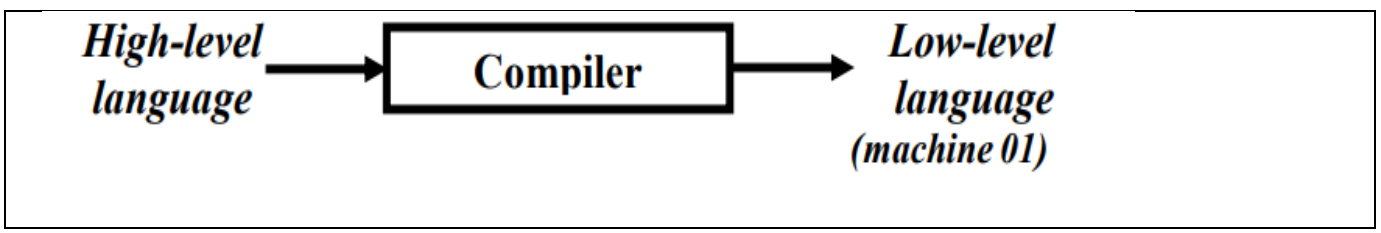
---

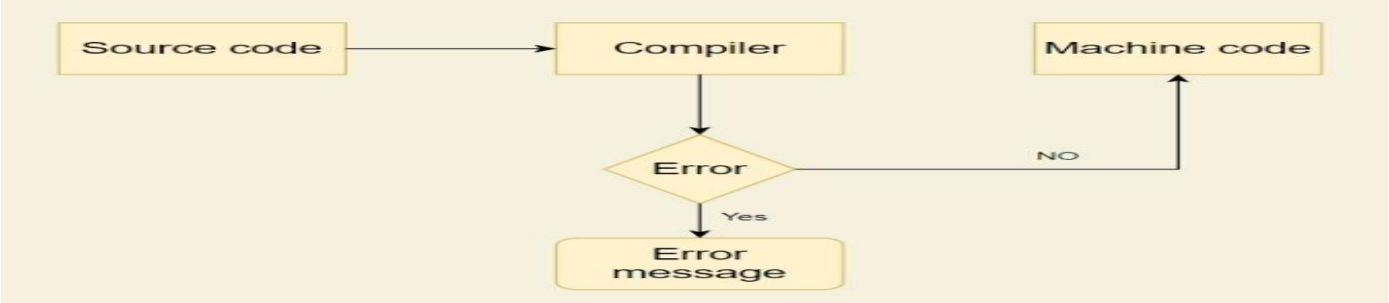
2. What is compiler?

Ans:

Compiler is a translator that translates a high-level language program such as FORTRAN, PASCAL, C++, to low-level language program such as an assembly language or machine language.

Note: The translation process should also report the presence of errors in the source program.





---

3. What is input buffering?

Ans:

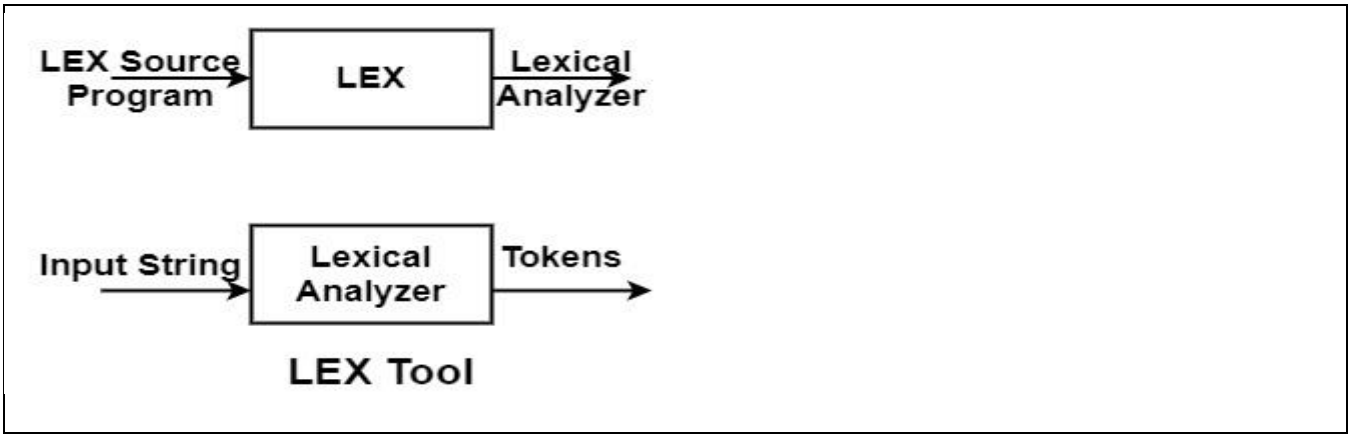
- The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block.
- The size of the buffer can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled.
- For example, a compiler for a high-level programming language may use a larger buffer than a compiler for a low-level language, since high-level languages tend to have longer lines of code.
- One of the main advantages of input buffering is that it can reduce the number of system calls required to read input from the source code.
- There are also some potential disadvantages to input buffering. For example, if the size of the buffer is too large, it may consume too much memory, leading to slower performance.

---

4. What is lex? Give the structure of LEX.

Ans:

- LEX is a tool or software or program which automatically generates a lexical analyser.
- It takes as its input a LEX source program and produces lexical Analyzer as its output.
- Lexical Analyzer will convert the input string entered by the user into tokens as its output.



---

5. What are steps involved in Optimizing the DFA.

Ans:

The following steps involved in Optimizing the DFA are

1.Create Syntax tree

2.Numbering the leaf node

3.Find the nullable of each node

4.Find the firstpos of each node

5.Find the lastpos of each node

6.Finf followpos of each node

Long Answer Questions:

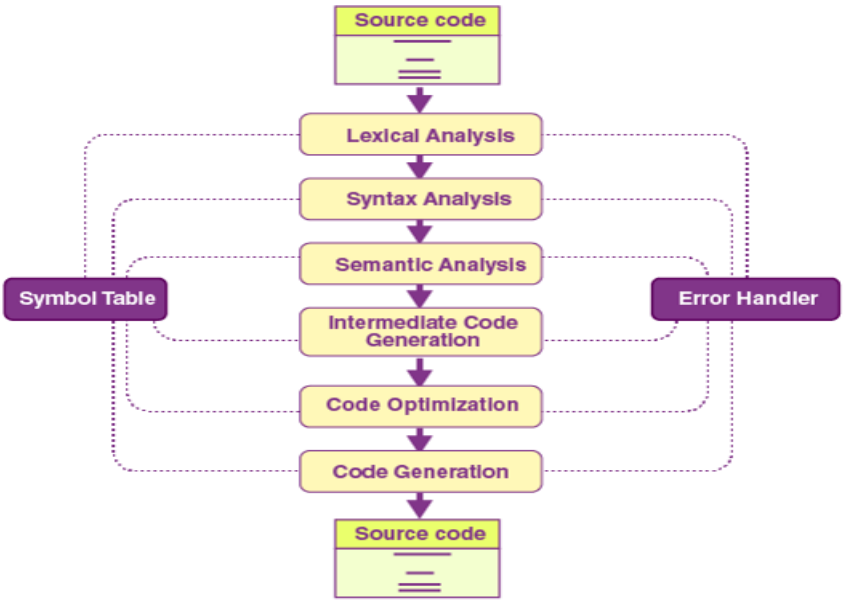
1. Explain the various phases of compiler with an illustrative example.

Ans:

Structure of a compiler or phases of a compiler:

The 6 phases of a compiler are:

1. Lexical Analysis
2. Syntactic Analysis or Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



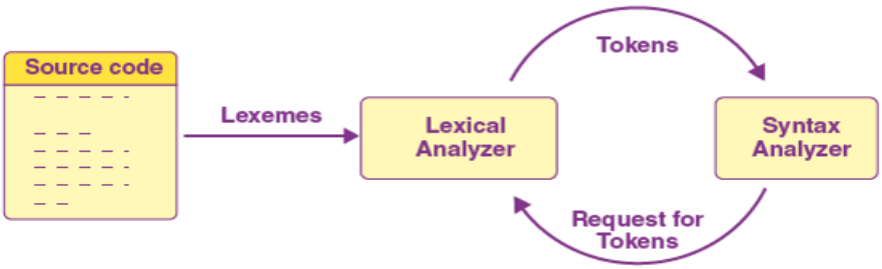
1.Lexical Analysis:

- Lexical analysis or Lexical analyzer is the initial stage or phase of the compiler.
- This phase scans the source code and transforms the input program into a series of a token.
- A token is basically the arrangement of characters that defines a unit of information in the source code.

Roles and Responsibility of Lexical Analyzer

The lexical analyzer performs the following tasks-

- The input characters are read by the lexical analyzer from the source code.
- The lexical analyzer is responsible for removing the white spaces and comments from the source program.
- It corresponds to the error messages with the source program.
- It helps to identify the tokens.



Example:

INPUT: Source code

Sum=oldsum+rate*50

sum -> Identifier- (id1)

= -> Operator - Assignment

oldsum -> Identifier- (id2)

+ -> Operator - Binary Addition

rate -> Identifier- (id3)

* -> Operator - Multiplication

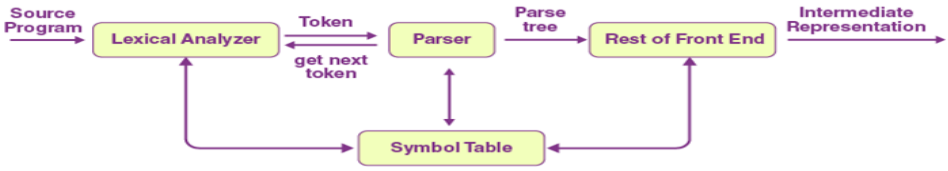
50 -> Constant - (id4)

OUTPUT: Tokens

(id1) = (id2) + (Id3)*(id4)

2. Syntax Analysis:

- The Syntax analysis is the second stage or phase of the compiler.
- It accepts tokens as input and provides a parse tree as output. It is also known as parsing in a compiler.



Roles and Responsibilities of Syntax Analyzer

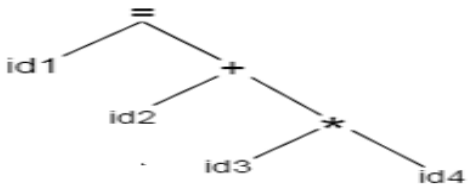
- Note syntax errors.
- Helps in building a parse tree.
- Acquire tokens from the lexical analyzer.
- Scan the syntax errors, if any.

Example:

INPUT: Tokens

(id1) = (id2) + (Id3)*(id4)

OUTPUT: Parse Tree



3.Semantic Analysis:

- semantic analysis is the third stage or phase of the compiler.
- It scans whether the parse tree follows the guidelines of language.
- semantic analyzer defines the validity of the parse tree, and the syntax tree comes as an output.

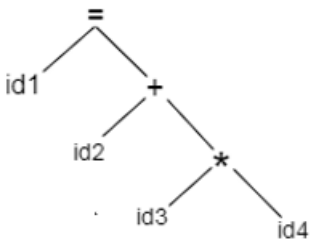
Roles and Responsibilities of Semantic Analyzer:

- Generate syntax trees from parse tree.
- It notifies semantic errors.

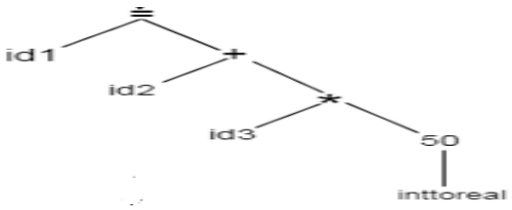
- Scanning for semantic errors.

Example:

INPUT: Parse Tree

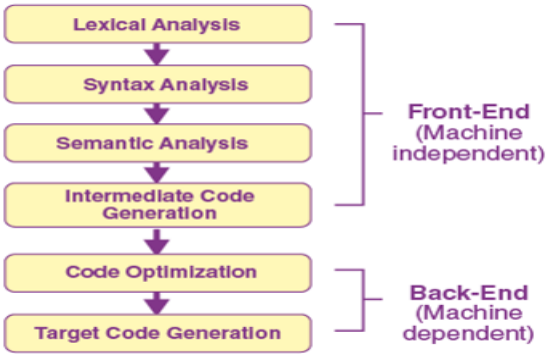


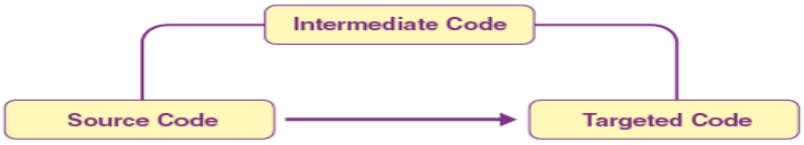
OUTPUT: Syntax Tree



4. **Intermediate Code Generation:**

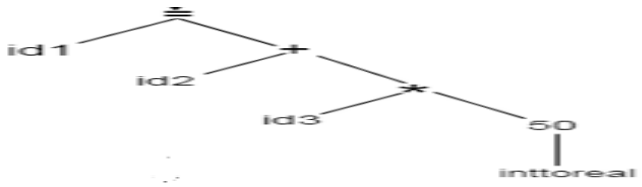
- The parse tree is semantically confirmed; now, an intermediate code generator develops three address codes.
- A code that is neither high-level nor machine code, but a middle-level code is an intermediate code.
- We can translate this code to machine code later.
- This stage serves as a bridge or way from analysis to synthesis.





Example:

INPUT: Syntax Tree



OUTPUT: Intermediate Code.

temp1=intorreal(50)

temp2=id3*temp1

temp3=id2+temp2

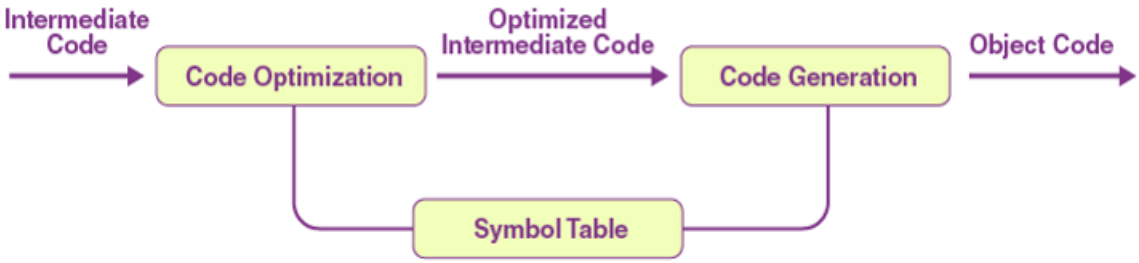
id1=temp3

5. **Code optimizer:**

- It is used to enhance the intermediate code. This way, the output of the program is able to run fast and consume less space.
- To improve the speed of the program, it eliminates the unnecessary strings of the code and organizes the sequence of statements.

Roles and Responsibilities of Semantic Analyzer:

- Remove the unused variables and unreachable code.
- Enhance runtime and execution of the program.



Example:

INPUT: Intermediate Code.

temp1=intorreal(50)

temp2=id3*temp1

temp3=id2+temp2

id1=temp3

OUTPUT: Optimized Intermediate Code.

temp1=id3*50.0

Id1=id2+temp1

6. **Code Generator:**

- The final stage of the compilation process is the code generation process.
- it tries to acquire the intermediate code as input which is fully optimized and map it to the machine code or language.

Roles and Responsibilities:

- Translate the intermediate code to target machine code.
- Select and allocate memory spots and registers.

Example:

INPUT: Optimized Intermediate Code.

temp1=id3*50.0

Id1=id2+temp1

OUTPUT: Target machine code.

MOVF id3, R1

MULF #50.0*R1

MOVF id2, R2

ADDF R2, R1

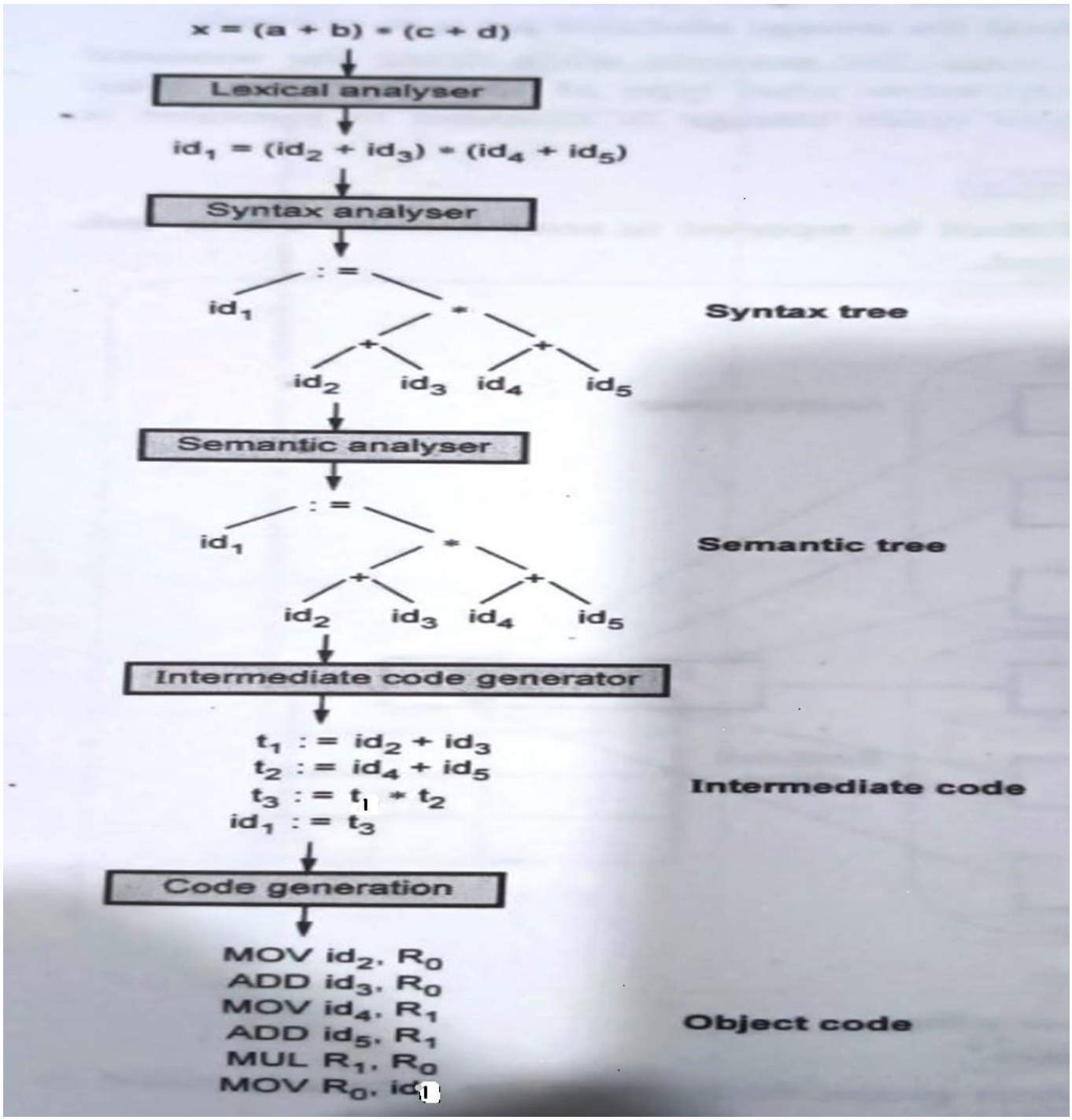
MOVF R1, id1

---

2. Show the output generated by each phase for the following expression

x= (a + b) * (c + d).

Ans:



---

3. What is input buffering? Explain How is input buffering implemented.

Ans:

Input Buffering:

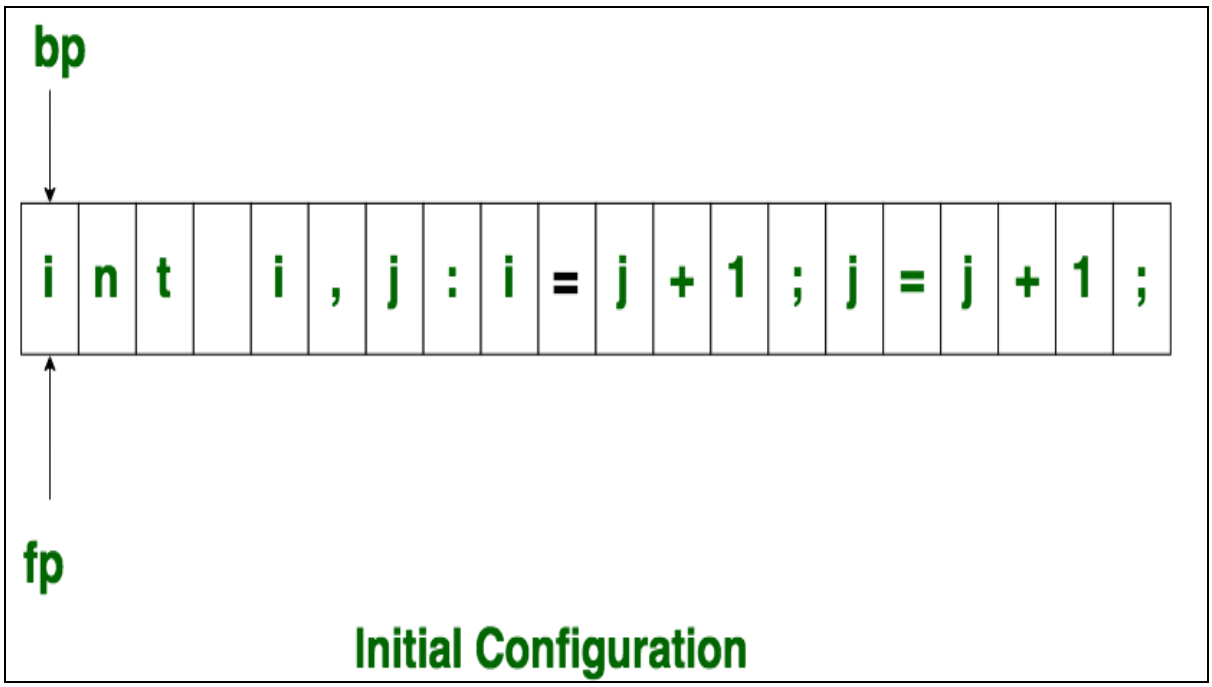
- The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block.

- The size of the buffer can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled.
- For example, a compiler for a high-level programming language may use a larger buffer than a compiler for a low-level language, since high-level languages tend to have longer lines of code.
- One of the main advantages of input buffering is that it can reduce the number of system calls required to read input from the source code.
- There are also some potential disadvantages to input buffering. For example, if the size of the buffer is too large, it may consume too much memory, leading to slower performance.

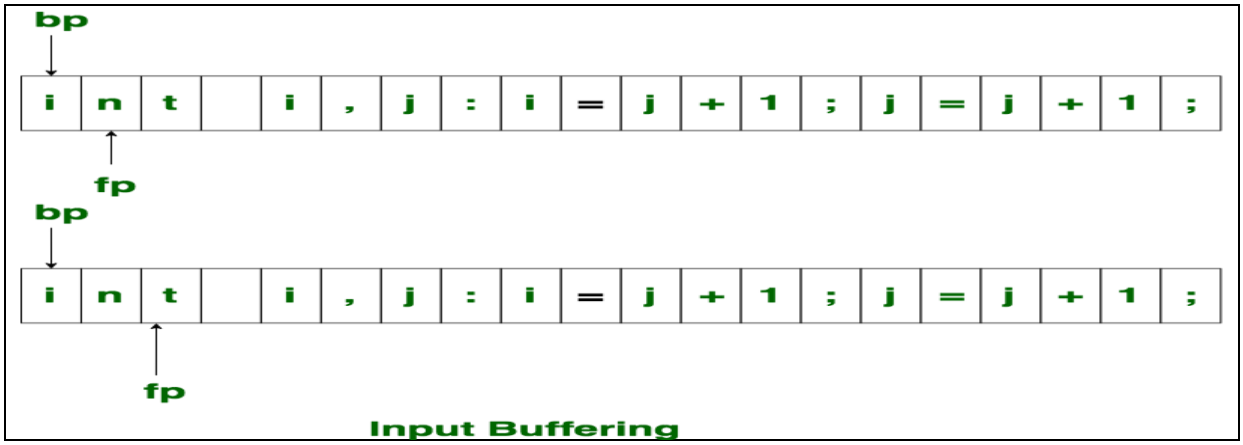
Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens −

1. **Begin Pointer (bp)** − It points to the beginning of the string to be read.
2. **Forward Pointer (fp)** − It moves ahead to search for the end of the token.

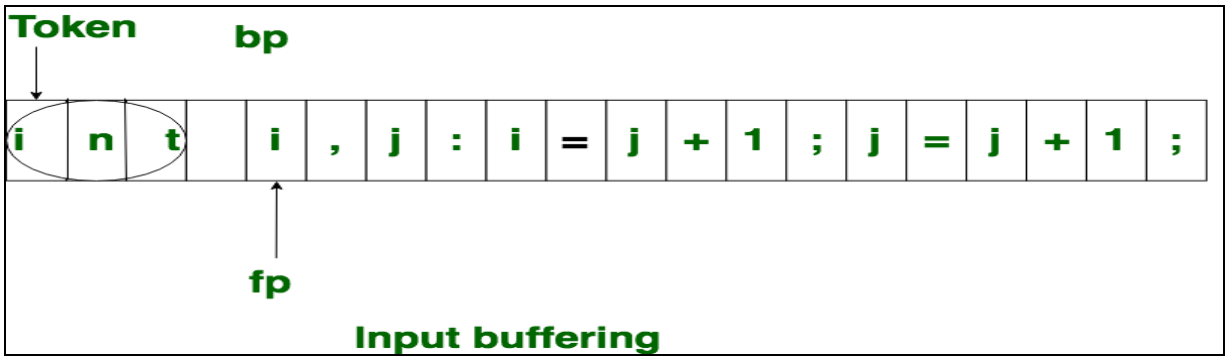
Example: For statement int i , j ; I = j + 1 ; j = j + 1;
Both pointers start at the beginning of the string, which is stored in the buffer.



Forward Pointer(fp) scans buffer until the token is found.



Input Buffering



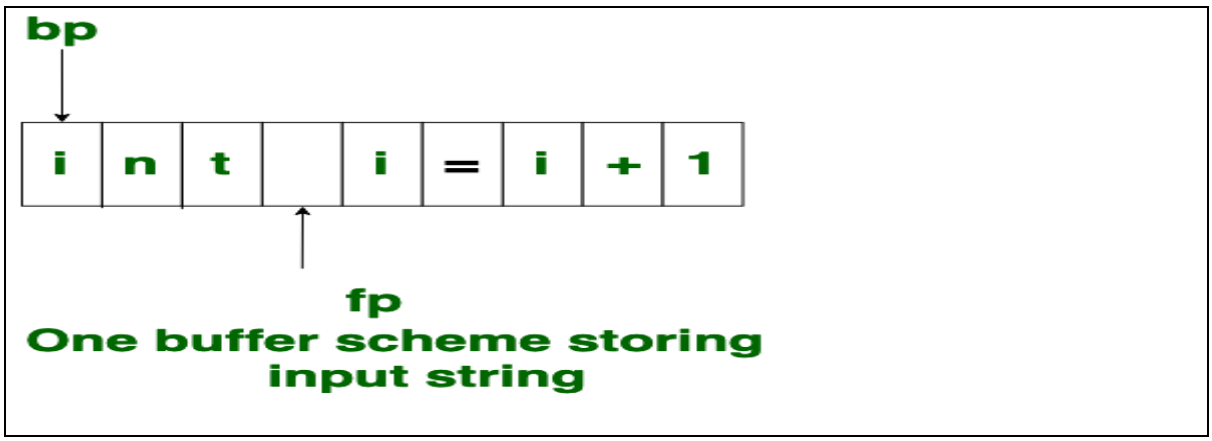
Input buffering

There are two methods used in this context:

1. One Buffer Scheme
2. Two Buffer Scheme

1.One Buffer Scheme:

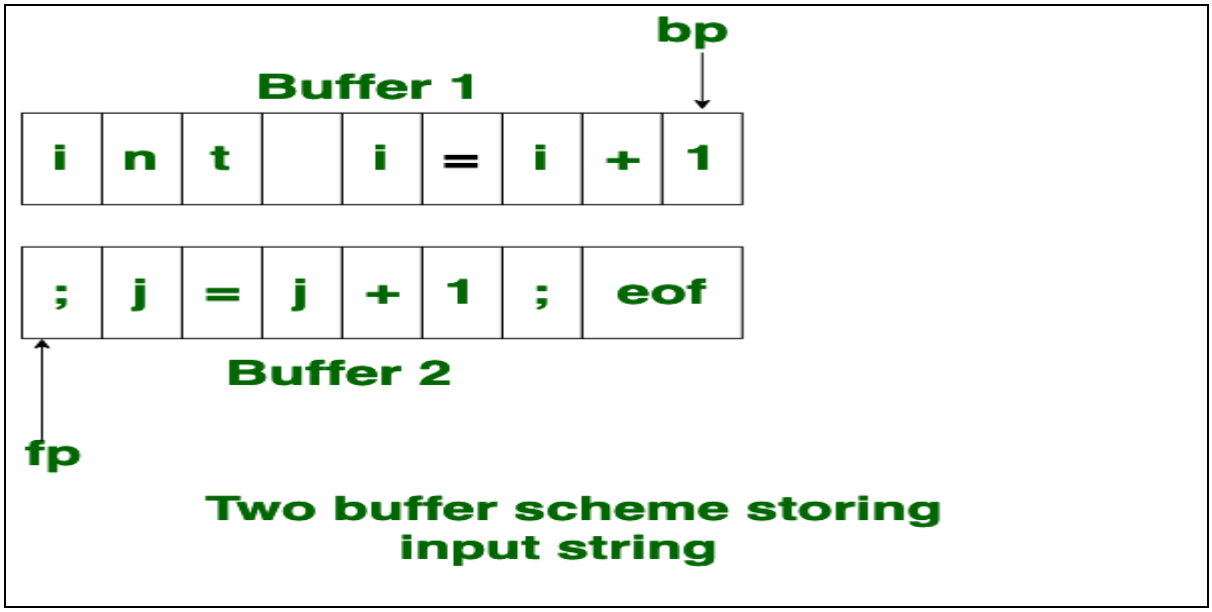
In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



One buffer scheme storing input string

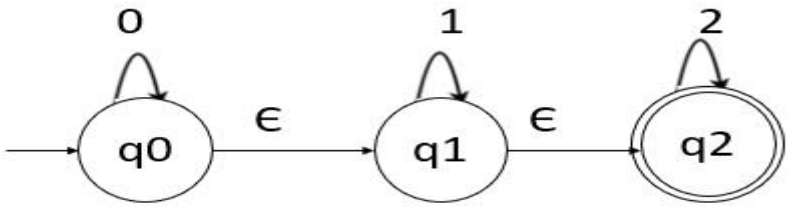
2.Two Buffer Scheme:

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled.



---

4. Convert the given NFA with epsilon to NFA without epsilon.



Ans:

ε-closure of each state i.e.,

ε-closure(q0) = {q0, q1, q2}

ε-closure(q1) = {q1, q2}

ε-closure(q2) = {q2}

Now we will obtain δ^1 transitions for each state on each input symbol as shown below −

δ'(q0, 0) = ε-closure(δ(q0, q1,q2), 0))

= ε-closure(δ(q0, 0) ∪ δ(q1, 0) U δ(q2, 0))

= ε-closure(q0 U Φ ∪ Φ)

= ε-closure(q0)

= {q0, q1, q2}

δ'(q0, 1) = ε-closure(δ(q0,q1,q2), 1))

= ε-closure(δ(q0, 1) ∪ δ(q1, 1) U δ(q2, 1))

= ε-closure(Φ ∪q1 U Φ)

= ε-closure(q1)

= {q1, q2}

δ'(q0, 2) = ε-closure(δ(q0,q1,q2), 2))

= ε-closure(δ(q0, 2) ∪ δ(q1, 2) U δ(q2, 2))

= ε-closure(Φ U ΦU q2)

= ε-closure(q2)

= {q2}

δ'(q1, 0) = ε-closure(δ(q1,q2), 0))

= ε-closure(δ(q1, 0) U δ(q2, 0))

= ε-closure(Φ ∪ Φ)

= ε-closure(Φ)

= Φ

δ'(q1,1) = ε-closure(δ(q1,q2), 1))

= ε-closure(δ(q1, 1) U δ(q2, 1))

= ε-closure(q1 ∪ Φ)

= ε-closure(q1)

= {q1,q2}

δ'(q1, 2) = ε-closure(δ(q1,q2), 2))

= ε-closure(δ(q1, 2) U δ(q2, 2))

= ε-closure(Φ ∪ q2)

= ε-closure(q2)

= {q2}

δ'(q2, 0) = ε-closure(δ(q2), 0))

= ε-closure(δ(q2, 0))

= ε-closure(Φ) = Φ

δ'(q2, 1) = ε-closure(δ(q2), 1)

= ε-closure(δ(q2, 1))

= ε-closure(Φ)

= Φ

δ'(q2, 2) = ε-closure(δ(q2), 2))

= ε-closure(δ(q2, 2))

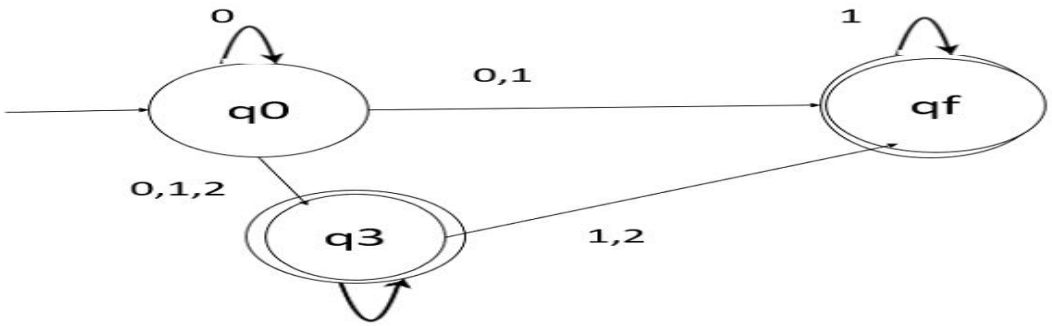
= ε-closure(q2)

= {q2}

The **transition table** is given below −

States\inputs	**0**	**1**	**2**
q0	{q0,q1,q2}	{q1,q2}	{q2}
q1	Φ	{q1,q2}	{q2}
q2	Φ	Φ	{q2}

NFA without epsilon



Here, q0, q1, q2 are final states because ε-closure(q0), ε-closure(q1) and ε-closure(q2) contain a final state q2.

---

5. Write Rules for computing nullable, firstpos, and lastpos.

Ans:

Node n	**nullable(n)**	**firstpos(n)**	**lastpos(n)**
n is a leaf node labeled €	true	∅	∅
n is a leaf node labelled with position i	false	{ i }	{ i }

Node n	nullable(n)	firstpos(n)	lastpos(n)
n is an or node with left child c1 and right child c2	nullable(c1) or nullable(c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
n is a cat node with left child c1 and right child c2	nullable(c1) and nullable(c2)	If nullable(c1) then firstpos(c1) ∪ firstpos(c2) else firstpos(c1)	If nullable(c2) then lastpos(c2) ∪ lastpos(c1) else lastpos(c2)
n is a star node with child node c1	true	firstpos(c1)	lastpos(c1)

UNIT-2

Short Answer Questions:

1. Define Context-Free Grammar (CFG).

Ans:

Context-Free Grammar (CFG):

Context-Free Grammar (CFG) plays an important role in describing the syntax of programming languages.

A context-free grammar is a set of recursive rules used to generate patterns of strings.

A context-free grammar can be described by a four-element tuple (*V*, Σ, *R*, *S*), where

- *V* is a finite set of variables (which are non-terminal);
- Σ or T is a finite set (disjoint from *V*) of terminal symbols;
- *R* is a set of production rules where each production rule maps a variable to a string.
- *S* is a start symbol.

---

2. What is Left Recursion (LR) and how it is eliminated?

Ans:

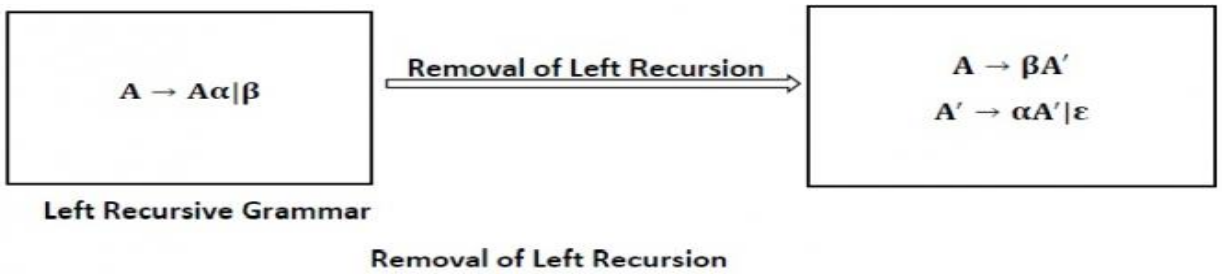
A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

$A \rightarrow A\ \alpha\ |\beta$.

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$A \rightarrow \beta A'$

$A \rightarrow \alpha A' | \epsilon$



---

3. Define Handle pruning.

Ans:

Handle pruning is the technique used to optimise the parsing process of a grammar. In order to replace nonterminal symbols to handle, parsing algorithms apply reduction.

Example:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

---

4. What are the Steps for constructing the LR parsing table.

Ans:

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.
4. Stack implementation
5. Parse Tree

---

5. What does the terms LL and LR stands for?

Ans:

LL : First L of LL is for left to right and second L is for leftmost derivation.

LR: L of LR is for left to right and R is for rightmost derivation.

Long Answer Questions:

1. Explain in brief about leftmost, rightmost and mixed derivations.

Ans:

There are three types of Derivation trees follows:

- Leftmost Derivation tree
- Rightmost derivation tree
- Mixed derivation tree

Leftmost derivation: A leftmost derivation is obtained by applying production to the leftmost variable in each successive step.

Example:

Consider the grammar G with production:

S → aSS (Rule: 1)

S → b (Rule: 2)

Compute the string w = ‘aababbb’ with left most derivation.

Sol:

S ⇒ aSS (Rule: 1)

⇒ aaSSS (Rule: 1)

⇒ aabSS (Rule: 2)

⇒ aabaSSS (Rule: 1)

⇒ aababSS (Rule: 2)

⇒ aababbS (Rule: 2)

⇒ aababbb (Rule: 2)

To obtain the string ‘w’ with “leftmost derivation”, follows “1121222” sequence rules.

Rightmost derivation: A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example:

Consider the grammar G with production:

S → aSS (Rule: 1)

S → b (Rule: 2)

Compute the string w = ‘aababbb’ with right most derivation.

Sol:

S ⇒ aSS (Rule: 1)

⇒ aSb (Rule: 2)

⇒ aaSSb (Rule: 1)

⇒ aaSaSSb (Rule: 1)

⇒ aaSaSbb (Rule: 2)

⇒ aaSabbb (Rule: 2)

⇒ aababbb (Rule: 2)

To obtain the string ‘w’ with “Rightmost derivation”, follows “1211222” sequence rules.

Mixed Derivation: In a mixed derivation the string is obtained by applying production to the leftmost variable and rightmost variable simultaneously as per the requirement in each successive step.

S ⇒ aSS (Rule: 1)

⇒ aSb (Rule: 2)

⇒ aaSSb (Rule: 1)

⇒ aaSaSSb (Rule: 1)

⇒ aabaSSb (Rule: 2)

⇒ aabaSbb (Rule: 2)

⇒ aababbb (Rule: 2)

To obtain the string ‘w’ with “mixed derivation”, follows “1211222” sequence rules.

---

2. What is Backtracking? Show the Backtracking in the following given grammar
S -> r P d
P -> m | m n
Input string: 'rmnd.'

Ans:

Backtracking parsers:

- Backtracking parsers are a type of top-down parser that can handle non-deterministic grammar.
- When a parsing decision leads to a dead end, the parser can backtrack and try another alternative.
- Backtracking parsers are not as efficient as other top-down parsers because they can potentially explore many parsing paths.

Example:

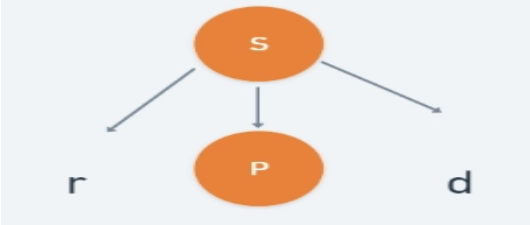
The following is a grammar rule:

S -> r P d

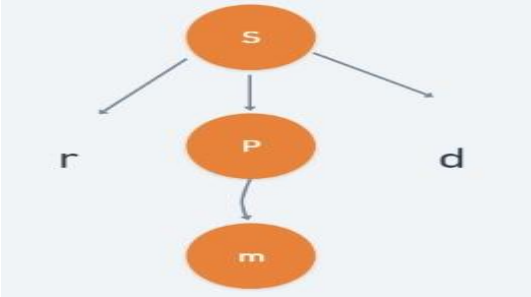
P -> m | m n

Input string: 'rmnd.'

Building the parse tree will start with the symbol S.

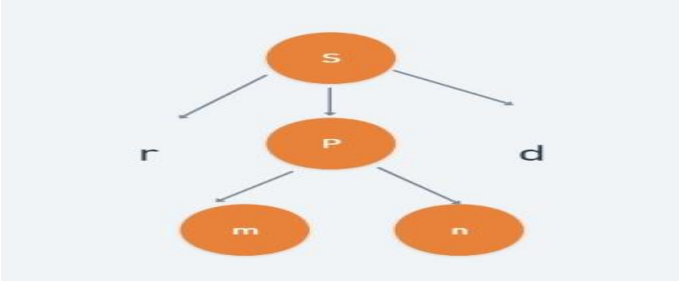


The next leaf of the parse tree 'P' has two production rules. Let's expand the symbol 'P' with the given first production of 'P,' i.e., 'm' we get the following parse tree.



We get a mismatch when comparing the third input symbol 'n' against the next leaf labeled 'd'. Therefore, the parser must return to the symbol 'P' and look for the other alternative.

The other alternative for the production of 'P' is 'mn .'Also, the input pointer must be back-tracked and reset to the position where it was first before we expanded the symbol 'P .'It means the input pointer must point to 'm' again.



Now we reached Input string: 'rmnd.' Using backtracking method.

---

3. List out the rules for FIRST and FOLLOW? Construct FIRST and FOLLOW for the grammar

E -> TE'

E' -> +T E'|Є

T -> F T'

T' -> *F T' | Є

F -> (E) | id

Ans:

FIRST() and FOLLOW() calculation on each non-terminal:

First(): The first terminal symbol is referred to as the First(). if there is a variable, and we attempt to derive all the strings from that variable.

Rule 1: For a production rule X → Є ,

First(X) = { Є }

Rule 2: For a production rule X →ab,

First(X) = { a }

Rule 3: For a production rule X →BC,

if First(B) does not contain Є First(X) = First(B)

if First(B) contains Є then First(X)=First(B) U First(C)

Example:

Production Rules:

E -> TE'

E' -> +T E'|Є

T -> F T'

T' -> *F T' | Є

F -> (E) | id

FIRST set

FIRST(E) = FIRST(T) = { (, id }

FIRST(E') = { +, Є }

FIRST(T) = FIRST(F) = { (, id }

FIRST(T') = { *, Є }

FIRST(F) = { (, id }

Follow(): The terminal symbol that follows a variable throughout the derivation process.

Rule 1: For the start symbol S,

Follow(S) = {$}

Rule 2: For any production rule A → αB

Follow(B) = Follow(A)

Rule 3: For any production rule A → αBβ,

if First(B) does not contain Є Follow(B) = First(β)

if First(B) contains Є then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Example:

Production Rules:

E -> TE'

E' -> +T E'|Є

T -> F T'

T’ -> *F T’ | Є

F -> (E) | id

FOLLOW Set

FOLLOW(E) = { $,) }

FOLLOW(E’) = FOLLOW(E) = { $,) }

FOLLOW(T) = { FIRST(E’) – Є } U FOLLOW(E) = { + , $,) }

FOLLOW(T’) = FOLLOW(T) = { + , $,) }

FOLLOW(F) = { FIRST(T’) – Є } U FOLLOW(T) = { *, +, $,) }

---

4. What is Shift-reduce parsing? Consider the grammar

S –> S + S

S –> S * S

S –> id

Perform Shift Reduce parsing for input string “id - id * id”.

Ans:

Bottom-up Parser (Shift Reduce Parser):

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

Shift: This involves moving symbols from the input buffer onto the stack.

Reduce: If the handle appears on top of the stack, then, its reduction by using appropriate production rule is done.

Example: Consider the grammar

S –> S + S

S –> S * S

S –> id

Perform Shift Reduce parsing for input string “id + id + id”.

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce S->id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce S->id
$S+S	+id$	Reduce S->S+S
$S	+id$	Shift
$S+	id$	Shift
$S+id	$	Reduce S->id
$S+S	$	Reduce S->S+S
$S	$	Accept

---

5. Find the LR(0) set of items for the following grammar and Describe state diagram and construct parse table of that

S→AA

A→aA|b

Ans:

1. LR(0) Parser:

- L stands for the left to right scanning
- R stands for rightmost derivation in reversew
- 0 stands for no. of input symbols of lookahead.

Steps for constructing the LR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.
4. Stack implementation
5. Parse Tree

1.Augmented grammar :

If G is a grammar with starting symbol S, then G' (augmented grammar for G) is a grammar with a new starting symbol S' and productions S'→ .S

Example:

S–>AA

A–>aA | b

Augmented grammar is:-

S'–>.S [0th production]

S–>.AA [1st production]

A–>.aA [2nd production]

A–>.b [3rd production]

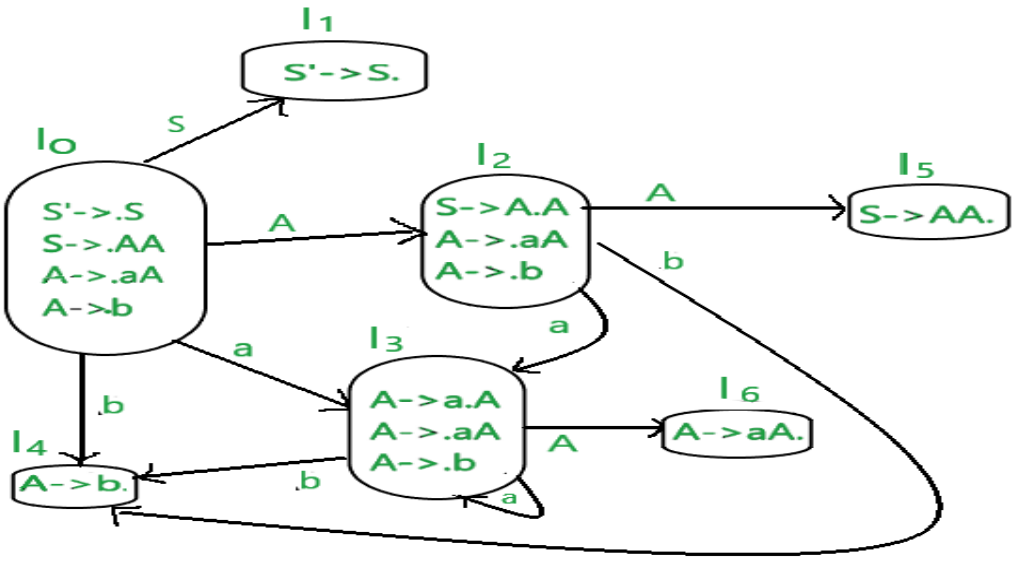
2. LR(0) collection of items to be found

The terminals of this grammar are {a,b}

The non-terminals of this grammar are {S,A}

RULE 1 – if any nonterminal has ' . ' preceding it, we have to write all its production and add ' . ' preceding each of its-production.

RULE 2 – from each state to the next state, the ' . ' shifts to one place to the right.



3. Defining 2 functions: goto(list of non-terminals) and action(list of terminals) in the parsing table.

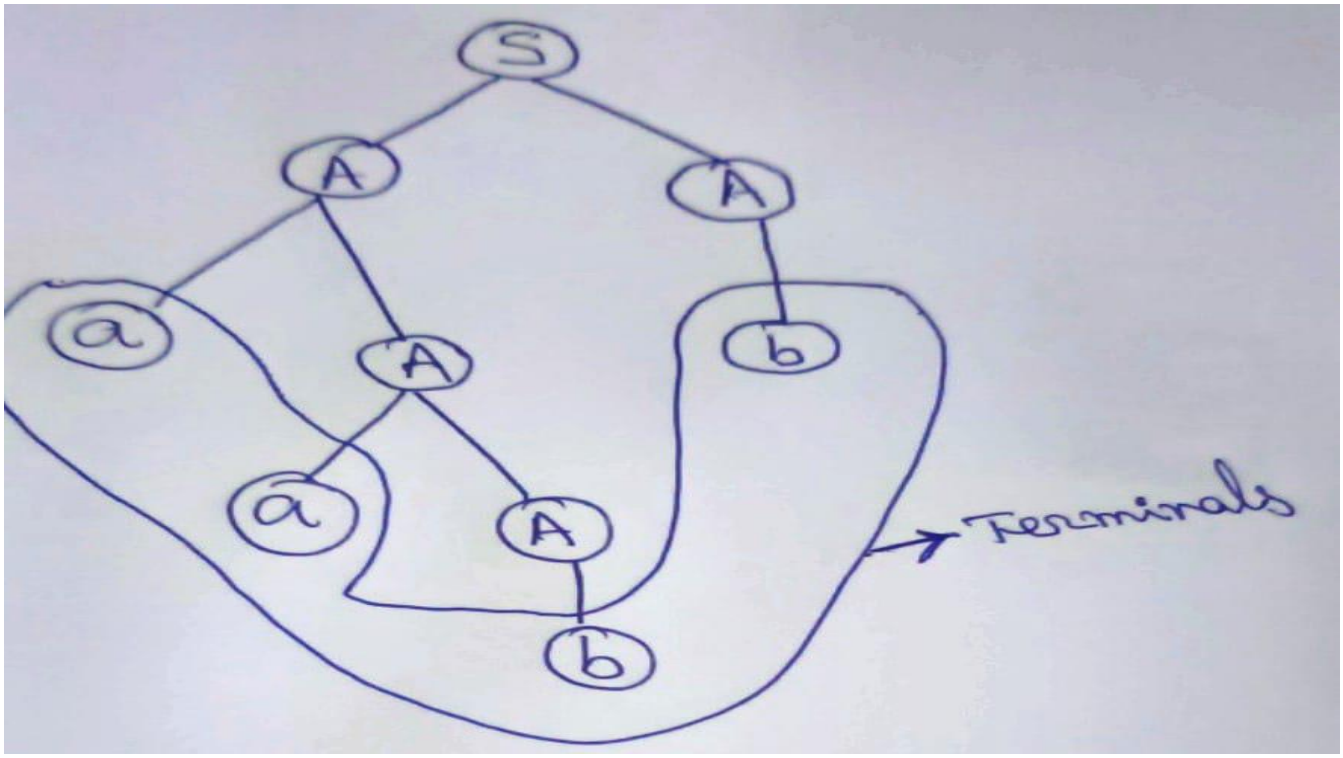
	ACTION			GOTO	
	a	b	$	A	S

0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5	R1	R1	R1		
6	R2	R2	R2		

4.Stack implementation for string aabb

Stack	Input	Action
$0	aabb$	Shift a into Stack & goto state 3
$0a3	abb$	Shift a into Stack & goto state 3
$0a3a3	bb$	Shift b into Stack & goto state 4
$0a3a3b4	b$	Reduce A→b
$0a3a3A6	b$	Reduce A→aA
$0a3A6	b$	Reduce A→aA
$0A2	b$	Shift b into Stack & goto state 4
$0A2b4	$	Reduce A→b
$0A2A5	$	Reduce S→AA
$0S1	$	Accept

Step 5: Parse Tree



UNIT-3

Short Answer Questions:

1. Explain the concept of Syntax Directed Definitions (SDD)?

Ans:

Syntax-Direct Definition (SDD):

- Syntax Directed Definition (SDD) is a kind of abstract specification.
- A CFG with attributes and rules is called a **syntax-directed definition (SDD).**
- A grammar symbol's attribute can now be integers, types, table references, or a string.

Example:

E → E1 + T { E.val = E1.val + T.val}

---

2. Define Annotated Parse Tree?

Ans:

Annotated Parse Tree: The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

---

3. What is S and L attributes?

Ans:

Implementing L Attributed SDD:

S-attributed SDT:

- The grammar that contains all the syntactic rules along with the semantic rules having synthesized attributes only, is called S-attributed.
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

L-attributed SDT:

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

Example: S->ABC, here attribute B can only obtain its value either from the parent – S or its left sibling A but It can't inherit from its right sibling C. Same goes for A & C – A can only get its value from its parent & C can get its value from S, A, & B as well because C is the rightmost attribute in the given production.

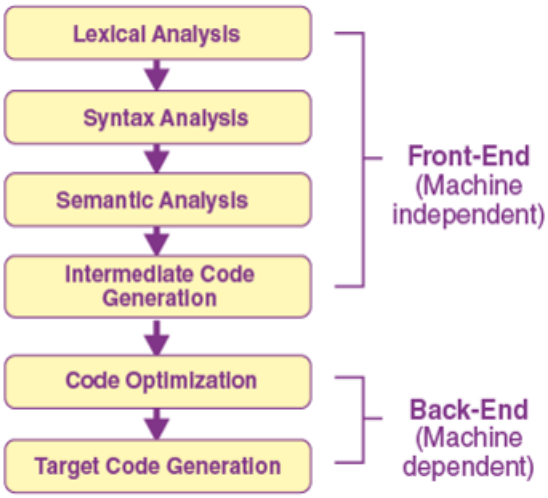
---

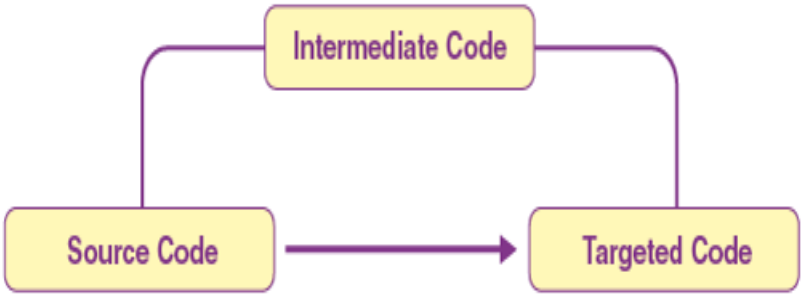
4. What is Intermediate code Generator?

Ans:

Intermediate Code Generation:

- The parse tree is semantically confirmed; now, an intermediate code generator develops three address codes.
- A code that is neither high-level nor machine code, but a middle-level code is an intermediate code.
- We can translate this code to machine code later.
- This stage serves as a bridge or way from analysis to synthesis.





---

5. What are The Benefits of Static Type Checking?

Ans:

1. Runtime Error Protection.

2. It catches syntactic errors like spurious words or extra punctuation.
3. It catches wrong names like Math and Predefined Naming.
4. Detects incorrect argument types.
5. It catches the wrong number of arguments.
6. It catches wrong return types, like return “70”, from a function that’s declared to return an int.

Long Answer Questions:

1. Design the dependency graph for the following grammar

S --> T List

T --> int

T --> float

T --> double

List --> List1, id

List --> id

Ans. : The dotted line is for representing the parse tree.

The semantic rules for the above grammar is as given below.

Production rule	Semantic actions
S → T List	List.in:=T.type
T → int	T.type:=integer
T → float	T.type:=float
T → char	T.type:=char
T → double	T.type:=double
List → $List_1$, id	$List_1$.in:=List.in Enter_type(id.entry, List.in)
List → id	Enter_type(id.entry, List.in)

The dependency graph is as shown in Fig. Q.7.1.

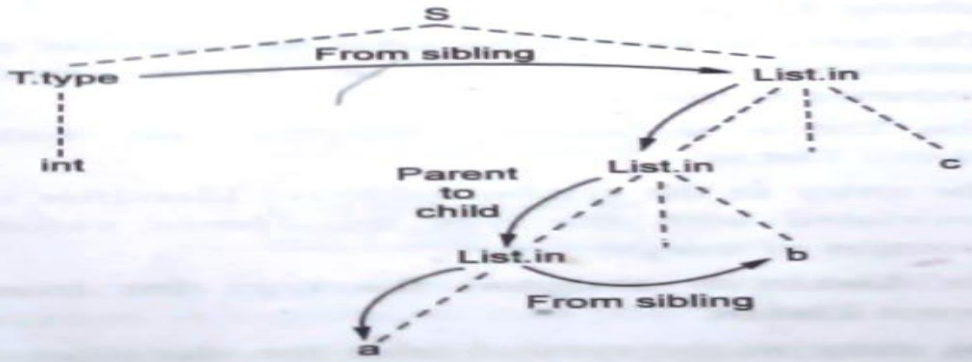


Fig. Q.7.1 Dependency graph

---

2. Construct Syntax Trees for x*y-5+z used for SDT

Ans:

Step1: Convert the expression from infix to postfix i.e xy*5-z+

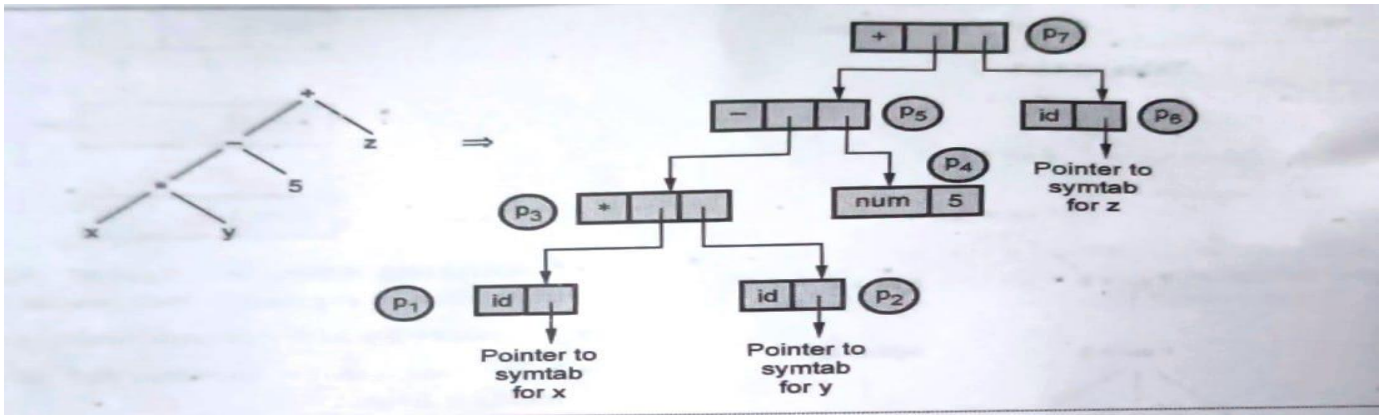
Step2: Make use of the functions mknode(), mkleaf(id,ptr) and mkleaf(num,val)

Step3: The sequence of function calls.

Symbol	Operation
x	P1=mkleaf(id.ptr to entry x)
y	P2=mkleaf(id.ptr to entry y)
*	P3=mknode(*,p1,p2)
5	P4=mkleaf(num,5)

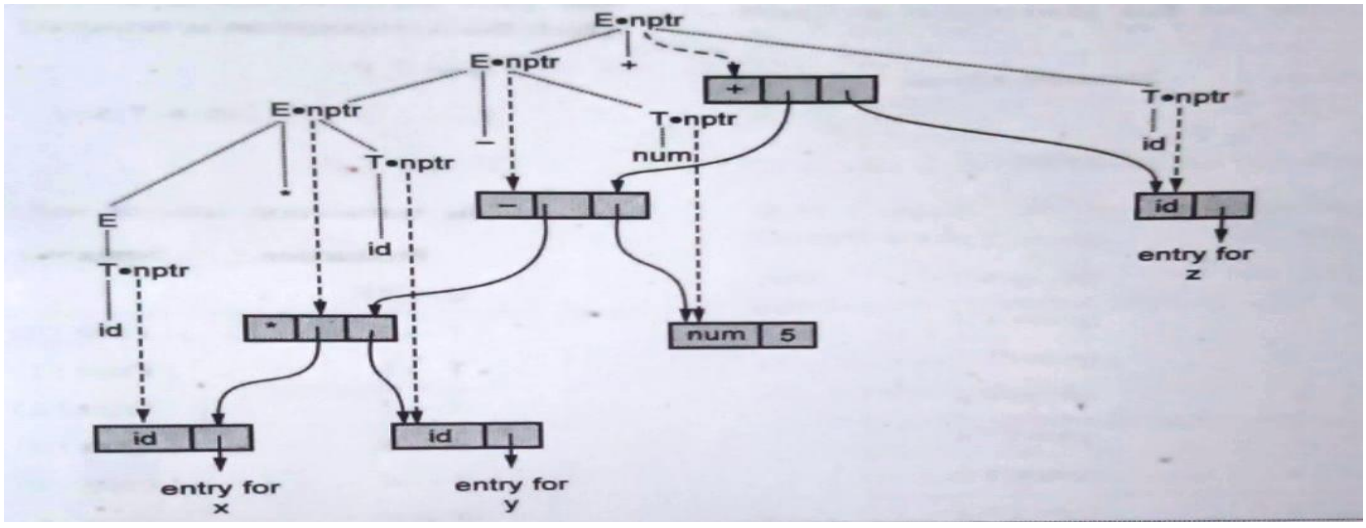
-	P5=mknode(-,p3,p4)
z	P6=mkleaf(id.ptr to entry z)
+	P7=mknode(+,p5,p6)

Syntax tree:



The syntax-directed definition for the above grammar is as given below.

Production rule	Semantic operation
$E \rightarrow E_1 + T$	E.nptr:=mknode('+',E_1.nptr,T.nptr)
$E \rightarrow E_1 - T$	E.nptr:=mknode('–',E_1.nptr,T.nptr)
$E \rightarrow E_1 * T$	E.nptr:=mknode('*',E_1.nptr,T.nptr)
$E \rightarrow T$	E.nptr:=T.nptr
$T \rightarrow id$	E.nptr:=mkleaf(id,id.ptr_entry)
$T \rightarrow num$	T.nptr:=mkleaf(num,num.val)



---

3. Construct DAG for the given expression a + a * (b – c) + (b – c) * d.

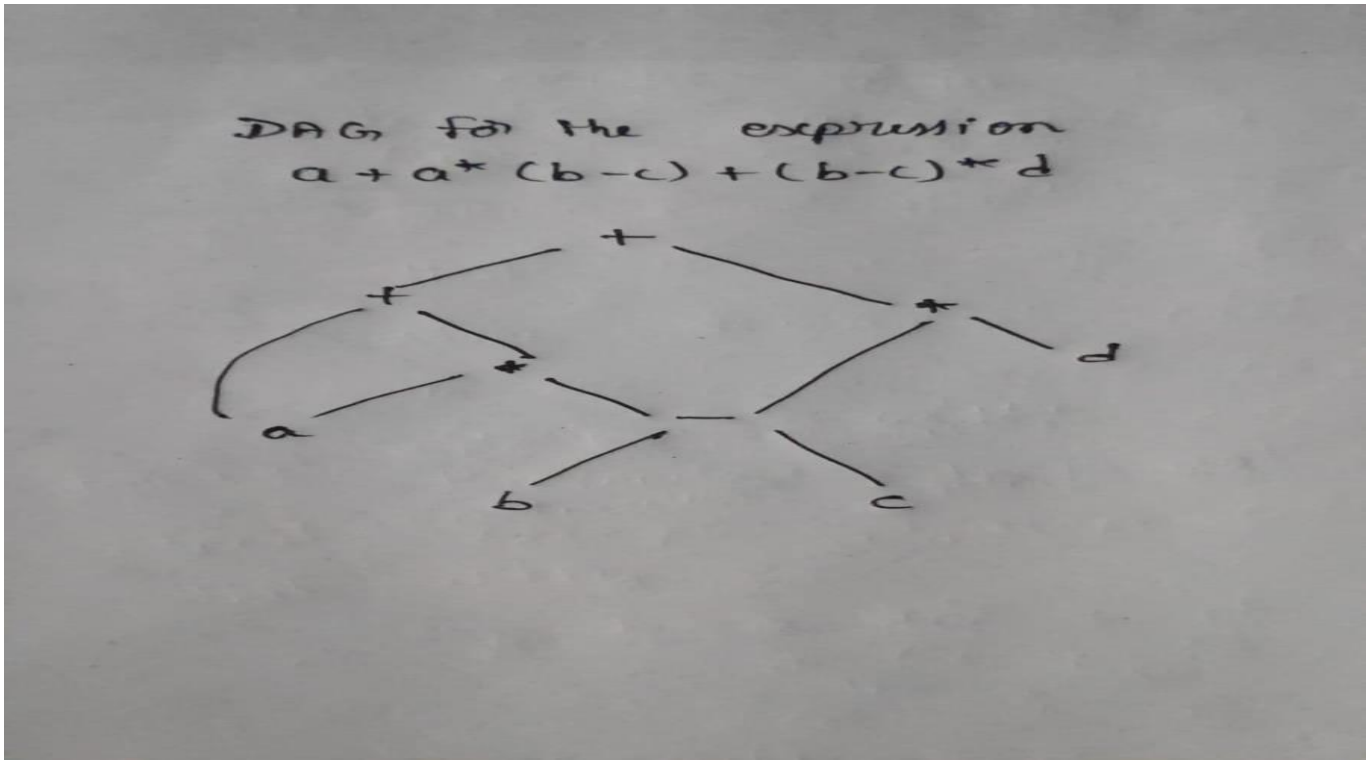
Ans:

STEPS FOR CONSTRUCTING A DAG:

1. d1=leaf(id,entry-a)
2. d2=leaf(id,entry-a)=d1
3. d3=leaf(id,entry-b)
4. d4=leaf(id,entry-c)
5. d5=node('-',d3,d4)
6. d6=node('*',d1,d5)
7. d7=node('+',d1,d6)
8. d8=leaf(id,entry-b)=d3
9. d9=leaf(id,entry-c)=d4
10. d10=node('-',d3,d4)=d5
11. d11=leaf(id,entry-d)
12. d12=node('*',d5,d11)
13. d13=node('+',d7,d12)

OR

Infix notation: + + a * a – b c * – b c d



---

4. Explain Three address code, Implement Three Address Code for the given expression a = (b * – c) + (b * – c).

Ans:

Implementation of Three Address Code

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –

- It is a structure which consists of 4 fields namely op, arg1, arg2 and result.
- op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Example – Consider expression a = (b * – c) + (b * – c). The three-address code is:

t1 = uminus c

t2 = b * t1

t3 = uminus c

t4 = b * t3

t5 = t2 + t4

a = t5

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples –

- It is a structure which consists of 3 fields namely op, arg1, arg2.

Example – Consider expression a =(b * – c) + (b * – c)

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –

- This representation makes use of pointer to the listing of all references to computations which is made separately and stored.
- Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = (b * – c) + (b * – c)

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

5.Consider the following grammar

S --> T L

T --> int

T --> float

T --> double

L --> L1, id

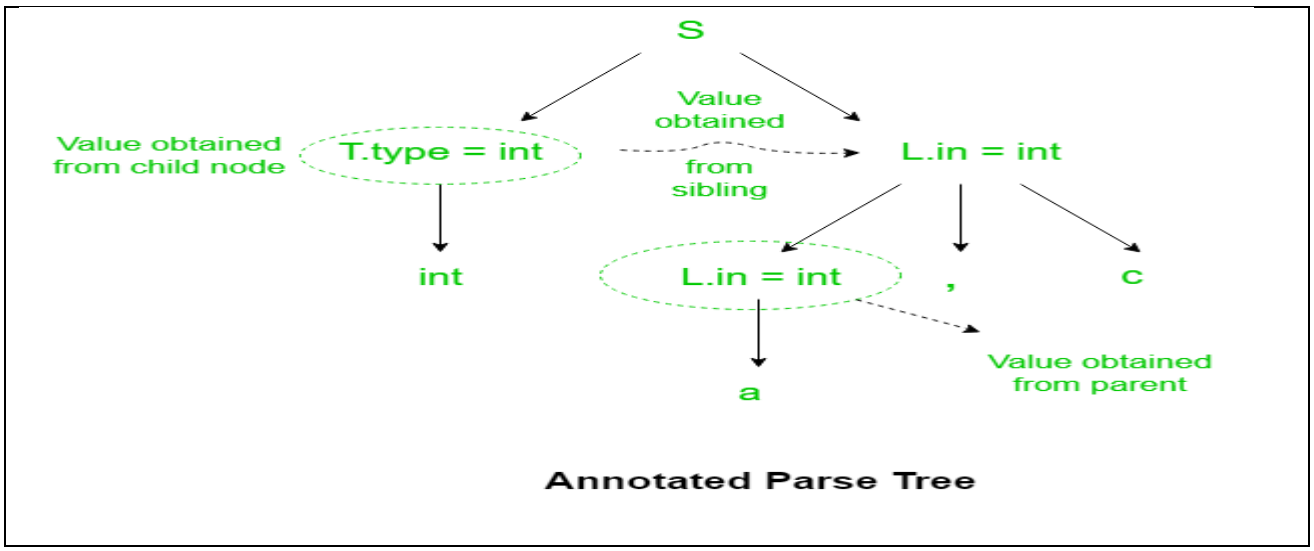
L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string int a, c for computing inherited attributes. The annotated parse tree for the input string.

Ans:



Annotated Parse Tree

UNIT-4

Short Answer Questions:

1. What are the Addresses in the Target Code?

Ans:

The address in the target code is

1. Static allocation
2. Stack allocation

- In static allocation, the position of an activation record is fixed in memory at compile time.
- In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.

---

2. Define Basic Block?

Ans:

Basic block contains a sequence of statement. The flow of control enters at the beginning of the statement and leave at the end without any halt (except may be the last instruction of the block). The following sequence of three address statements forms a basic block:

1. t1:= x * x
2. t2:= x * y
3. t3:= 2 * t2
4. t4:= t1 + t3
5. t5:= y * y
6. t6:= t4 + t5

---

3. What is Dead Code Elimination?

Ans:

The operation on DAG's that corresponds to dead-code elimination can be im-plemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

---

4. What is Target program?

Ans:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

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

---

5. What is Copy Propagation ?

Ans:

Copy propagation is defined as an optimization technique used in compiler design. Copy propagation is used to replace the occurrence of target variables that are the direct assignments with their values. Copy propagation is related to the approach of a common subexpression. In common subexpression, the expression values are not changed since the first expression is computed.

Long Answer Questions:

1. Explain Code Generation process.

Ans:

- The final phase in compiler model is the code generator.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.
- The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

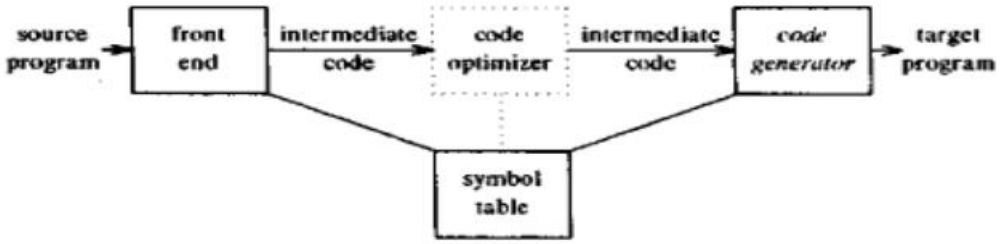


Figure Position of code generator

- The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.
- The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:
  – It should carry the exact meaning of the source code.
  – It should be efficient in terms of CPU usage and memory management.
- Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement x:= y + z. It can have the following sequence of codes:

MOV x, R

ADD y, R_0

---

2. Explain Design Issues or Issues in the Design of code Generator.

Ans:

In the code generation phase, various issues can arises:

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:

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
    a) Postfix notation
    b) Syntax tree
    c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

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

3. Memory management:

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection:

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

1. a:= b + c
2. d:= a + e

Inefficient assembly code is:

1. MOV b, R0 R0→b
2. ADD c, R0 R0 →c + R0
3. MOV R0, a a → R0
4. MOV a, R0 R0 → a
5. ADD e, R0 R0→ e + R0
6. MOV R0, d d → R0

5. Register allocation:

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

Register allocation: In register allocation, we select the set of variables that will reside in register.

Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

Example:

Consider the following division instruction of the form:

1. D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.

Old register is used to hold the quotient.

6. Evaluation order:

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

---

3. Explain Basic Blocks and Flow Graphs?

Ans:

Basic Block

Basic block contains a sequence of statement. The flow of control enters at the beginning of the statement and leave at the end without any halt (except may be the last instruction of the block). The following sequence of three address statements forms a basic block:

7. t1:= x * x
8. t2:= x * y
9. t3:= 2 * t2
10. t4:= t1 + t3
11. t5:= y * y
12. t6:= t4 + t5

Basic block construction:

Algorithm: Partition into basic blocks

Input: It contains the sequence of three address statements

Output: it contains a list of basic blocks with each three address statement in exactly one block

Method: First identify the leader in the code. The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

For each leader, its basic block consists of the leader and all statement up to. It doesn't include the next leader or end of the program.

Consider the following source code for dot product of two vectors a and b of length 10:

1. begin
2. prod :=0;
3. i:=1;
4. do begin

5. prod :=prod+ a[i] * b[i];
6. i :=i+1;
7. end
8. while i <= 10
9. end

The three address code for the above source program is given below:

B1

1. (1) prod := 0
2. (2) i := 1

B2

1. (3) t1 := 4* i
2. (4) t2 := a[t1]
3. (5) t3 := 4* i
4. (6) t4 := b[t3]
5. (7) t5 := t2*t4
6. (8) t6 := prod+t5
7. (9) prod := t6
8. (10) t7 := i+1
9. (11) i := t7
10. (12) if i<=10 goto (3)

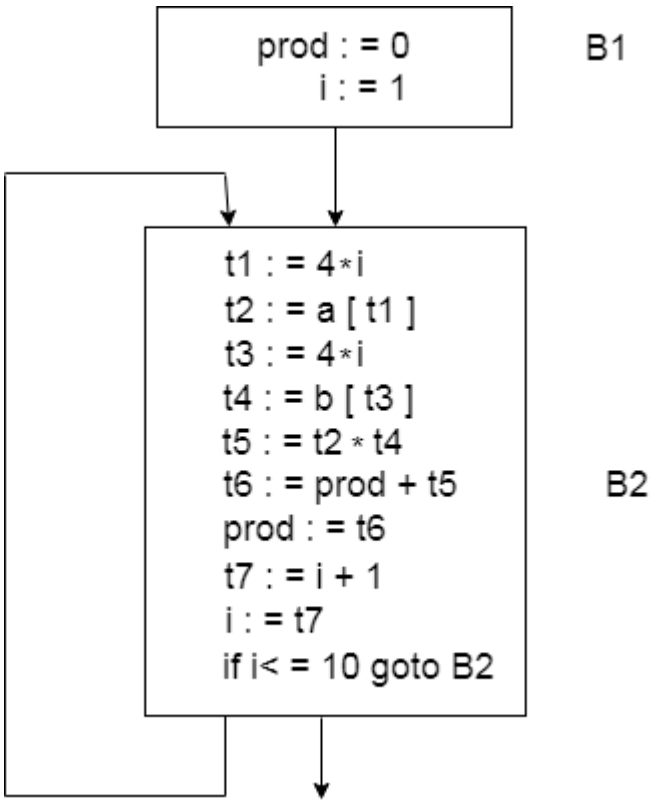
Basic block B1 contains the statement (1) to (2)

Basic block B2 contains the statement (3) to (12)

Flow Graph:

- Flow graph is a directed graph.
- It contains the flow of control information for the set of basic block.
- A control flow graph is used to depict that how the program control is being parsed among the blocks. It is useful in the loop optimization.

Flow graph for the vector dot product is given as follows:



- Block B1 is the initial node. Block B2 immediately follows B1, so from B2 to B1 there is an edge.
- The target of jump from last statement of B1 is the first statement B2, so from B1 to B2 there is an edge.
- B2 is a successor of B1 and B1 is the predecessor of B2.

---

4. Discuss and analyze all the allocation strategies in a run-time storage environment.

Ans:

Storage Allocation Strategies in Compiler Design

A compiler is a program that converts HLL(High-Level Language) to LLL(Low-Level Language) like machine language. In a compiler, there is a need for storage allocation strategies in Compiler design because it is very important to use the right strategy for storage allocation as it can directly affect the performance of the software.

Storage Allocation Strategies

There are mainly three types of Storage Allocation Strategies:

1. Static Allocation
2. Heap Allocation
3. Stack Allocation

1. Static Allocation

Static allocation lays out or assigns the storage for all the data objects at the compile time. In static allocation names are bound to storage. The address of these identifiers will be the same throughout. The memory will be allocated in a static location once it is created at compile time. C and C++ use static allocation.

Eample:

int number = 1;

static int digit = 1;

Advantages of Static Allocation

1. It is easy to understand.
2. The memory is allocated once only at compile time and remains the same throughout the program completion.
3. Memory allocation is done before the program starts taking memory only on compile time.

Disadvantages of Static Allocation

1. Not highly scalable.
2. Static storage allocation is not very efficient.
3. The size of the data must be known at the compile time.

2. Heap Allocation

Heap allocation is used where the Stack allocation lacks if we want to retain the values of the local variable after the activation record ends, which we cannot do in stack allocation, here LIFO scheme does not work for the allocation and de-allocation of the activation record. Heap is the most flexible storage allocation strategy we can dynamically allocate and de-allocate local variables whenever the user wants according to the user needs at run-time. The variables in heap allocation can be changed according to the user's requirement. C, C++, Python, and Java all of these support Heap Allocation.

Example:

int* ans = new int[5];

Advantages of Heap Allocation

1. Heap allocation is useful when we have data whose size is not fixed and can change during the run time.
2. We can retain the values of variables even if the activation records end.
3. Heap allocation is the most flexible allocation scheme.

Disadvantages of Heap Allocation

1. Heap allocation is slower as compared to stack allocation.
2. There is a chance of memory leaks.

3. Stack Allocation

Stack is commonly known as Dynamic allocation. Dynamic allocation means the allocation of memory at run-time. Stack is a data structure that follows the LIFO principle so whenever there is multiple activation record created it will be pushed or popped in the stack as activations begin and ends. Local variables are bound to new storage each time whenever the activation record begins because the storage is allocated at runtime every time a procedure or function call is made. When the activation record gets popped out, the local variable values get erased because the storage allocated for the activation record is removed. C and C++ both have support for Stack allocation.

Example:

```
void sum(int a, int b){int ans = a+b;cout<<ans;}
// when we call the sum function in the example above,
memory will be allotted for the variable ans
```

---

5. Explain various method to handle peephole optimization.

Ans:

Peephole Optimization in Compiler Design

Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.

It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

Objectives of Peephole Optimization:

The objective of peephole optimization is as follows:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Peephole Optimization Techniques

A. Redundant load and store elimination: In this technique, redundancy is eliminated.

Initial code:

```
y = x + 5;
i = y;
z = i;
w = z * 3;
```

Optimized code:

```
y = x + 5;
w = y * 3; //* there is no i now
```

//* We've removed two redundant variables i & z whose value were just being copied from one another.

B. Constant folding: The code that can be simplified by the user itself, is simplified. Here simplification to be done at runtime are replaced with simplified code to avoid additional computation.

Initial code:

```
x = 2 * 3;
```

Optimized code:

```
x = 6;
```

C. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

y = x * 2;

Optimized code:

y = x + x; or y = x << 1;

Initial code:

y = x / 2;

Optimized code:

y = x >> 1;

D. Null sequences/ Simplify Algebraic Expressions : Useless operations are deleted.

a := a + 0;
a := a * 1;
a := a/1;
a := a - 0;

E. Combine operations: Several operations are replaced by a single equivalent operation.

F. Deadcode Elimination:- Dead code refers to portions of the program that are never executed or do not affect the program's observable behavior. Eliminating dead code helps improve the efficiency and performance of the compiled program by reducing unnecessary computations and memory usage.

Initial Code:-

```
int Dead(void)
{
   int a=10;
   int z=50;
   int c;
   c=z*5;
   printf(c);
```

```
    a=20;
    a=a*10; //No need of These Two Lines
    return 0;
    }
```

Optimized Code:-

```
int Dead(void)
{
    int a=10;
    int z=50;
    int c;
    c=z*5;
    printf(c);
    return 0;
    }
```

UNIT-5

Short Answer Questions:

1. List out the common issues in the design of code generator.

Ans:

Design Issues In the code generation phase, various issues can arises:

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

---

2. What is the Basic Block in Compiler Design?

Ans:

In compiler design, a basic block is a straight-line piece of code that has only one entry point and one exit point. Basic block construction is the process of dividing a program's control flow graph into basic blocks.

The task is to partition a sequence of three-address codes into the basic block. The new basic block always begins with the first instruction and continues to add instructions until a jump or a

label is reached. If no jumps or labels are identified, control will flow sequentially from one instruction to another.

Task: Partition a sequence of three-address codes into basic blocks.

Input: Sequence of three address statements.

Output: A sequence of basic blocks.

---

2. Write the Properties of Flow Graphs.

Ans:

Properties of Flow Graphs

The control flow graph is process-oriented.

A control flow graph shows how program control is parsed among the blocks.

The control flow graph depicts all of the paths that can be traversed during the execution of a program.

It can be used in software optimization to find unwanted loops.

---

4. Define loop Unrolling.Give an Example

Ans:

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Example:

```
// This program does not uses loop unrolling.
#include<stdio.h>
int main(void)
 {
    for (int i=0; i<5; i++)
    printf("Hello\n"); //print hello 5 times
    return 0;
}
```

---

5. Define the term-Dominators.

Ans:

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

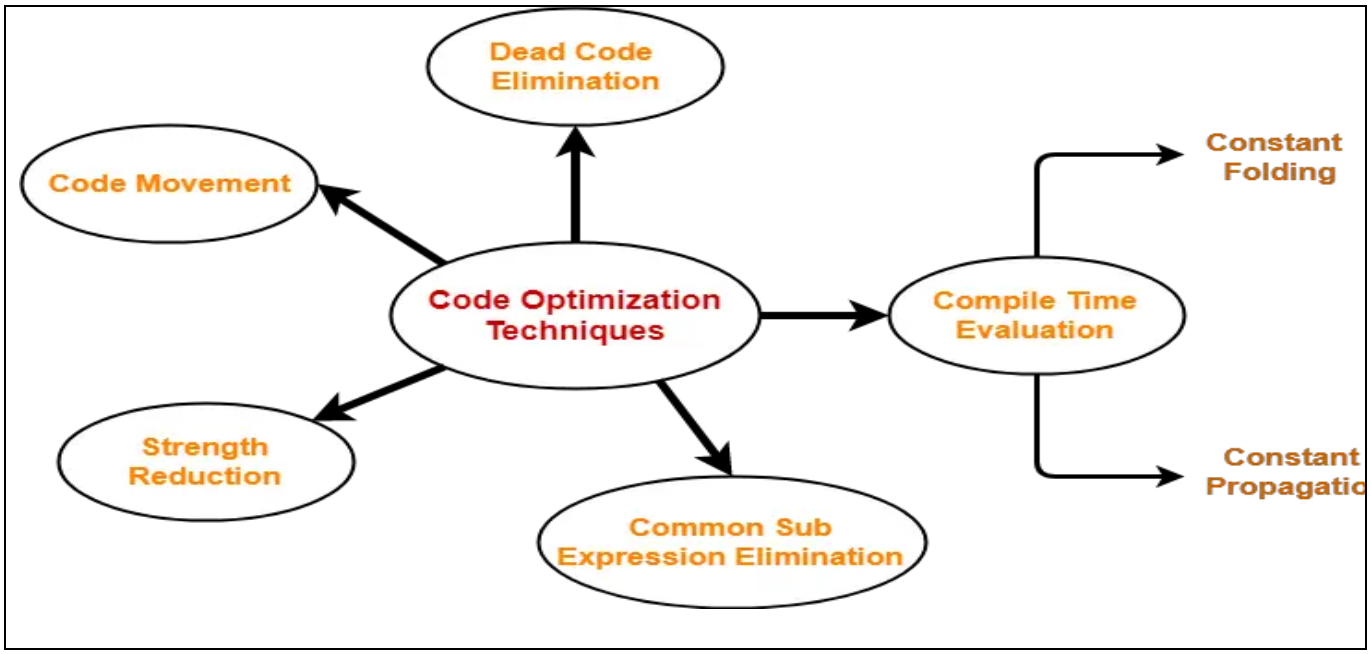
Long Answer Questions:

1. Explain principal sources of optimization in detail?

Ans:

Code Optimization or Principal Sources of Optimization:

Code Optimization or Principal Sources of Optimization is done in the following different ways:



1. Compile Time Evaluation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction

1. Compile Time Evaluation:

Two techniques that falls under compile time evaluation are-

A) Constant Folding-

In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

Example: Circumference of Circle = (22/7) x Diameter
Here,

- This technique evaluates the expression 22/7 at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

B) Constant Propagation-

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

Example: pi = 3.14
radius = 10
Area of circle = pi x radius x radius

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves the time at run time.

2. Common Sub-Expression Elimination:

The expression that has been already computed before and appears again in the code for computation is called as **Common Sub-Expression**.

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Example:

Code Before Optimization	Code After Optimization
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i **// Redundant Expression** S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

3.Code Movement-

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.

- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Example:

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

4. Dead Code Elimination-

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Example:

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

5. Strength Reduction-

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example:

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

- The expression “A x 2” is replaced with the expression “A + A”.
- This is because the cost of multiplication operator is higher than that of addition operator.

---

2. Explain how Constant Propagation can be done using data flow equation?

Ans:

Constant Propagation:

Constant Propagation is one of the local code optimization techniques in Compiler Design. It can be defined as the process of replacing the constant value of variables in the expression.

In simpler words, we can say that if some value is assigned a known constant, than we can simply replace the that value by constant. Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.

Constant propagation is executed using reaching definition analysis results in compilers, which means that if reaching definition of all variables have same assignment which assigns a same constant to the variable, then the variable has a constant value and can be substituted with the constant.

Suppose we are using pi variable and assign it value of 22/7

pi = 22/7 = 3.14

In the above code the compiler has to first perform division operation, which is an expensive operation and then assign the computed result 3.14 to the variable pi. Now if anytime we have to use this constant value of pi, then the compiler again has to look – up for the value and again perform division operation and then assign it to pi and then use it. This is not a good idea when we can directly assign the value 3.14 to pi variable, thus reducing the time needed for code to run.

Also, Constant propagation reduces the number of cases where values are directly copied from one location or variable to another, in order to simply allocate their value to another variable. For an example:

Consider the following pseudocode :

```
a = 30
b = 20 - a /2
c = b * ( 30 / a + 2 ) -  a
```

We can see that in the first expression value of a have assigned a constant value that is 30. Now, when the compiler comes to execute the second expression it encounters a, so it goes up to the first expression to look for the value of a and then assign the value of 30 to a again, and then it executes the second expression. Now it comes to the third expression and encounters b and a again, and then it needs to evaluate the first and second expression again in order to compute the value of c. Thus, a needs to be propagated 3 times This procedure is very time consuming.

We can instead, rewrite the same code as:

a = 30
b = 20 - 30/2
c = b * (30 / 30 + 2) – 30

This updated code is faster as compared to the previous code as the compiler does not need to again and again go back to the previous expressions looking up and copying the value of a variable in order to compute the current expressions. This saves a lot of time and thus, reducing time complexity and perform operations more efficiently.

---

3. Explain the Data Flow Analysis Properties

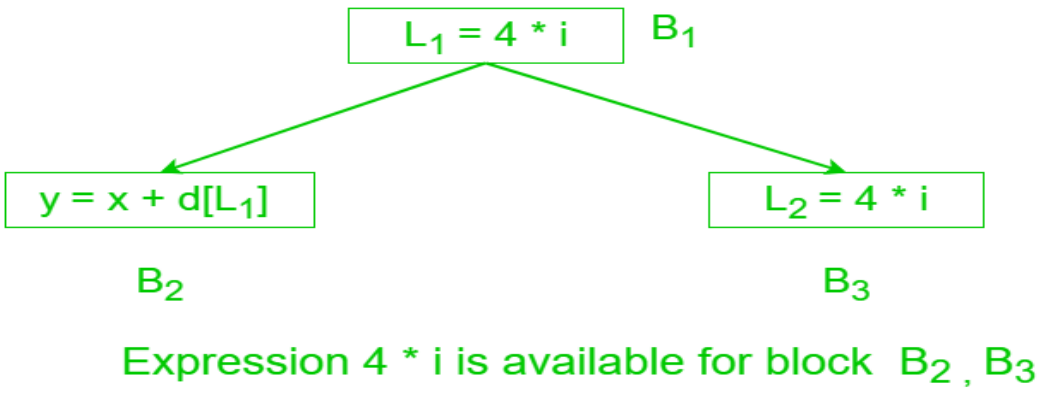
Ans:

Data Flow Analysis Properties:

1.Available Expression: A expression is said to be available at a program point x if along paths its reaching to x. A Expression is available at its evaluation point.

An expression a+b is said to be available if none of the operands gets modified before their use.

Example:

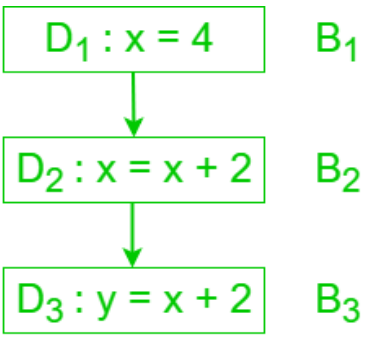


Advantage:

It is used to eliminate common sub expressions.

2.Reaching Definition: A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.

Example:



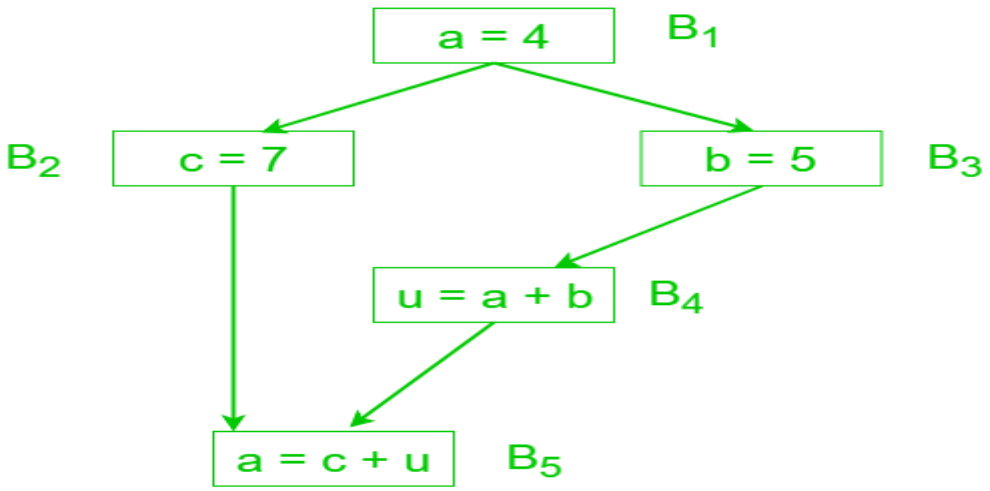
D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2

Advantage:

It is used in constant and variable propagation.

3.Live variable: A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.

Example:



a is live at block B_1 , B_3 , B_4 but killed at B_5

Advantage:

It is useful for register allocation.

It is used in dead code elimination.

4.Busy Expression: An expression is busy along a path if its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.

Advantage:

It is used for performing code movement optimization.

---

4. Explain the Loop Optimization in Compiler Design.

Ans:

Loop Optimization in Compiler Design:

- Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.
- Loop Optimization is a machine independent optimization. Whereas Peephole optimization is a machine dependent optimization technique.
- Decreasing the number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

Loop Optimization Techniques

In the compiler, we have various loop optimization techniques, which are as follows:

1. Code Motion (Frequency Reduction)

In frequency reduction, the amount of code in the loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

Example:

Before optimization:

```
while(i<100)
{
 a = Sin(x)/Cos(x) + i;
 i++;
}
```

After optimization:

```
t = Sin(x)/Cos(x);
```

```
while(i<100)
{
 a = t + i;
 i++;
}
```

2. Induction Variable Elimination

If the value of any variable in any loop gets changed every time, then such a variable is known as an induction variable. With each iteration, its value either gets incremented or decremented by some constant value.

Example:

Before optimization:
B1
i:= i+1
x:= 3*i
y:= a[x]
if y< 15, goto B2
In the above example, i and x are locked, if i is incremented by 1 then x is incremented by 3. So, i and x are induction variables.

After optimization:
B1
i:= i+1
x:= x+4
y:= a[x]
if y< 15, goto B2

3. Strength Reduction

Strength reduction deals with replacing expensive operations with cheaper ones like multiplication is costlier than addition, so multiplication can be replaced by addition in the loop.

Example:

Before optimization:

```
while (x<10)
  {
    y := 3 * x+1;
    a[y] := a[y]-2;
    x := x+2;
   }
```

After optimization:

```
t= 3 * x+1;
    while (x<10)
      {
        y=t;
        a[y]= a[y]-2;
        x=x+2;
        t=t+6;
      }
```

4. Loop Invariant Method

In the loop invariant method, the expression with computation is avoided inside the loop. That computation is performed outside the loop as computing the same expression each time was overhead to the system, and this reduces computation overhead and hence optimizes the code.

Example:

Before optimization:

```
for (int i=0; i<10;i++)
t= i+(x/y);
...
end;
```

After optimization:

```
s = x/y;
for (int i=0; i<10;i++)
t= i+ s;
...
end;
```

5. Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Example:

Before optimization:

```
for (int i=0; i<5; i++)
printf("Pankaj\n");
```

After optimization:

```
printf("Pankaj\n");
```

```
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
printf("Pankaj\n");
```

6. Loop Jamming

Loop jamming is combining two or more loops in a single loop. It reduces the time taken to compile the many loops.

Example:

Before optimization:

```
for(int i=0; i<5; i++)
    a = i + 5;
for(int i=0; i<5; i++)
    b = i + 10;
```

After optimization:

```
for(int i=0; i<5; i++)
{
 a = i + 5;
 b = i + 10;
}
```

7. Loop Fission

Loop fission improves the locality of reference, in loop fission a single loop is divided into multiple loops over the same index range, and each divided loop contains a particular part of the original loop.

Example:

Before optimization:

```
for(x=0;x<10;x++)
{
  a[x]=…
  b[x]=…
}
```

After optimization:

```
for(x=0;x<10;x++)
   a[x]=…
for(x=0;x<10;x++)
   b[x]=…
```

8. Loop Interchange

In loop interchange, inner loops are exchanged with outer loops. This optimization technique also improves the locality of reference.

Example:

Before optimization:
for(x=0;x<10;x++)
for(y=0;y<10;y++)
a[y][x]=…

After optimization:
for(y=0;y<10;y++)
for(x=0;x<10;x++)
a[y][x]=…

9. Loop Reversal

Loop reversal reverses the order of values that are assigned to index variables. This help in removing dependencies.

Example:

Before optimization:
for(x=0;x<10;x++)
a[9-x]=…

After optimization:
for(x=9;x>=0;x--)
a[x]=…

10. Loop Splitting

Loop Splitting simplifies a loop by dividing it into numerous loops, and all the loops have some bodies but they will iterate over different index ranges. Loop splitting helps in reducing dependencies and hence making code more optimized.

Example:

Before optimization:
for(x=0;x<10;x++)
if(x<5)
a[x]=…
else

b[x]=…

After optimization:
for(x=0;x<5;x++)
a[x]=…
for(;x<10;x++)
b[x]=…

11. Loop Peeling

Loop peeling is a special case of loop splitting, in which a loop with problematic iteration is resolved separately before entering the loop.

Before optimization:
for(x=0;x<10;x++)
if(x==0)
a[x]=…
else
b[x]=…

After optimization:
a[0]=…
for(x=1;x<100;x++)
b[x]=…

12. Unswitching

Unswitching moves a condition out from inside the loop, this is done by duplicating loop and placing each of its versions inside each conditional clause.

Before optimization:
for(x=0;x<10;x++)
if(s>t)
a[x]=…
else
b[x]=…

After optimization:
if(s>t)
for(x=0;x<10;x++)
a[x]=…
else
for(x=0;x<10;x++)

---

5. Explain the Foundation of Data Flow Analysis in Code Optimization Techniques.

Ans:

Foundations of Data Flow Analysis:

A data-flow analysis framework (D, V, A, F) consists of

- A direction of the data flow D, which is either F O R W A R D S or B A C K W A R D S .
- A semilattice which includes a do-main of values V and a meet operator A.
- A family F of transfer functions from V to V. This family must include functions suitable for the boundary conditions, which are constant transfer functions for the special nodes E N T R Y and E X I T in any flow graph.

1 Semilattices

2 Transfer Functions

3 The Iterative Algorithm for General Frameworks

4 Meaning of a Data-Flow Solution

1. Semilattices

A *semilattice* is a set *V* and a binary meet operator A such that for all *x, y,* and *z* in *V:*

1 *x A x — x* (meet is *idempotent).*

2. *x Ay = y A x* (meet is *commutative).*

3 *x A (y A z) = (x A y) A z* (meet is *associative).*

A semilattice has a *top* element, denoted **T,** such that

for all *x* in *V,* **T** A *x — x.*

Optionally, a semilattice may have a *bottom* element, denoted ±, such that

for all *x* in V, _L A *x* = ±.

2. Transfer Functions

The family of transfer functions F : V -» V in a data-flow framework has the following properties:

1. F has an identity function i", such that I(x) = x for all x in V.
2. F is closed under composition; that is, for any two functions / and g in F, the function h defined by h(x) = g(f(x)) is in F.

3. The Iterative Algorithm for General Frameworks

Algorithm 9 . 2 5 : Iterative solution to general data-flow frameworks.

INPUT: A data-flow framework with the following components:

1. A data-flow graph, with specially labeled E N T R Y and E X I T nodes,
2. A direction of the data-flow D,
3. A set of values V,
4. A meet operator A,
5. A set of functions F, where fs in F is the transfer function for block B,and A constant value GENTRY or f EXIT in V, representing the boundary condition for forward and backward frameworks, respectively.

OUTPUT: Values in V for IN[B] and OUT [B] for each block B in the data-flow graph.

4. Meaning of a Data-Flow Solution

We now know that the solution found using the iterative algorithm is the maximum fixedpoint, but what does the result represent from a program-semantics point of view? To understand the solution of a data-flow framework (D, F, V, A), let us first describe what an ideal solution to the framework would be.

Important Questions:

1. What is translator? Give examples of translators.
2. What is lex? Give the structure of LEX.
3. What is Left-factoring (LF) and how it is applied for the following grammar
   A→ab1|ab2|ab3
4. Explain in brief about leftmost, rightmost and mixed derivations.
5. What is the Error recovery actions in Lexical analyzer.
6. Explain the various phases of compiler with an illustrative example.
7. What is input buffering? Explain How is input buffering implemented.
8. Convert The regular expression R= a+ba* into NFA with ε moves.
9. Define ambiguous grammar. Check whether the grammar S→aAB, A→bC|cd, C→cd, B→c|d is ambiguous or not.
10. Consider S→a | ^ | (T) T→T, S | S
    a) Draw parse trees for (a,a) and (a,(a,a))
    b) Construct leftmost and rightmost derivation for (a,e) and (a,(a,e))
11. Find the LR(0) set of items for the following grammar and Describe state diagram and construct parse table of that S→CC, C→cC|d

12. What is compiler?
13. Explain the role of lexical analyser.
14. Define Context-Free Grammar (CFG).
15. Define Reductions and Handle pruning.
16. What is input buffering?
17. What are the Two parts of a compilation? Explain briefly.
18. Construct Transition Diagram for the token relation operators “relop”.
19. Write Rules for computing nullable, firstpos, and lastpos.
20. Consider the grammar E→E+E | E-E | E*E | E\E | a | b obtains leftmost and rightmost derivation for the string a + b* a-b.
21. What is Shift-reduce parsing? Consider the grammar

    S –> S - S

    S –> S * S

    S –> id

    Perform Shift Reduce parsing for input string “id - id * id”.
22. List out the rules for FIRST and FOLLOW? Construct FIRST and FOLLOW for the grammar

    E→ E+T | T

    T→ T * F | F

    F→(E) | id
23. Difference Between Compiler and Interpreter.
24. What are steps involved in Optimizing the DFA
25. Write Rules for FIRST() and FOLLOW() calculation on each non-terminal.
26. What is Left Recursion (LR) and how it is eliminated?
27. What does the terms LL and LR stands for?
28. Show the output generated by each phase for the following expression

    x= (a + b) * (c + d).
29. Explain Optimization of DFA-Based pattern Machines for the given regular expression (a|b)*abb#.
30. Explain in details about the role of Lexical analyzer with possible error recovery actions.
31. Show the following grammar S→AaAb | BbBa, A→ Є, B→ Є Is LL(1).
32. Find the LR(0) set of items for the following grammar and Describe state diagram and

construct parse table of that

S→AA

A→aA|b

33. What is Backtracking? Show the Backtracking in the following given grammar

    S -> r P d

    P -> m | m n

    Input string: 'rmnd.'

34. Define Type Equivalence?
35. Which graph is used for identifying the common sub expression in an expression?
36. Define Basic Block.
37. How can you identify the leader in a Basic block?
38. What is meant by Reaching Definitions.
39. Translate the following expression:

    $(a+b)^*(c+d)+(a+b+c)$ into quadruples and triples and indirect triples.

40. What is liveness?Explain liveness with suitable example.
41. Write a procedure to identify basic blocks.
42. Write the code generation for the d:=(a-b)+(a-c)+(a-c).
43. Discuss about the following:

    a) Copy Propagation b) Dead code Elimination and c) Code motion

44. Explain about data flow analysis.
45. Explain in detail about machine dependent code optimization techniques.
46. What is activation record?
47. What is dead code elimination and reduction in strength?
48. Define loop unrolling. Give an example.
49. What is meant by register descriptor and address descriptor?
50. How to allocate registers to instruction?
51. What is DAG? Construct DAG for the following Basic block.

    D: = B*C; E:= A+B; B:=B+C; A:=E-D;

52. Explain how copy propagation can be done using data flow equation.
53. Generate the code for the following expression: x = (a + b) – ((c + d) – e). Also Compute its cost.
54. Write a procedure to identify basic blocks.
55. What are the object code forms? Explain the issues in code generation.

56. Illustrate loop optimization with suitable example.
57. Explain in detail the procedure that eliminates global common sub expression.
58. Explain various method to handle peephole optimization.
59. Define Flow Graph? Explain how a given program can be converted in to flow graph.
60. What are the benefits of intermediate code generation?
61. What is a basic block?
62. Discuss about common sub expression elimination.
63. How do you calculate the cost of an instruction?
64. List out the common issues in the design of code generator.
65. Give an example to show how DAG is used for register allocation.
66. Generate intermediate code for the following code segment along with the required syntax directed translation scheme:

    ```
    if(a>b)
    x=a+b
    else
    x=a-b
    ```

    Where a and x are of real and b of int type data.
67. Explain the following with an example: a) Redundant sub expression elimination b) Frequency reduction c) Copy propagation.
68. Optimize the following code using various optimization techniques: i=1; s=0;

    ```
    for (i=1; i<=3; i++)
    for (j=1;j<=3;j++)
    c[i][j]=c[i][j] + a[i][j] + b[i][j]
    ```

69. Discuss and analyze all the allocation strategies in a run-time storage environment.
70. Write the algorithm for a simple code generator. And explain various issues that affect the efficiency of generated code.
71. Explain in brief about function preserving transformations on basic blocks.
72. Explain in brief about Induction variable elimination.

    Describe the application of peephole? What kinds of peephole techniques can be used to perform machine-dependent optimizations?

***