

# Compiler Design

# Syllabus

## UNIT-I:

**Formal Language and Regular Expressions:** Languages, operations on languages, regular expressions (re), languages associated with (re), operations on (re), Identity rules for (re)

**Finite Automata:** DFA, NFA, Conversion of regular expression to NFA, NFA to DFA, Applications of Finite Automata to lexical analysis, lex tools

## UNIT-II:

**Context Free grammars and parsing:** Context free Grammars, Leftmost Derivations, Rightmost Derivations, Parse Trees, Ambiguity Grammars, Top-Down Parsing, Recursive Descent Parsers: LL(K) Parsers and LL(1)Parsers

**Bottom up parsing:** Rightmost Parsers: Shift Reduce Parser, Handles, Handle pruning, Creating LR (0) Parser, SLR (1) Parser, LR (1) & LALR (1) Parsers, Parser Hierarchy, Ambiguous Grammars, Yacc Programming Specifications.

## UNIT-III:

**Syntax Directed Translation:** Definitions, construction of Syntax Trees, S-attributed and L-attributed grammars, Intermediate code generation, abstract syntax tree, translation of simple

statements and control flow statements

# Syllabus

## UNIT-IV:

**Semantic Analysis:** Semantic Errors, Chomsky hierarchy of languages and recognizers, Type checking, type conversions, equivalence of type expressions, Polymorphic functions, overloading of functions and operators.

## UNIT-V:

**Storage Organization:** Storage language Issues, Storage Allocation, Storage Allocation Strategies, Scope, Access to Nonlocal Names, Parameter Passing, Dynamics Storage Allocation Techniques.

## UNIT-VI:

**Code Optimization:** Issues in the design of code optimization, Principal sources of optimization, optimization of basic blocks, Loop optimization, peephole optimization, flow graphs, Data flow analysis of flow graphs.

**Code Generation:** Issues in the design of code Generation, Machine Dependent Code Generation, object code forms, generic code generation algorithm, Register allocation and assignment, DAG representation of basic Blocks, Generating code from DAGs.

# Unit – 1

## Formal Language and Regular Expressions

1. Languages, operations on languages
2. Regular expressions (re)
3. Languages associated with re
4. Operations on (re)
5. Identity rules for (re)

## Finite Automata

6. DFA
7. NFA
8. Conversion of regular expression to NFA
9. NFA to DFA
10. Applications of Finite Automata to lexical analysis
11. lex tools

# Basic Terminology:

## Computation:

- Computation can be defined as finding solution to a problem from given inputs by means of algorithm

## Symbol:

- A symbol is a single object / character
- Normally, characters from a typical keyboard are used as symbols

**Example:** A, a,  $\alpha$ (alpha),  $\lambda$ (Lambda), etc...

## Alphabet:

- An alphabet is a finite, non-empty set of symbols / characters and is denoted by  $\Sigma$

**Example:**  $\Sigma = \{0, 1\}$  is the binary alphabet, consisting of the symbols 0 and 1

$\Sigma = \{a, b, c\}$  is an alphabet of three symbols

$\Sigma = \{A, B, \dots, Z\}$  is the uppercase English alphabet

- Power of an alphabet is the set of all strings of length k over alphabet i.e.,  $s^k$

**Example:**  $S = \{0, 1\}$  □  $s^1 = \{0, 1\}$ ,  $s^2 = \{00, 01, 10, 11\}$ ,  $s^3 = \{000, \dots, 111\}$ , etc...

## Basic Terminology (Cont...):

**String:** It is a finite sequence of symbols chosen from  $\Sigma$  and is denoted by  $w$  or  $s$

**Example:** If  $\Sigma = \{0, 1\}$  then 111, 11, 11, 10, 01... are some of the strings chosen from this  $\Sigma$

If  $\Sigma = \{a, b\}$  then aa, bb, ab, ba, abab, aabb... are the words chosen from this  $\Sigma$

□ Length of a string  $w$  is the no. of symbols composing the string and is denoted by  $|w|$

**Example:** If  $w = aabb$  then  $|w| = 4$

If  $w = 010110$  then  $|w| = 6$

□ Empty string is the string with zero symbols and zero length & is denoted by  $\epsilon$  &  $|\epsilon|=0$

□ A prefix of a string is any no. of leading symbols of that string

□ A suffix of a string is any no. of trailing symbols of that string

**Example:** string abc □ prefix:  $\epsilon$ , a, ab, abc and suffix:  $\epsilon$ , c, bc, abc

□ A prefix / suffix of a string other than the string itself is called a proper prefix / suffix

□ Concatenation of 2 strings is writing the 1<sup>st</sup> string followed by 2<sup>nd</sup> string with no space

**Example:** Let  $x=ab$ ,  $y=pq$  then  $xy=abpq$  and  $yx=pqab$

The empty string  $\epsilon$  is the identity for the concatenation operator i.e.,  $\epsilon w = w\epsilon = w$

**Kleene star:** It is the set of all strings over  $\Sigma$  is conventionally denoted by  $\Sigma^*$

**Example:** If  $\Sigma = \{0,1\}$  then  $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots, 111, \dots\} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$

**Positive closure:** It is the set of all non-empty strings over  $\Sigma$  is denoted by  $\Sigma^+$

**Example:** If  $\Sigma = \{0,1\}$  then  $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, \dots, 111, \dots\} = \Sigma^* - \{\epsilon\}$

# Formal Language:

## Automaton:

- An automaton (plural: automata) is a self-operating machine. The term "Automata" is derived from the Greek word "αὐτόματα".

## Language:

- A language L is a set of strings over the given alphabet and it can be finite or infinite

**Example:** If  $\Sigma = \{a, b\}$  then

1.  $L1 = \{ \text{set of all strings of length 2} \} = \{aa, ab, ba, bb\}$  --- Finite
2.  $L2 = \{ \text{set of all strings which starts with a} \} = \{a, aa, ab, aaa, aba, abb, \dots\}$  --- Infinite
3. Language containing the null string is  $\{ \epsilon \}$
4. Null language:  $\{ \} = \phi$

## **Operations on Language:**

- |                         |                                     |
|-------------------------|-------------------------------------|
| 1. Union (or)           | 6. Complementation (not)            |
| 2. Intersection (and)   | 7. Symmetric difference (xor)       |
| 3. Concatenation (join) | 8. Length sub-setting of a language |
| 4. Reversal             | 9. Kleene star / Positive closure   |
| 5. Palindrome           | 10. De Morgan's Laws                |

# Operations on Language:

## 1. Union:

□ If  $L_1$  &  $L_2$  are two languages then the  $L_1 \cup L_2$  containing all strings from both languages

**Example:** Let  $L_1 = \{0,01,10,11\}$ ,  $L_2 = \{0,01,001\}$  □  $L_1 \cup L_2 = \{0,01, 001,10,11\}$

Let  $L_1 = \{0,00,000,\dots\}$ ,  $L_2 = \{0,000,0000,\dots\}$  □  $L_1 \cup L_2 = \{0\}^+$

## 2. Intersection:

□ If  $L_1$  &  $L_2$  are two languages then the  $L_1 \cap L_2$  containing common strings from both languages

**Example:** Let  $L_1 = \{0,01,10,11\}$ ,  $L_2 = \{0,01,001\}$  □  $L_1 \cap L_2 = \{0,01\}$

Let  $L_1 = \{0,00,000,\dots\}$ ,  $L_2 = \{0,000,0000,\dots\}$  □  $L_1 \cap L_2 = \{0,000\}$

## 3. Concatenation:

□ If  $L_1$  &  $L_2$  are two languages then the  $L_1 L_2$  containing all strings

**Example:** Let  $\Sigma = \{0,1\}$  and  $L_1, L_2$  are two languages over  $S$

$L_1 = \{00,11\}$ ,  $L_2 = \{0,1\}$  □  $L_1 L_2 = \{000,001,110,111\}$

□  $L_2 L_1 = \{000,011,100,111\}$



# Operations on Language (Cont...):

## 4. Reversal:

□ This operation is similar to the reversal of a string operation. It can be written as  $L^R = \{W^R \mid W \in L\}$

**Example:** If  $L = \{ab, bc, cd\}$  then,  $L^R = \{ba, cb, dc\}$

## 5. Palindrome:

□ A language called Palindrome over  $\Sigma = \{a, b\}$  is defined as:

Palindrome =  $\{\epsilon, \text{all strings } W, \text{ such that } W^R = W \text{ and } W \in \Sigma\} \equiv \{\epsilon, a, b, aa, bb, aaa, bbb, aba, abba, aaaa, \dots\}$

## 6. Complementation:

□ Let  $L$  is a language over an alphabet, then the compliment of  $L$  is denoted by  $L^1$

□ It consisting of all those strings that are not in  $L$  over the alphabet

□ Mathematically, it is expressed as  $x \in L^1$  if and only if  $x \in \Sigma^* - L$

**Example:** If  $\Sigma = \{a, b\}$  and  $L = \{a, b, aa\}$  then the complement of  $L$  is  $\{\epsilon, ab, ba, bb, \dots\}$

i.e.,  $L^1 = \Sigma^* - L = \{\epsilon, a, b, aa, bb, ab, ba, aaa, bbb, \dots\} - \{a, b, aa\} = \{\epsilon, bb, ab, ba, \dots\}$

# Operations on Language (Cont...):

## 7. Symmetric difference:

- Let  $L_1$  and  $L_2$  are two languages over an alphabet  $\Sigma$  and is denoted by  $L_1 \oplus L_2$
- It can be defined as  $L_1 \oplus L_2 = (L_1 \cup L_2) - (L_1 \cap L_2)$
- Elements of  $L_1 \oplus L_2$  are contained either in  $L_1$  or  $L_2$  but not in both

### **Example:**

Let  $L$  be a language over an alphabet  $\Sigma$  then clearly  $L \oplus \phi = L$ ,

$$L \oplus L = \phi,$$

$$L \oplus \Sigma^* = L^1$$

$$L \oplus L^1 = \Sigma^*$$

## 8. Length sub-setting of a language:

- Let  $L$  is a language over an alphabet
- Then for sake of convenience or compactness, it may be required to specify the strings in  $L$ , of length less than or equal to a specific value or a fixed size

### **Example:**

Let  $L = \{1, 11, 111, 1111, \dots\}$  then  $L_{=4} = \{1111\}$  and  $L_{\leq 4} = \{1, 11, 111, 1111\}$

# Operations on Language (Cont...):

## 9. Kleene Closure and Positive Closure:

□ Kleene star is the set of all strings of any length over  $\Sigma$  and is denoted by  $\Sigma^*$

**Example:** If  $\Sigma = \{0,1\}$  then  $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots, 111, \dots\} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$

□ Positive closure is the set of all non-empty strings over  $\Sigma$  is denoted by  $\Sigma^+$

**Example:** If  $\Sigma = \{0,1\}$  then  $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, \dots, 111, \dots\} = \Sigma^* - \{\epsilon\}$

## 10. De Morgan's Laws:

□ It allows one to express the intersection of two languages over  $\Sigma$ , in terms of the two operation like, union and complementation

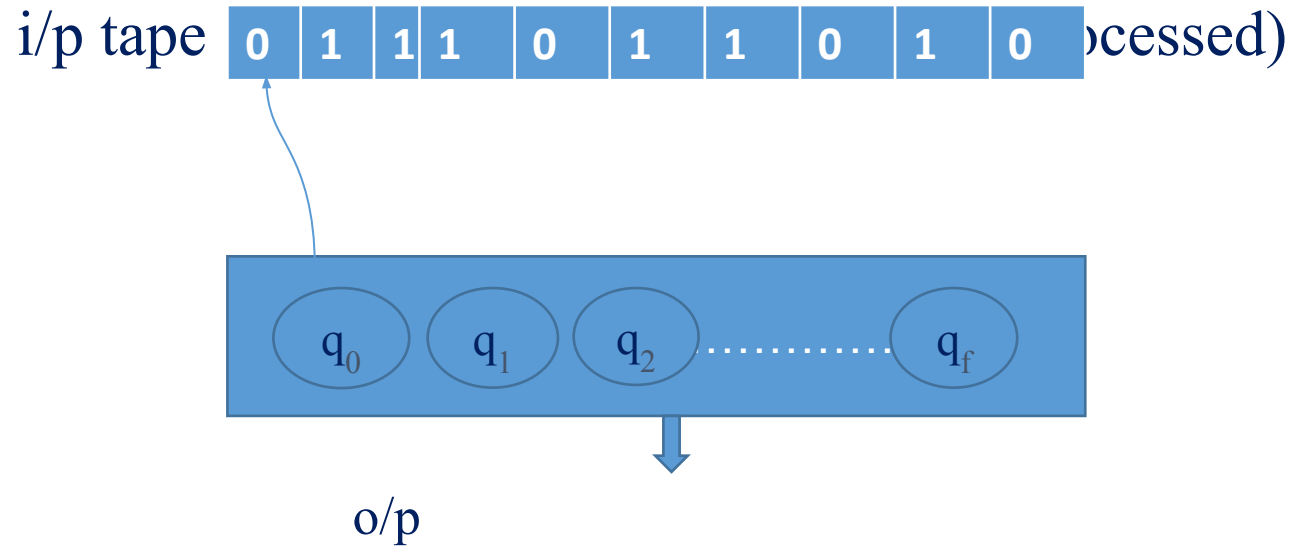
□ These are the fundamental laws in set theory

Let  $L_1, L_2, L_3$  be any three languages, then  $L_1 - (L_2 \cap L_3) = (L_1 - L_2) \cup (L_1 - L_3)$

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

# Finite Automata:

- An automata is defined as a system or model where information is transformed and used for performing some functions without direct participation of man.
- Below figure shows Finite Automata (FA) model



## Input:

- ✓ It is divided into several cells and each cell stores a single character
- ✓ At each discrete instance of time  $t_1, t_2, \dots, t_n$ , i/p values are  $i_1, i_2, \dots, i_n$
- ✓ Each of which can take a finite number of fixed values from i/p alphabet summation ( $\Sigma$ ) are applied to the i/p side of the model

# Finite Automata (Cont...):

## Output:

- ✓  $o_1, o_2, \dots, o_n$  are outputs of the model
- ✓ Each of which can take finite number of fixed values, common o/p that is yes / no in other way accepted/not accepted

## States:

- ✓ At any instant of time the FA can be one of the states  $q_0, q_1, \dots, q_n$

## State relation:

- ✓ At any instant of time the next state of automata is determined by the present state, present i/p

## Output relation:

- ✓ Output is related to either state only **or** both i/p and state

# Finite Automata (Cont...):

## Applications:

- Traffic lights
- Video Games
- Speech Recognition
- Text Parsing
- Natural Language Processing
- Highly useful in designing Lexical analyzers, designing text editors, designing spell checkers, designing Sequential Circuit design (or) Hardware Design, String processing, designing BOT (Web Robot)

## Types:

- Deterministic Finite Automata (DFA / DFSM)
- Non-Deterministic Finite Automata (NDFA / NFA / NDFSM)

# Deterministic Finite Automata (DFA):

- It consists of finite no. of states and set of transitions from one state to another state on i/p symbols choosing from an input alphabet
- For each i/p symbol their exactly only one transition (only one path from current state to next state on same i/p symbol)
- Analytically it can be defined as quin tuple or 5 tuples:  $\mathbf{Q, \Sigma, \delta, q_0, F}$

Where  $\mathbf{Q}$  is finite non-empty set of states

$\Sigma$  is input alphabet summation

$\delta$  is transition function  $\mathbf{Q * \Sigma \rightarrow Q}$

$\mathbf{q_0}$  is start / initial state (an arrow along with first state)

$\mathbf{F}$  is set of final states (double circles)

- For each state in  $\mathbf{Q}$ , there is a corresponding node in the transition diagram
- For each state in ' $\mathbf{Q}$ ' and each i/p symbol ' $\mathbf{a}$ ' in ' $\Sigma$ ' and if  $\delta(\mathbf{Q}, \mathbf{a}) = \mathbf{P}$  then there is an arc or edge from  $\mathbf{Q}$  to  $\mathbf{P}$

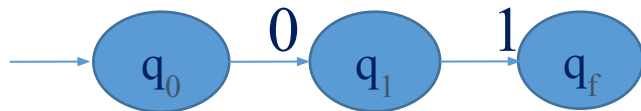
# Deterministic Finite Automata (DFA) Cont...:

## Transition Diagram:

- It is a directed graph associated with finite automata, the vertex of the graph corresponds to the state of the finite automata
- If there is a transition from state **P** to state **Q** on i/p symbol **i** then there is an edge / arc labelled by **i** from **P** to **Q**
- The finite automata accepts a string “**X**”, if the sequence of transitions corresponding to the symbols of “**X**” leads from the starting to the final state

## □ Transition Table:

- It is a conventional tabular representation of a transition function “ $\delta$ ” that takes two arguments *i.e., state and i/p symbol* and returns a value *i.e., state*



□

$\delta$	0	1
$q_0$	$q_1$	--
$q_1$	--	$q_f$
$q_f$	--	--

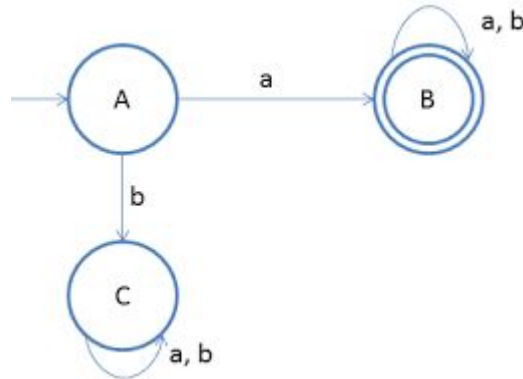


# Deterministic Finite Automata (DFA) Cont...:

## Dead / Dead End State :

- A rejecting state that is essentially a dead end
- Once the machine enters a dead state, there is no way for it to reach an accepting state, so we already know that the string is going to be rejected
- Graphically, the dead state is often omitted and assumed for any input that the machine does not have explicit instructions on what to do with
- A machine may have multiple dead states, but at most only one dead state is needed per machine

**Ex:**  $R = a(a+b)^*$



Here A is start state, B is final state and C is dead state

# Problems on DFA:

- Design a DFA which accepts the string 1010 only
- Design a DFA which accepts the string 1010 or 1100
- Design a DFA which accepts set of all binary strings
- Design a DFA over  $\Sigma=\{0,1\}$  in which the strings starting with 1 and such that the number of 0's is divisible by 3
- Design a DFA over  $\Sigma=\{0,1\}$  which accepts set of all strings consists 3 consecutive 0's
- Design a DFA over  $\{0,1\}$  that doesn't accepts set of all strings consists 3 consecutive 0's
- Construct a DFA that accepts all strings **1.** with exactly one 'a'      **2.** with atleast one 'a'
- Design a DFA to accept the language  $L=\{x \in \{a,b,c\}^* \mid |x|_a \text{ is divisible by } 3\}$
- Construct a DFA that accepts the strings having **1.** even no. of a's   **2.** all strings with atleast one 'a' and exactly two b's over  $\Sigma=\{a,b\}$
- Design a DFA over  $\Sigma=\{a,b\}$  which accepts strings having **1.** even no. of a's & even no. of b's   **2.** odd no. of a's & odd no. of b's   **3.** even no. of a's & odd no. of b's   **4.** odd no. of a's & even no. of b's
- Design a DFA over  $\Sigma=\{0,1\}$  which accepts set of all strings ending with 00
- Construct a DFA over  $\Sigma=\{a,b\}$  that accepts the strings which is not more than 3 a's
- Design a DFA which accepts set of binary strings that begins and ends with the same symbol

# Non-Deterministic Finite Automata (NFA):

- It allows 0 / 1 / more transitions from a state for the same input alphabet
- For every language accepted by some NFA, DFA also accept the same language
- Analytically it can be defined as quin tuple or 5 tuples:  $\mathbf{Q}, \Sigma, \delta, \mathbf{q}_0, \mathbf{F}$

Where  $\mathbf{Q}$  is finite non-empty set of states

$\Sigma$  is input alphabet summation

$\delta$  is transition function  $\mathbf{Q} * \Sigma \rightarrow 2^{\mathbf{Q}}$

$\mathbf{q}_0$  is start / initial state

$\mathbf{F}$  is set of final states

# DFA Vs NFA:

S. No.	DFA	NFA
1	Deterministic Finite Automata	Non-Deterministic Finite Automata
2	Transition function $Q * \Sigma \rightarrow Q$	Transition function $Q * \Sigma \rightarrow 2^Q$
3	It allows only one transition on each i/p symbol	It allows 0 / 1 / more transitions on each i/p symbol
4	It requires more space	It requires less space
5	Construction is more difficult	Construction is easier than DFA
6	It can't use an empty string transition	It can use an empty string transition
7	Backtracking is allowed	Backtracking is not allowed

# Problems on NFA:

- Design NFA over  $\Sigma=\{0,1\}$  which accepts set of all strings with 3 consecutive 0's
- Design NFA over  $\Sigma=\{0,1\}$  which accepts set of all strings ending with 00
- Design NFA over  $\Sigma=\{0,1\}$  which accepts all strings containing 1100 as a substring
- Design NFA over  $\{0,1\}$  which accepts set of all strings such that the 3<sup>rd</sup> symbol from right end is 1
- Design NFA over  $\{0,1\}$  which accepts all strings such that 3<sup>rd</sup> symbol & 2<sup>nd</sup> symbol from left end is 1 & 0
- Design NFA over  $\{0,1\}$  which accepts the strings having either two consecutive 0's & two consecutive 1's
- Design NFA which accepts set of all binary strings containing 1010 or 1100
- Design NFA which accepts set of all strings in  $\{a,b,c\}^*$  such that the last symbol in input string also appears earlier in the string
- Consider the given NFA and check the acceptance of strings    1. 01001      2. 1000 3. 101

# Conversion from NFA to DFA:

- Let NFA,  $M = (Q, \Sigma, \delta, q_0, F)$  and its equivalent DFA is  $M^1 = (Q^1, \Sigma^1, \delta^1, q_0^1, F^1)$
- While construction of DFA from the given NFA, the input alphabet and the starting state are same for both NFA and DFA
- $Q^1$  is the set of all subsets of  $Q$
- $F^1$  is the set of all subsets of  $Q$  which are having any final state of given NFA
- $\delta^1$  is calculated from  $\delta$  as given below
  - If  $q_1$  &  $q_2$  are two states and 'a' is an input alphabet then
    - $\delta^1(q_1, a) = \delta(q_1, a)$
    - $\delta^1((q_1, q_2), a) = \delta(q_1, a) \cup \delta(q_2, a)$
- We stop or halt the process when no new state appears

## NFA with null / $\epsilon$ -moves:

- Finite automata with null /  $\epsilon$ -transitions is same as NFA except that we use a special input symbol called “ $\epsilon$ ”
- Using this symbol we can jump to any state without reading any input symbol
- Analytically defined as  $M = (Q, \Sigma, \delta, q_0, F)$

where  $\delta$  is transition function  $Q * (\Sigma \cup \epsilon) \rightarrow 2^Q$

- If a language “L” is accepted by NFA with  $\epsilon$ -transitions then the same language is accepted by NFA

### $\epsilon$ -closure:

- $\epsilon$ -closure of a state ‘Q’ is defined as a set of all states ‘P’ such that there is a path from Q to P ( $Q \rightarrow P$ ) labelled by  $\epsilon$  only

# Conversion from NFA with $\epsilon$ -moves to NFA without $\epsilon$ -moves:

- Let NFA,  $M = (Q, \Sigma, \delta, q_0, F)$  with  $\epsilon$ -moves then there exists NFA is  $M^1 = (Q, \Sigma, \delta^{\wedge}, q_0, F^1)$  without  $\epsilon$ -moves
- Find the  $\epsilon$ -closure of all states in the given NFA
- Calculate extended transition function using following conversion formulas for all the states with all possible input symbols
  - $\delta^{\wedge}(Q, \epsilon) = \epsilon\text{-closure}(Q)$
  - $\delta^{\wedge}(Q, a) = \epsilon\text{-closure}(\delta(\delta^{\wedge}(Q, \epsilon), a)) \equiv \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(Q), a))$
- $F^1$  is the set of all final states whose  $\epsilon$ -closure contains a final state of given NFA i.e., the state which is reachable to the final state only on  $\epsilon$  symbol is going to be final state



# Regular Expression:

- A language 'L' is accepted by finite automata that is expressed in an expression is known as regular expression
- Analytically it is defined as
  1. Any terminal symbol i.e., an element of  $\Sigma$ ,  $\epsilon$ ,  $\emptyset$  are regular expressions
  2. The union of 2 regular expressions  $R_1$ ,  $R_2$  written as  $R_1 + R_2$  is also a regular expression
  3. The concatenation of 2 regular expressions  $R_1$ ,  $R_2$  written as  $R_1 R_2$  is also a regular expression
  4. The iteration or closure of a regular expression  $R$  is written as  $R^*$  is also a regular expression
  5. If  $R$  is a regular expressions then  $R^r$  is also a regular expression

**Example:** Describe the following set by RE

1.  $\{101\} = 101$
2.  $\{ab\} = ab$
3.  $\{\epsilon, ab\} = \epsilon + ab$
4.  $\{1, 11, 111, 1111, \dots\} = 1^+$
5.  $\{\epsilon, 1, 11, 111, 1111, \dots\} = 1^*$

## **Regular Set:**

- The set is represented by the regular expression is called regular set

# Closure properties of Regular sets:

□ If a class of language is closed under a particular operation then we can call it as closure property of class of language

**1. Union:** The regular sets are closed under union. If  $L_1, L_2$  are two RE's then  $L_1 \cup L_2$  is also a RE

**2. Concatenation:** The regular sets are closed under concatenation. If  $L_1, L_2$  are two RE's then  $L_1 L_2$  also a RE

**3. Intersection:** The regular sets are closed under intersection. If  $L_1, L_2$  are two RE's then  $L_1 \cap L_2$  also a RE

**4. Kleen star:** The regular sets are closed under kleen star. If  $L_1$  is a RE then  $L_1^*$  also a RE

**5. Complement:** The regular sets are closed under complement. If  $L_1$  is a RE then  $L_1'$  also a RE

**6. Reverse:** The regular sets are closed under reverse. If  $L_1$  is a RE then  $L_1^r$  also a RE

**7. Difference:** The regular sets are closed under difference. If  $L_1, L_2$  are two RE's then  $L_1 - L_2 (L_1 \cap L_2')$  also a RE

**8. Substitution:** The regular sets are closed under substitution. If  $L_1$  is a RE then  $S(L_1)$  also a RE

**9. Homomorphism:** The regular sets are closed under homomorphism. If  $L_1$  is a RE then  $H(L_1)$  also a RE

**10. Inverse Homomorphism:** The regular sets are closed under inverse homomorphism. If  $L_1$  is a RE then  $H^{-1}(L_1)$  is also a RE

# Problems on RE:

- Write the RE for the following sets
  - $L_1$  = set of all strings of 0's and 1's ending with 00
  - $L_2$  = set of all strings of 0's and 1's beginning with 0 and ending with 1
  - $L_3$  =  $\{\epsilon, aa, aaaa, \dots\}$
- Write RE to denote any no. of 0's followed by any no. of 1's followed by any no. of 2's
- Write RE to denote any no. of 0's followed by any no. of 1's followed by any no. of 2's with at least any one of the each input symbol
- Write the RE for the following sets
  - $L_1$  =  $\{W \in (0,1)^*, W \text{ has at least one pair of consecutive 0's}\}$
  - $L_2$  =  $\{W \in (0,1)^*, \text{all the strings having at least two 0's}\}$
  - $L_3$  =  $\{W \in (0,1)^*, \text{all the strings having exactly two 0's}\}$
- Write RE for set of all strings with even no. of a's followed by odd no. of b's
- Write RE for set of all strings exactly one a over the alphabet  $\Sigma = \{a, b, c\}$
- Write RE for ternary language
- Write RE for the strings having 101 as a substring

# Identity rule for RE:

□ Two RE's are equivalent if P and Q represents the same set of strings

1.  $\emptyset + R = R + \emptyset = R$
2.  $\emptyset R = R \emptyset = \emptyset$
3.  $\epsilon R = R \epsilon = R$
4.  $\epsilon^* = \epsilon$
5.  $\emptyset^* = \epsilon$
6.  $R + R = R$
7.  $R^* R^* = R^*$
8.  $RR^* = R^* R = R^+$
9.  $(R^*)^* = R^*$
10.  $\epsilon + RR^* = \epsilon + R^* R = R^*$
11.  $(PQ)^* P = P(QP)^*$
12.  $(P+Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$
13.  $(P+Q)R = PR+QR$
14.  $R(P+Q) = RP+RQ$

# Language Processing System / Steps for Executing a Program:

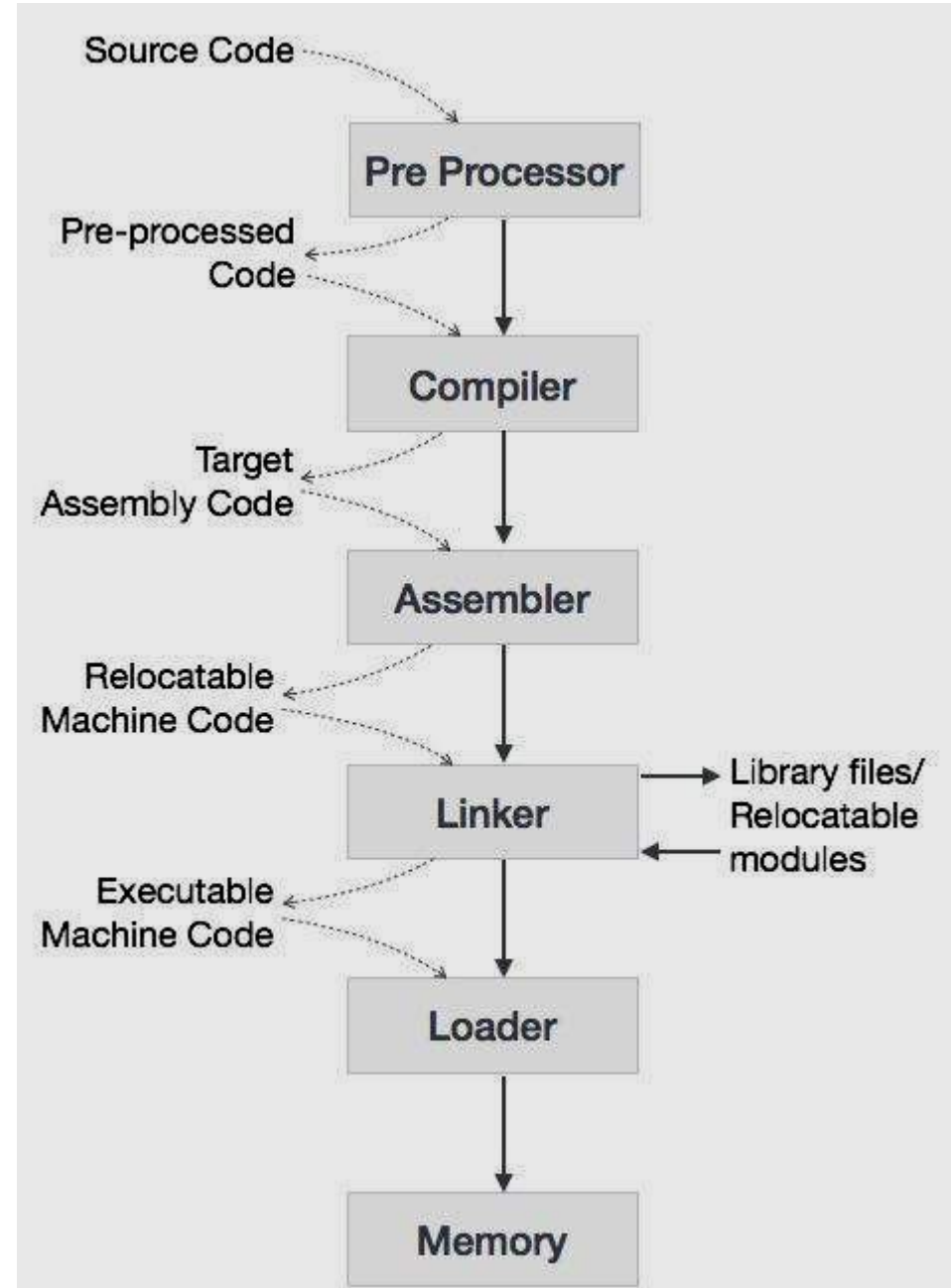
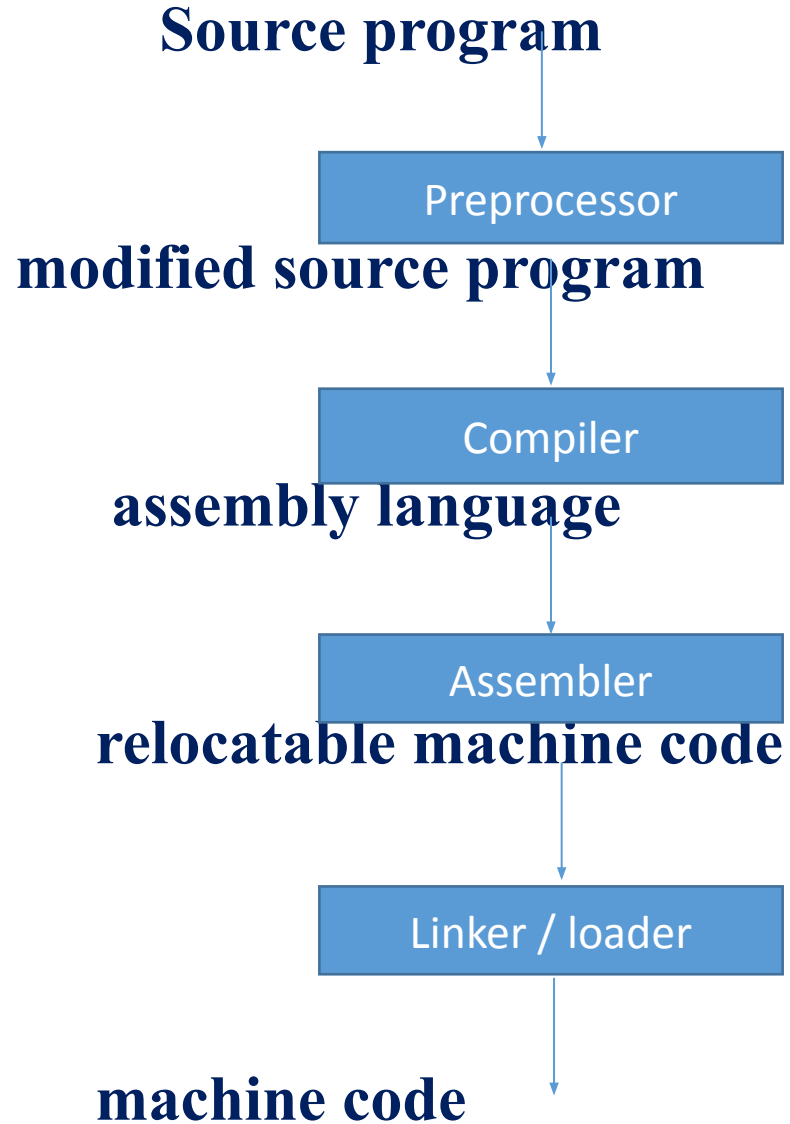
## Computer:

- Combination of software and hardware
  - Hardware Physical parts of a computer (keyboard, mouse, monitor, printer etc...)
  - Software: It is a program (set of instructions which perform the given task)
    - Application Software: User develop a program/software (Banking, railway etc)
    - System Software: User can work with the system by using this software (OS, Compiler, Debugger etc...)

## Languages:

- High level languages: By using this we can develop application software (symbols)  
**EX:** C++, Java, C#
- Medium level Languages: Combination of both the features **EX:** C
- Low level Languages: By using this we can develop system software (0, 1) **EX:** OS

# Language Processing System / Steps for Executing a Program:



# Language Processing System / Steps for Executing a Program:

## Preprocessor:

- It takes the source code as input and generates modified source code
- It removes the preprocessor statements and replaces with the definition of this
- The statements which are processed before compilation i.e., code with no preprocessor statements
- Two types of preprocessor
  - File inclusion (`#include<stdio.h>` i.e., header files **Example**: `printf()`, `clrscr()`, etc...)
  - Macros (`#define PI 3.14` i.e., static values **Example**: `PI` , etc...)

## Compiler:

- It takes the modified source code as input and generates assembly language
- It checks whether the given program follows the language syntax rules or not

# Language Processing System / Steps for Executing a Program:

## Assembler:

- It takes the assembly language as input and generates relocatable machine code
  - Logical address: This is generated by the program
  - Physical address: It checks the free memory location in main memory
  - $\text{physical address} = \text{logical address} + \text{offset}$

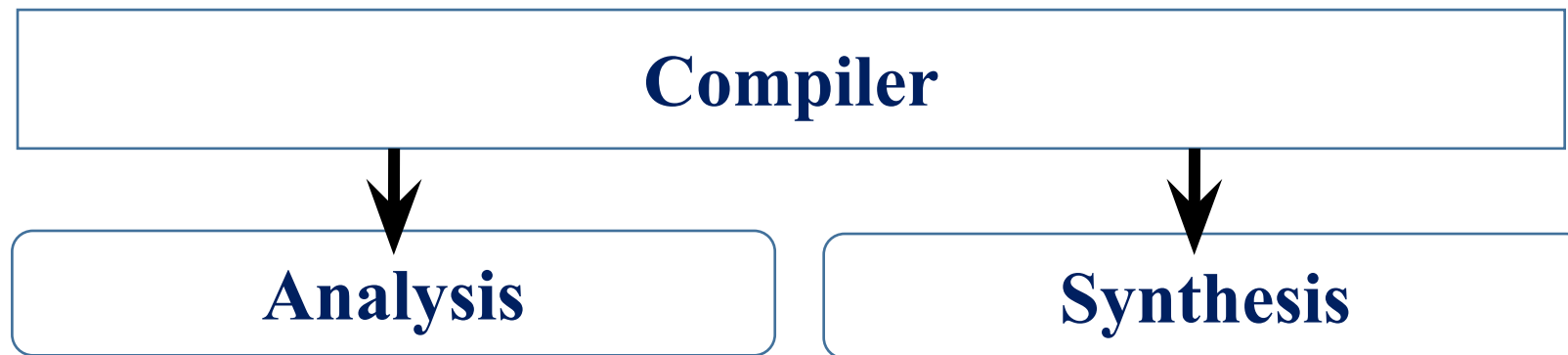
## Linker / Loader:

- It takes the relocatable machine code as input and generates machine code
- It links relocatable m/c code of various library functions to main program
- Loader loads the executable files (.exe) into main memory for execution

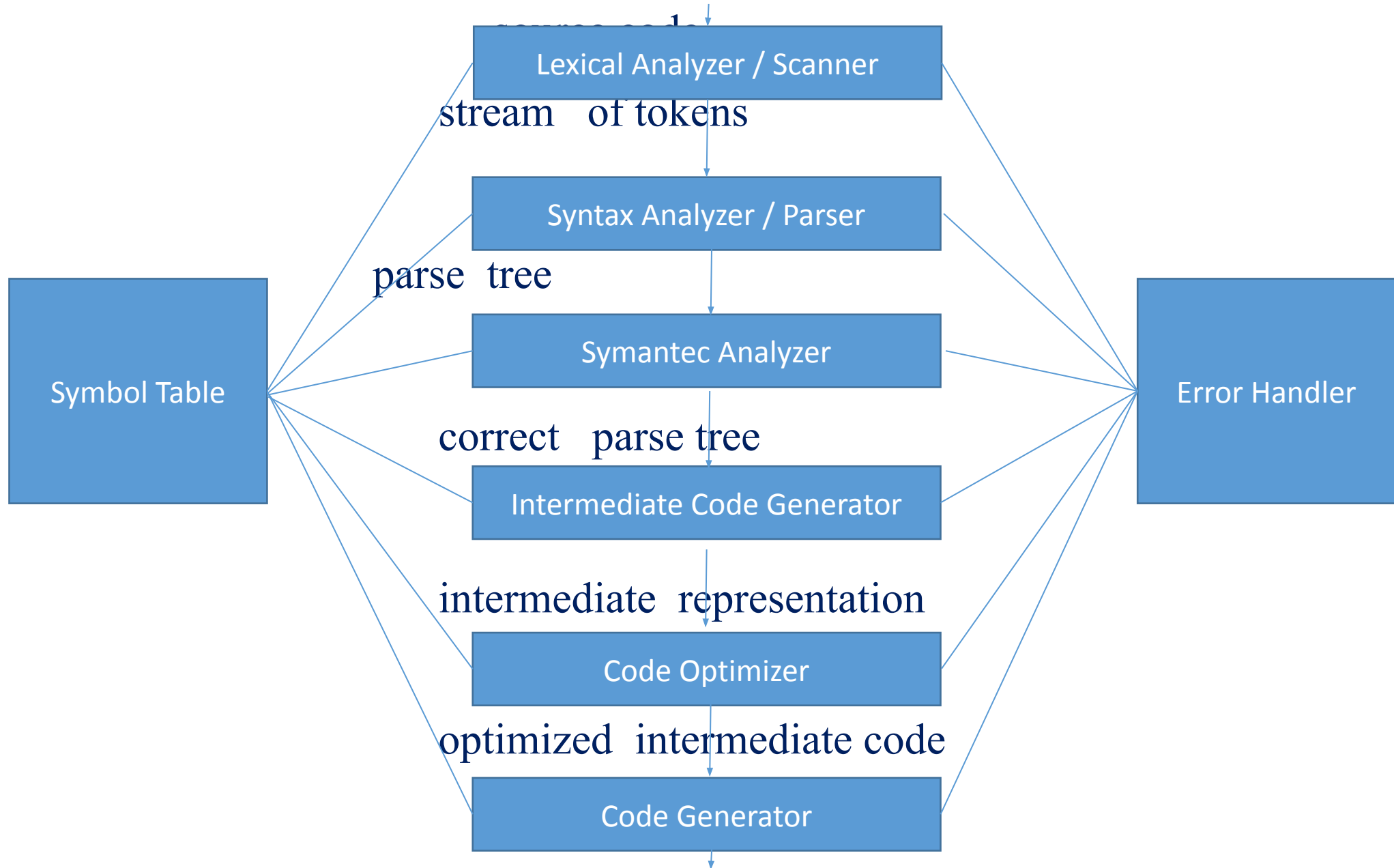


# Structure of a Compiler:

- Basically there are two phases of compilers
  - Analysis phase
  - Synthesis phase
- Analysis phase creates an intermediate representation from the given source code
- Synthesis phase creates an equivalent target program from the intermediate representation



# Phases of a compiler:



## Phases of a compiler (Cont...):

### 1. Lexical Analyzer / Scanner:

- It reads the program character by character and converts it into meaning full sequences (lexeme □ a sequence of character)
- Each lexeme is represented in the form of tokens (identifier, keyword, constant, operator)
- Each token is represented in the form of <token name, attribute value>
- Tokens are defined by regular expressions which are understood by the lexical analyzer
- Transition diagram is used to recognizing the tokens

### 2. Syntax Analyzer / Parser:

- It checks the syntax for the corresponding code is correct or not
- It takes the token one by one and uses Context Free Grammar to construct the parse tree
- While constructing a parse tree, first identify the symbol with high priority as internal node and left & right child of internal node may be identifier or constant

## Phases of a compiler (Cont...):

### 3. Semantic Analyzer:

- It verifies the parse tree, whether it's meaningful or not otherwise it produces a verified parse tree
- It uses a software/ tool called type checker, will check the datatype of a variable and it performs if any need of type conversion

### 4. Intermediate Code Generator:

- It generates the code that is very easy to convert this to machine code
- Many ways to represent the intermediate code representation like three address, polish notation, etc...
  - **Three address code**
    - Assignment instruction must have at most one operator on the right hand side
    - Compiler must generate temporary variables for storing the result
    - Instructions may contain fewer than 3 operands i.e., variables

## Phases of a compiler (Cont...):

### 5. Code Optimizer:

- It removes the unnecessary statements in the code so the length of the program is reduced as well as memory is saved and the CPU can execute the program in a fast manner
- Optimisation can be categorized into two types
  - machine dependent
  - machine independent

### 6. Code Generator:

- The main purpose of code generator is to write a code that the machine can understand
- Inorder to perform the operations it uses some assembly level languages instructions like **LD R,M, ST M,R, ADD R1,R2, ADD R1,R2,R3, etc...**

## Phases of a compiler (Cont...):

### Symbol Table:

- It is a data structure used and stores information about the tokens
- It helps the compiler to function smoothly by finding the identifiers quickly

### Error Handler:

- The tasks of the Error Handler are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error

# Example:

```
position := initial + rate * 60
```

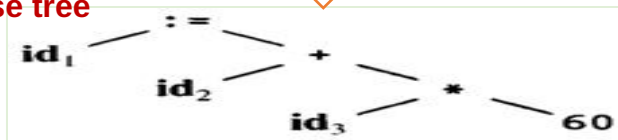
**Scanner**  
[Lexical Analyzer]

Tokens

```
id1 := id2 + id3 * 60
```

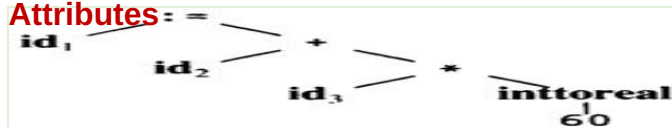
**Parser**  
[Syntax Analyzer]

Parse tree



**Semantic Process**  
[Semantic analyzer]

Abstract Syntax Tree w/  
Attributes



1	position	...
2	initial	...
3	rate	...
4		



**Code Generator**  
[Intermediate Code Generator]

Non-optimized Intermediate  
Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

**Code Optimizer**

Optimized Intermediate  
Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

**Code Optimizer**

Target machine  
code

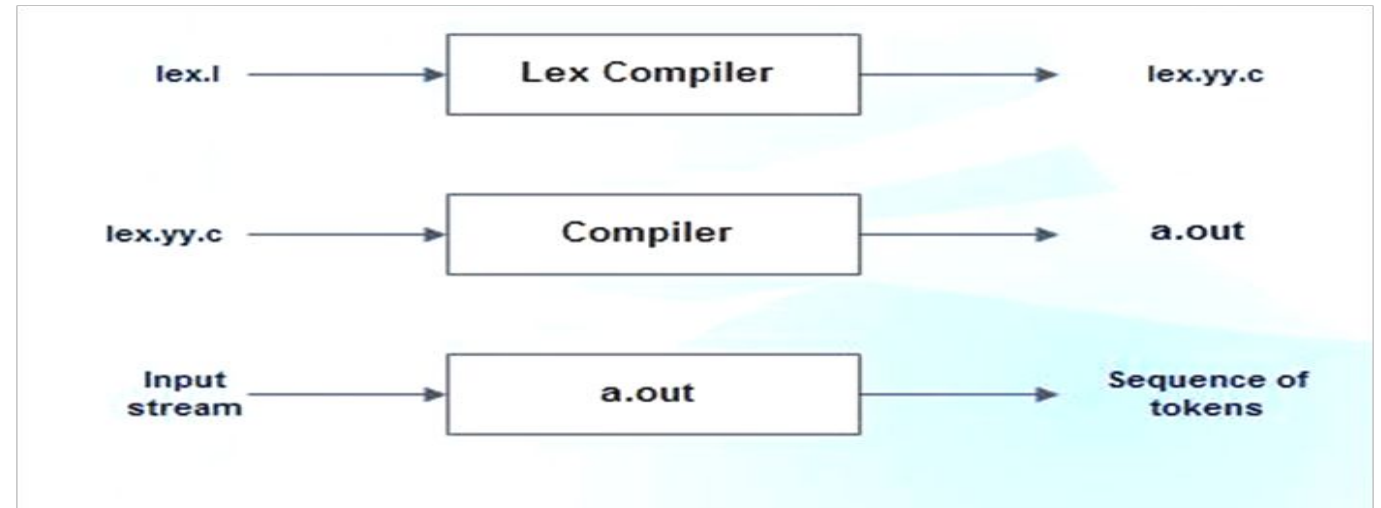
```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

# LEX Tool:

- A language for specifying lexical analyzer or The lexical analyzer generator (LEX)
- Lex is a tool or language in order to generate lexical analyzer & itself is a lex compiler
- Lex is an acronym that stands for "lexical analyzer generator "

**Example:** lex, jflex, BOT, etc ...

- Lex file execution □



## Lex program structure:

1. declarations
2. %%  
translation rules  
%%
3. auxiliary functions



# LEX Tool (Cont...):

## Declarations:

□ Used to declare c variables & constants and must be between `%{`      `%}`

□ **Example**    `%{ int a,b; float count=0; %}`

□ Used to define regular expression and no need to include `%{`      `%}`

□ **Example**    `digit [0-9]    letter [a-zA-Z]`

## Translation rules:

□ Rules must write in between `%%`      `%%`

□ **Syntax:** `pattern {action}` where pattern is a RE and action is a C language statements

□ Must separate pattern and action with at least one blank space

□ `%%`

`pattern1 {action1}`

`pattern2 {action2}`

`pattern3 {action3}`

`%%`

□ Must write each pattern in a new line

## Auxiliary functions:

□ All the functions are defined in this section

# **Unit – 2**

## **Context Free grammars and parsing**

1. Context free Grammars
2. Leftmost and Rightmost Derivations
3. Parse Trees
4. Ambiguity Grammars
5. Top-Down Parsing
6. Recursive Descent Parsers
7. LL(K) Parsers
8. LL(1)Parsers

## **Bottom up parsing**

9. Rightmost Parsers
10. Shift Reduce Parser
11. Handles and Handle pruning
12. Creating LR (0) Parser
13. SLR (1) Parser
14. LR (1)
15. LALR (1) Parsers
16. Parser Hierarchy
17. Ambiguous Grammars
18. Yacc Programming Specifications

# Context Free Grammar:

## Grammar:

- A grammar is indicated with set of statements called production P and production may contains symbols called variables (non-terminals) and terminals.
- It is analytically defined as  $G = \{V, T, P, S\}$ 
  - where V=non empty finite set of variables / non-terminals (represented with upper case letters letters)
  - T=non empty finite set of terminals (represented with lower case letters letters)
  - P=finite set of productions i.e.,  $P: \alpha \rightarrow \beta$
  - S=special symbol called as start symbol

## Linear Grammar:

- A grammar with at most one variable at the right side of the production

## Right Linear Grammar (RLG):

- A grammar  $G = \{V, T, P, S\}$  is said to be RLG if all the productions are in the form of

$$A \rightarrow CB, A \rightarrow aA, A \rightarrow bCB, B \rightarrow a$$

## Left Linear Grammar (LLG):

- The leftmost symbols in the right hand side of the production are variables then such type of grammar is called as left linear grammar i.e.,  $A \rightarrow Sa, A \rightarrow CBb, B \rightarrow a$

# Context Free Grammar (Cont...):

## Regular Grammar:

- A regular grammar is a grammar that may be either RLG or LLG

## Context Free Grammar (CFG):

- A grammar  $G = \{V, T, P, S\}$ 
  - where  $V$  = non empty finite set of variables / non-terminals (represented with upper case letters)
  - $T$  = non empty finite set of terminals (represented with lower case letters)
  - $P$  = finite set of productions i.e.,  $P: \alpha \rightarrow \beta$  where  $\alpha \in V$  and  $\beta \in (V \cup T)^*$
  - $S$  = special symbol called as start symbol

# Derivation Tree:

## Sentential Form:

- Let  $W \in L(G)$  then the sequence  $S \square W_1 \square W_2 \dots \square W$  is the derivation of the sentence. The strings  $W_1, W_2, \dots, W$  containing variables and terminals are called sentential form of the derivation **(or)**
- Let  $G = \{V, T, P, S\}$  be a CFG and  $\alpha \in (V \cup T)^*$  if  $S \square \alpha$  then we say  $\alpha$  is a sentential form

## Derivation:

- The process of derivation always from the start symbol
- It is a process of applying sequence of production rules in order to derive a string of the corresponding language. There are 2 types of derivation trees.

## Parse Tree / Derivation Tree:

- It is a pictorial / diagrammatical representation of derivation process **(or)**
- It is an ordered tree in which nodes are labelled with left hand side of the production and the children of the nodes represents corresponding right hand side of the production
- More formally,  $G = \{V, T, P, S\}$  be a CFG. A tree is a derivation tree or parse tree for  $G$  if it satisfies
  1. Every node / vertex has a label which is variable / terminal /  $\epsilon$
  2. The label of a root is a start symbol
  3. The label of internal nodes are non-terminal / variable (capital alphabets)
  4. The label of leaf node are a terminal (small alphabets, special symbols and operators)

# Ambiguous Grammar:

## Right Most Derivation Tree (RMD):

- A derivation tree is said to be RMD if we replace right most variable in the sentential form of each step

## Left Most Derivation Tree (LMD):

- A derivation tree is said to be LMD if we replace left most variable in the sentential form of each step

## Yield of the tree:

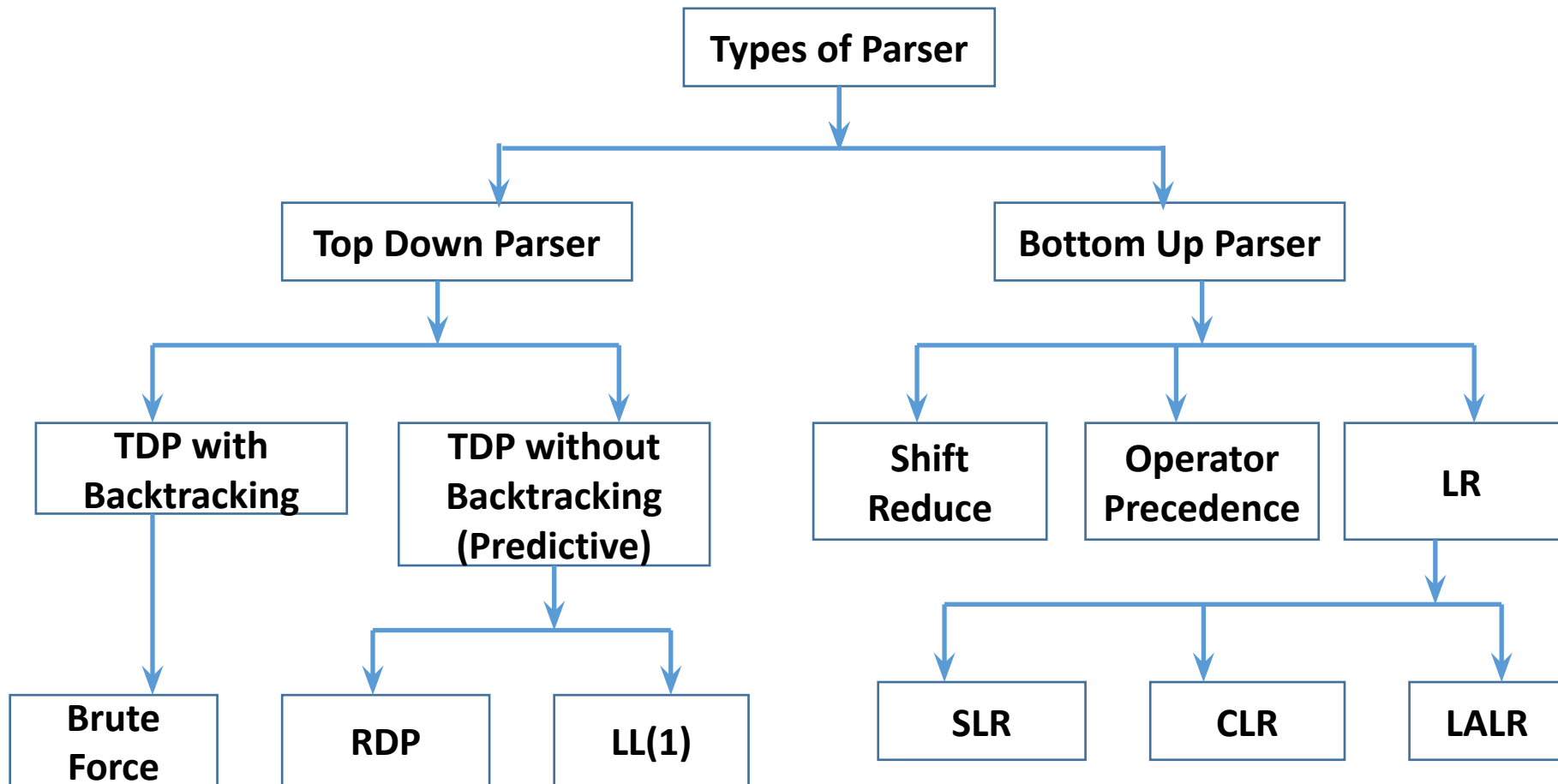
- The connection of leaf nodes from left to right

## Ambiguous Grammar:

- A CFG is said to be ambiguous if there exists some  $W \in L(G)$  which has at least two derivation trees **(or)**
- If there exists two or more LMD's / RMD's that derive a string from the given grammar then such type of grammar is called as ambiguous grammar

# Parsing Techniques:

- The way the production rules are implemented (derivation) divides parsing into two types
  - Top-down parsing
  - Bottom-up parsing



# Parsing Techniques (Cont...):

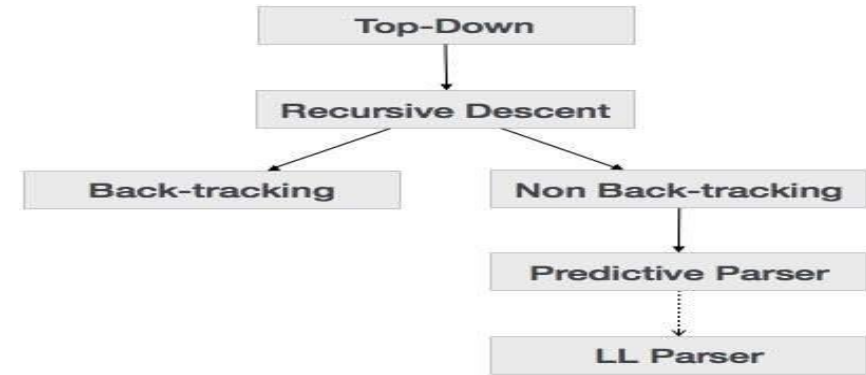
## Top-down Parsing:

- When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing

Start symbol (Starting non terminal symbol of grammar )



Bottom (Leaf Nodes) □ Input String



## Types of Top-down Parsing Techniques:

- **Recursive descent parsing:** It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input.
- **Backtracking:** If one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production
- **LL parsing:** LL parser accepts LL grammar and it is a subset of CFG. It can be implemented by means of both recursive descent and table driven algorithms

## Example:

$E \rightarrow TE'$        $E' \rightarrow +TE' \mid \epsilon$        $T \rightarrow FT'$        $T' \rightarrow *FT' \mid \epsilon$        $F \rightarrow (E) \mid id$



# Parsing Techniques (Cont...):

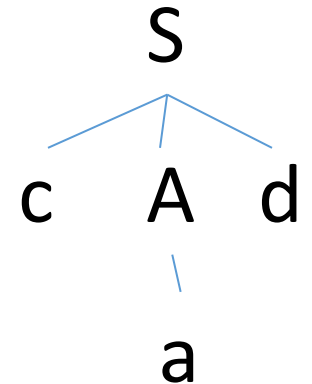
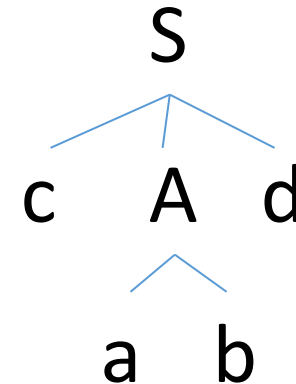
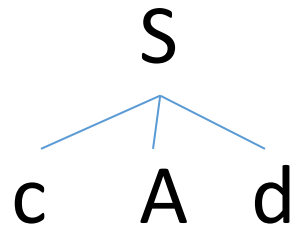
## Brute-Force method

- General recursive descent may require backtracking
- In general form it can't choose an production easily
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Top-down parsers cant be used for left-recursive grammars

**Ex:**

$S \rightarrow cAd$   
 $A \rightarrow ab \mid a$

Input: cad



## **Drawbacks:**

1. Inefficient i.e., more time taken
2. Backtracking
3. More complex to design

# Parsing Techniques (Cont...):

## Bottom-up Parsing:

□ Bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol

Start symbol (Starting non terminal symbol of grammar)

Bottom (Leaf Nodes) □ Input String

## Example:

$S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

Input string is “abcde”

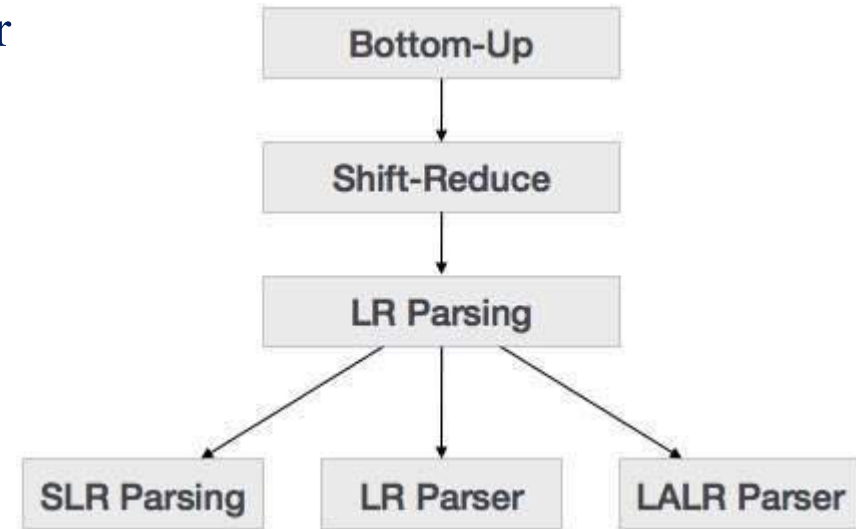
abcde ( $A \rightarrow b$ ) (Handle is a substring which matches with right side of the production)

aAbcde ( $A \rightarrow Abc$ ) (Reduction is the process to replace the RHS of production with LHS variable)

aAde ( $B \rightarrow d$ )

aABe ( $S \rightarrow aABe$ )

This derivation is nothing but reverse of RMD i.e.,  $S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abcde$



# Recursive descent Parser:

- We have to write the recursive procedure for each and every non-terminal that is available in the given grammar

## Steps for construction:

- 1) If the input symbol is non-terminal then call corresponding procedure of a non-terminal
- 2) If the input symbol is terminal then compare this symbol with input string. If both are matching then we increment the input pointer
- 3) If non-terminal symbol produces more than one production then all the production rules code should be written in the corresponding function / procedure
- 4) No need to define any main function or any variables. If we define a main function then we have to call the start symbol function from the main function

## Example:

1.  $E \rightarrow iE^1 \quad E^1 \rightarrow +iE^1 / \epsilon$
2.  $E \rightarrow TE^1 \quad E^1 \rightarrow +TE^1 / \epsilon \quad T \rightarrow *FT^1 / \epsilon \quad F \rightarrow (E) / id$

## Drawbacks:

1. They are not as fast as some other methods
2. It is difficult to provide really good error messages
3. They cannot do parses that require arbitrarily long lookaheads

# Elimination of Left Factoring:

- If a grammar contains a production rule in the form of  $A \rightarrow \alpha\beta_1 / \alpha\beta_2 \dots / \gamma_1 / \gamma_2 \dots$  then we can say it contains left factoring
- The top down parsers can't handle the grammar which contains left factoring. So we need eliminate this
- We can eliminate left factoring by replacing with the following productions for  $A \rightarrow \alpha\beta_1 / \alpha\beta_2 \dots / \gamma_1 / \gamma_2 \dots$ 
  - $A \rightarrow \alpha A^1 / \gamma_1 / \gamma_2 \dots$
  - $A^1 \rightarrow \beta_1 / \beta_2 \dots$

## Examples:

1.  $S \rightarrow iETS / iETSeS / a$        $E \rightarrow b$
2.  $A \rightarrow aAB / aA / a$        $B \rightarrow bB / b$
3.  $X \rightarrow X+X / X*X / D$        $D \rightarrow 1 / 2 / 3$
4.  $E \rightarrow T+E / T$        $T \rightarrow int / int*T / (E)$

# Elimination of Left Recursion:

- If a grammar contains a production rule in the form of  $A \rightarrow A\alpha / \beta$  (The left most symbol of the RHS of production is equal to the LHS non-terminal) then we can say it contains left recursion
- The top down parsers can't handle the grammar which contains left recursion. So we need eliminate this
- We can eliminate the left recursion by replacing with the following productions for  $A \rightarrow A\alpha / \beta$ 
  - $A \rightarrow \beta A^1$
  - $A^1 \rightarrow \alpha A^1 / \epsilon$
- If a grammar contains a production rule in the form of  $A \rightarrow A\alpha_1 / A\alpha_2 \dots / \beta_1 / \beta_2 \dots$  then replacing
  - $A \rightarrow \beta_1 A^1 / \beta_2 A^1 \dots$
  - $A^1 \rightarrow \alpha_1 A^1 / \alpha_2 A^1 \dots / \epsilon$

## Examples:

1.  $E \rightarrow E+T / T$        $T \rightarrow T * F / F$        $F \rightarrow (E) / id$
2.  $S \rightarrow S0S1S / 01$
3.  $S \rightarrow (L) / x$        $L \rightarrow L,S / S$
4.  $expr \rightarrow expr+expr / expr*expr / id$
5.  $S \rightarrow Sx / SSb / xS / a$
6.  $S \rightarrow Aa / b$        $A \rightarrow AC / Sd / f$

# First and Follow:

## First:

1. If  $A \rightarrow A\alpha$  where  $\alpha \in (V \cup T)^*$  then  $\text{First}(A) = \{a\}$
2. If  $A \rightarrow \epsilon$  then  $\text{First}(A) = \{\epsilon\}$
3. If  $A \rightarrow BC$  then  $\text{First}(A) = \text{First}(B)$  if  $\text{First}(B)$  doesn't contain  $\epsilon$   
 $\text{First}(A) = \text{First}(B) \cup \text{First}(C)$  if  $\text{First}(B)$  contain  $\epsilon$

## Follow:

1. If 'S' is a start symbol then  $\text{Follow}(S) = \{\$ \}$
2. If  $A \rightarrow \alpha B \beta$  then  $\text{Follow}(B) = \text{First}(\beta)$  if  $\text{First}(\beta)$  doesn't contain  $\epsilon$
3. If  $A \rightarrow \alpha B \beta$  then  $\text{Follow}(B) = \text{Follow}(A)$  if  $\text{First}(\beta)$  contain  $\epsilon$
4. If  $A \rightarrow \alpha B$  then  $\text{Follow}(B) = \text{Follow}(A)$
5. If  $A \rightarrow \alpha Ba$  then  $\text{Follow}(B) = \{a\}$

## Examples:

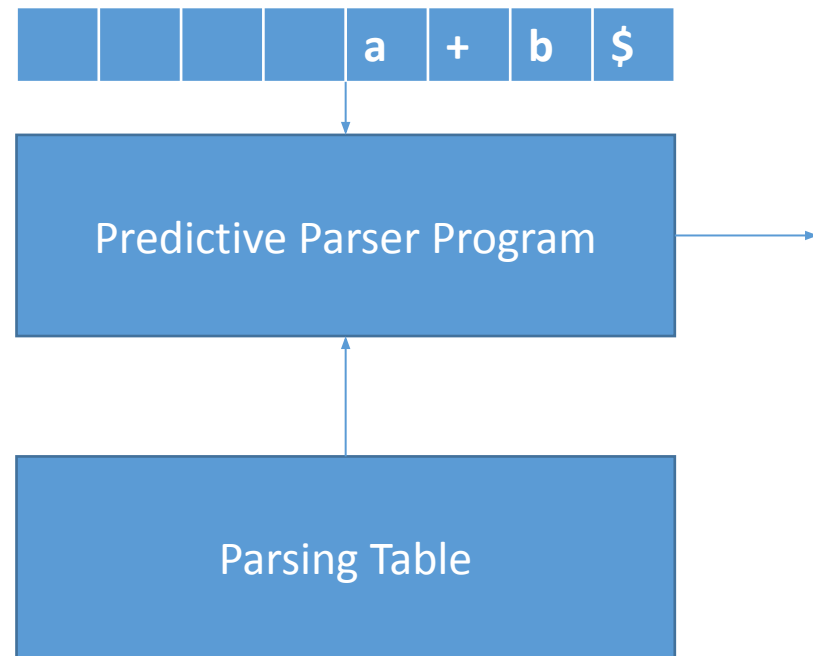
1.  $E \rightarrow TE^1 E^1 \mid + TE^1 / \epsilon$      $T \rightarrow FT^1$      $T^1 \rightarrow * FT^1 / \epsilon$      $F \rightarrow (E) / \text{id}$
2.  $S \rightarrow ABCDE$      $A \rightarrow a / \epsilon$      $B \rightarrow b / \epsilon$      $C \rightarrow c$      $D \rightarrow d / \epsilon$      $E \rightarrow e / \epsilon$
3.  $S \rightarrow Bb / Cd$      $B \rightarrow aB / \epsilon$      $C \rightarrow cC / \epsilon$
4.  $S \rightarrow ACB / ChB / Ba$      $A \rightarrow da / BC$      $B \rightarrow g / \epsilon$      $C \rightarrow h / \epsilon$

# LL(1) / Predictive / Non Recursive Descent Parser:

- LL parser accepts LL grammar and it is a subset of CFG i.e.,  $LL \subseteq CFG$
- It can be implemented by means of both recursive descent and table driven algorithms
- First L represents reading / scanning the input from left to right
- Second L represents use left most derivation (LMD)
- 1 represents read / scan only one symbol at a time from input
- If a grammar is not LL(1) then the grammar is not in LL(k) for any given k

## Model of LL(1):

i / p buffer



# LL(1) / Predictive / Non Recursive Descent Parser:

## Input Buffer

- Used to store the input string which is to be parsed and the last symbol must be \$

## Stack

- Used to insert/delete symbols from stack based on the actions performed by user i.e., push() & pop()
- Initially the top of the stack is \$ after that insert the starting symbol as the top of the stack

## Parsing Table

- Use 2-D array to store the symbols (variables and terminals) in the input string & is represented with  $M[A, a]$  Where M is a 2-D matrix, 'A' (rows) represents to store the variables and 'a' (columns) represents to store the terminals

## Output Buffer

- Used to store the output i.e., actions which is performed by the user

## Steps to Construct a LL(1) Parser:

1. Elimination of left recursion
2. Elimination of left factoring
3. Calculate First and Follow of all variables
4. Construct the parsing table
5. Check whether the given input string is accepted or not by the parser

Ex:  $S \rightarrow (L) / a$      $L \rightarrow L, S / S$



# Shift Reduce Parser:

□ It mainly uses 2 data structures

□ Stack (stores the symbols of the grammar & initially the bottom of the stack is \$)

□ Input Buffer (stores the symbols of the input to be parsed & end of the input symbol is \$)

## Actions of shift reduce parser:

1) Shift    2) Reduce    3) Action    4) Error

□ Shift-reduce parsing uses two unique steps for bottom-up parsing

□ These steps are known as shift-step and reduce-step

**1. Shift step:** It refers to the input symbol that is pushed into the stack

**2. Reduce step:** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

## Example:

$E \rightarrow E+T / T$        $T \rightarrow T * F / T$        $F \rightarrow (E) / id$     i/p: id\*id

$S \rightarrow (L) / a$        $L \rightarrow L, S / S$       i/p: (a,(a,a))

# Handle & Handle Pruning:

## Handle:

- It is a substring which matches with the RHS of the production
- If handle matches with RHS of the production then it is replaced with the corresponding LHS of the production i.e., non-terminal

## Handle Pruning:

The RMD in reverse order is obtained by this i.e., bottom up parser produces the RMD in reverse order and this process is known as handle pruning

## Example:

$S \rightarrow aABe$      $A \rightarrow Abc / b$      $B \rightarrow d$

i/p: abbcde

Handles during parsing of abbcde

$E \rightarrow E+T / T$      $T \rightarrow T * F / F$      $F \rightarrow (E) / id$

i/p: id \* id

Right Sentential Form	Handle	Reducing Productions
abbcde	b	$A \rightarrow b$
a <b>A</b> bcde	Abc	$A \rightarrow Abc$
aA <b>d</b> e	d	$B \rightarrow d$
a <b>A</b> Be	aABe	$S \rightarrow aABe$
S		

# LR(0) Parser:

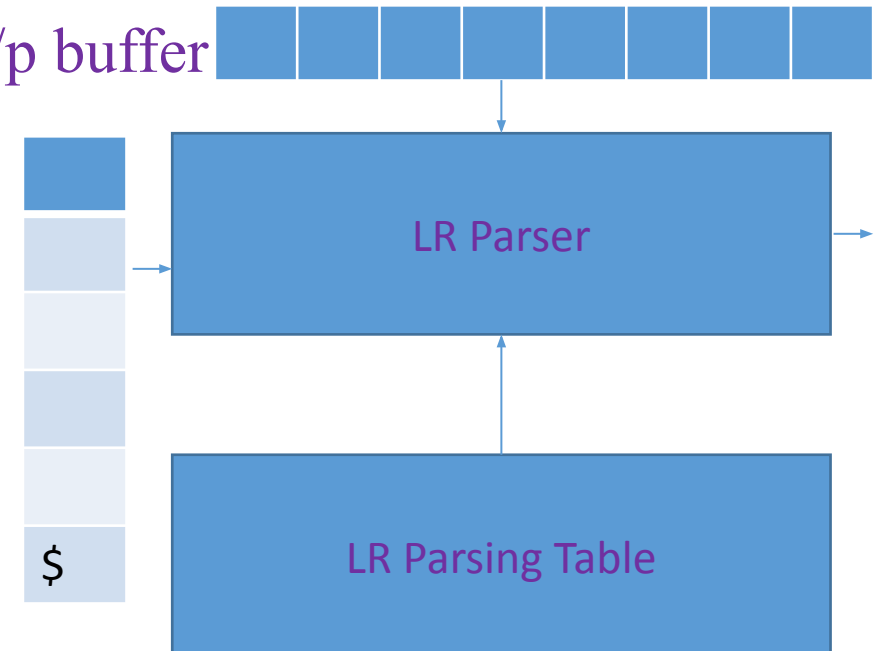
- It is a non recursive shift reduce bottom up parser and also known as LR(k) where L is reading / scanning the input from left to right, R is RMD in reverse for construction of the parser and k is look ahead symbol to make decision
- LR parser algorithm takes input as input buffer, stack and LR parsing table
- To construct LR(0), SLR(1) parsing tables, we use canonical collection of LR(0) items
- To construct LALR(1), CLR(1) parsing tables, we use canonical collection of LR(1) items
- Structure of all parsers are same except parsing table

## Steps required to construct parser

- For the given i/p string, write CFG
- Check ambiguity of the grammar
- Add augment production to the grammar
- Create canonical collection of LR(0) items
- Construct LR(0) parsing table

stack

i/p



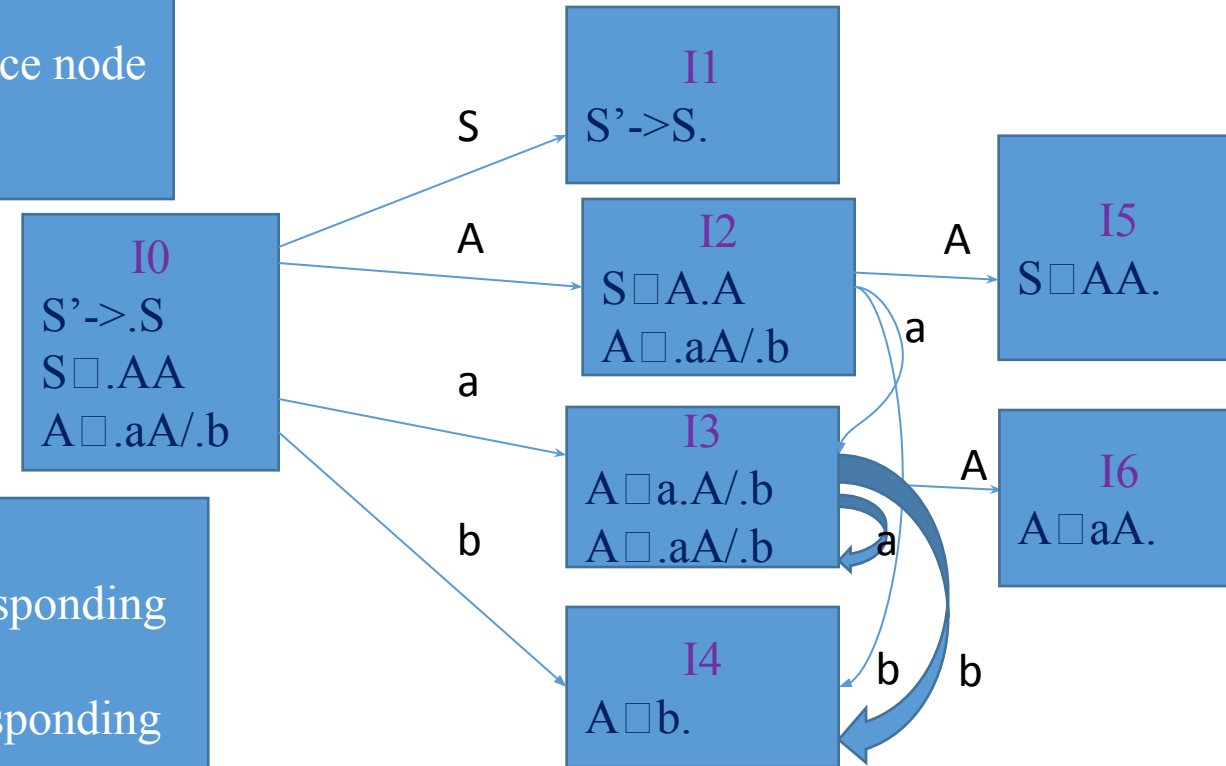
### Create LR(0) items:

1. LR(0) item is a production in G with .(dot) at some position on RHS
2. LR(0) item is useful to indicate that how much of the i/p has been scanned up to a given point in the process of parsing
3. If . at the end of production in LR(0) item, we place the reduce node in the entire row

**Ex:**  $S \rightarrow AA$   
 $A \rightarrow aA$   
 $A \rightarrow b$

### **Augment Grammar**

$S' \rightarrow S$   
 $S \rightarrow AA$   
 $A \rightarrow aA/b$



### Construct LR(0) parsing table:

1. If a state is going to some other state on terminal, it is corresponding to shift move
2. If a state is going to some other state on variable, it is corresponding to goto move
3. If a state contains the final item, write the reduce node in the particular row

# LR(0) Parser (Cont...):

## LR(0) table

States	Action / Shift (Terminals)			Goto (Variables)	
	a	b	\$	S	A
I0	S3	S4		1	2
I1			accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

## LR(0) parsing / parsing i/p string

Step no.	Parsing Stack	i/p	Action
1	\$0 (empty)	aabb\$	shift3
2	\$0a3	abb\$	shift3
3	\$0a3a3	bb\$	shift4
4	\$0a3a3b4	b\$	reduce r3
5	\$0a3a3A6	b\$	reduce r2
6	\$0a3A6	b\$	reduce r2
7	\$0A2	b\$	shift4
8	\$0A2b4	\$	reduce r3
9	\$0A2A5	\$	reduce r1
10	\$0S1	\$	accept

# SLR(1) Parser:

- Steps required to construct parser
  - For the given i/p string, write CFG
  - Check ambiguity of the grammar
  - Add augment production to the grammar
  - Create canonical collection of LR(0) items
  - Construct SLR(1) parsing table

# CLR(1) / LR(1) Parser:

## □ Steps required to construct parser

- While constructing a parse table for this, we use LR(1) items
- LR(1) items = LR(0) items + look ahead symbols
- For final items, we need to apply reduction on the look ahead symbols of the corresponding production
- Number of states might be greater or equal to the LR(0) or SLR(1) i.e.,  $CLR(1) \geq LR(0) = SLR(1)$
- Number of blanks in the parsing table is high
- Error detection capability becomes more because of more no. of blanks

## □ Example:

1.  $S \rightarrow AA$       $A \rightarrow Aa / d$
2.  $A \rightarrow (A)$       $A \rightarrow a$

# LALR(1) Parser:

## □ Steps required to construct parser

- While constructing a parse table for this, we use LR(1) items
- $\text{LR}(1) \text{ items} = \text{LR}(0) \text{ items} + \text{look ahead symbols}$
- For final items, we need to apply reduction on the look ahead symbols of the corresponding production
- Number of states are equal to the LR(0) and SLR(1) i.e.,  $\text{LALR}(1) = \text{LR}(0) = \text{SLR}(1)$
- Number of blanks in the parsing table is lesser than CLR(1)
- Error detection capability becomes less compared from CLR(1)

## □ Example:

1.  $S \rightarrow AA$       $A \rightarrow Aa / d$
2.  $A \rightarrow (A)$       $A \rightarrow a$



# YACC Tool:

- Yet another compiler compiler
- Lex □ lexical analyzer generator
- YACC □ parser generator
- Lex tool is an automated tool which is used to help to lexical analyzer to generate tokens

# Unit – 3

## Syntax Directed Translation

1. Definitions
2. Construction of Syntax Trees
3. S-attributed and L-attributed grammars
4. Intermediate code generation
5. Abstract syntax tree
6. Translation of simple statements and control flow statements

# Syntax Directed Definition:

- It is a CFG together with semantic rules (CFG + semantic rules)
- Attributes are associated with grammar symbols and semantic (it provides a meaning to the corresponding production) rules are associated with production rules
- Attributes may be numbers, strings, references, datatypes, memory location etc...
- If 'X' is a symbol and 'a' is one of its attribute then X.a denotes value at node 'X'
- Every symbol must contain an attribute
- Every production must contain semantic rule

Ex:     productions     semantic rules (informal notation)

$E \rightarrow E + T \quad E.val \rightarrow E.val + T.val$

$E \rightarrow T \quad E.val \rightarrow T.val$

## Types of attributes:

1. Synthesized attribute
2. Inherited attribute

# SDD (Cont...):

## 1. Synthesized attribute:

□ If a node takes value from its children

Ex:  $A \sqsupset BCD$  where A is a parent node and B, C, D are children nodes

$A.s \sqsupset B.s$

$A.s \sqsupset C.s$

$A.s \sqsupset D.s$

} parent node A is taking the values from its children nodes B, C, D

## 2. Inherited attribute:

□ If a node takes value from its parent / siblings

Ex:  $A \sqsupset BCD$  where A is a parent node and B, C, D are children nodes

$C.i \sqsupset A.i$

$C.i \sqsupset B.i$

$C.i \sqsupset D.i$

} children node C is taking the values from its parent A / sibling node (B,D)

# Syntax Directed Definition (Cont...):

## Types of SDD:

1. S-Attributed SDD / S-Attributed definitions / S-Attributed grammar
2. L-Attributed SDD / L-Attributed definitions / L-Attributed grammar

S.No.	S-Attributed SDD / definitions / grammar	L-Attributed SDD / definitions / grammar
1	It uses only synthesized attributes <b>Ex:</b> $A \rightarrow BCD$ $A.s \rightarrow B.s \quad A.s \rightarrow C.s \quad A.s \rightarrow D.s$	It uses both synthesized and inherited attributes but each inherited attribute is restricted to inherits from its parent or left sibling <b>Ex:</b> $A \rightarrow BCD$ $C.i \rightarrow A.i, \quad C.i \rightarrow B.i$ but $C.i \rightarrow D.i$ (*)
2	Semantic actions are always placed at right end of the productions. So it is also called as postfix SDD	Semantic actions are placed anywhere on RHS
3	Attributes are evaluated with bottom up parsing	Attributes are evaluated by traversing parse tree depth first left to right order

**SDD of a simple desk calculator / SDD for evaluation of an expression / Annotated parse tree for  $(3*5+4)n$**

**Example:**  $3 * 5 + 4n$

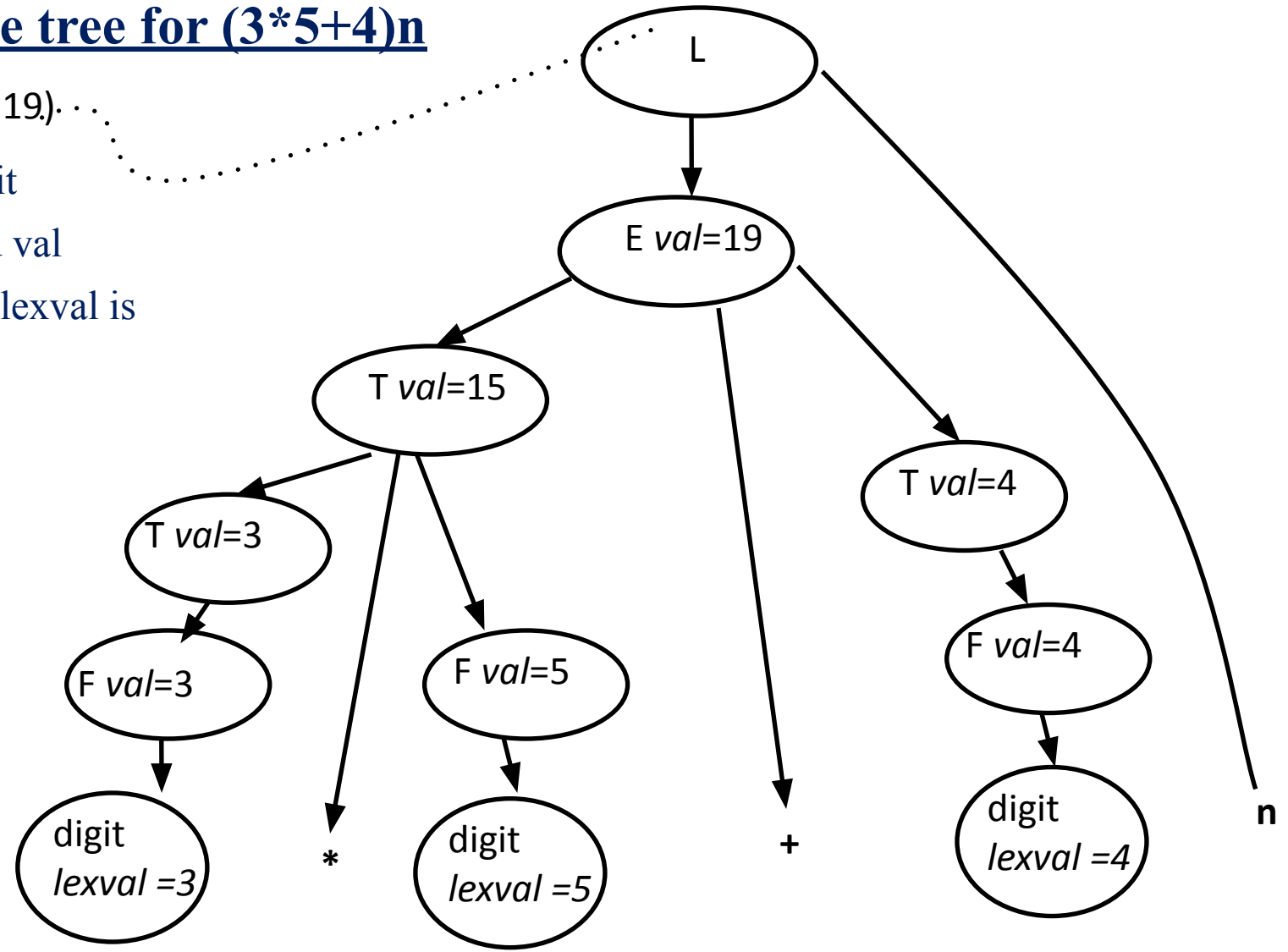
Print(19). . .

Grammar symbols: L, E, T, F, n , + , \* ,( , ) , digit

Non-terminals L, E, T, F have an attribute called val

Terminal digit has an attribute called lexval and lexval is provided by lexical analyzer

<u>Production</u>	<u>Semantic Rule</u>
$L \rightarrow E \text{ n}$	$L.val = E_1.val \quad \text{print}(E.val)$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

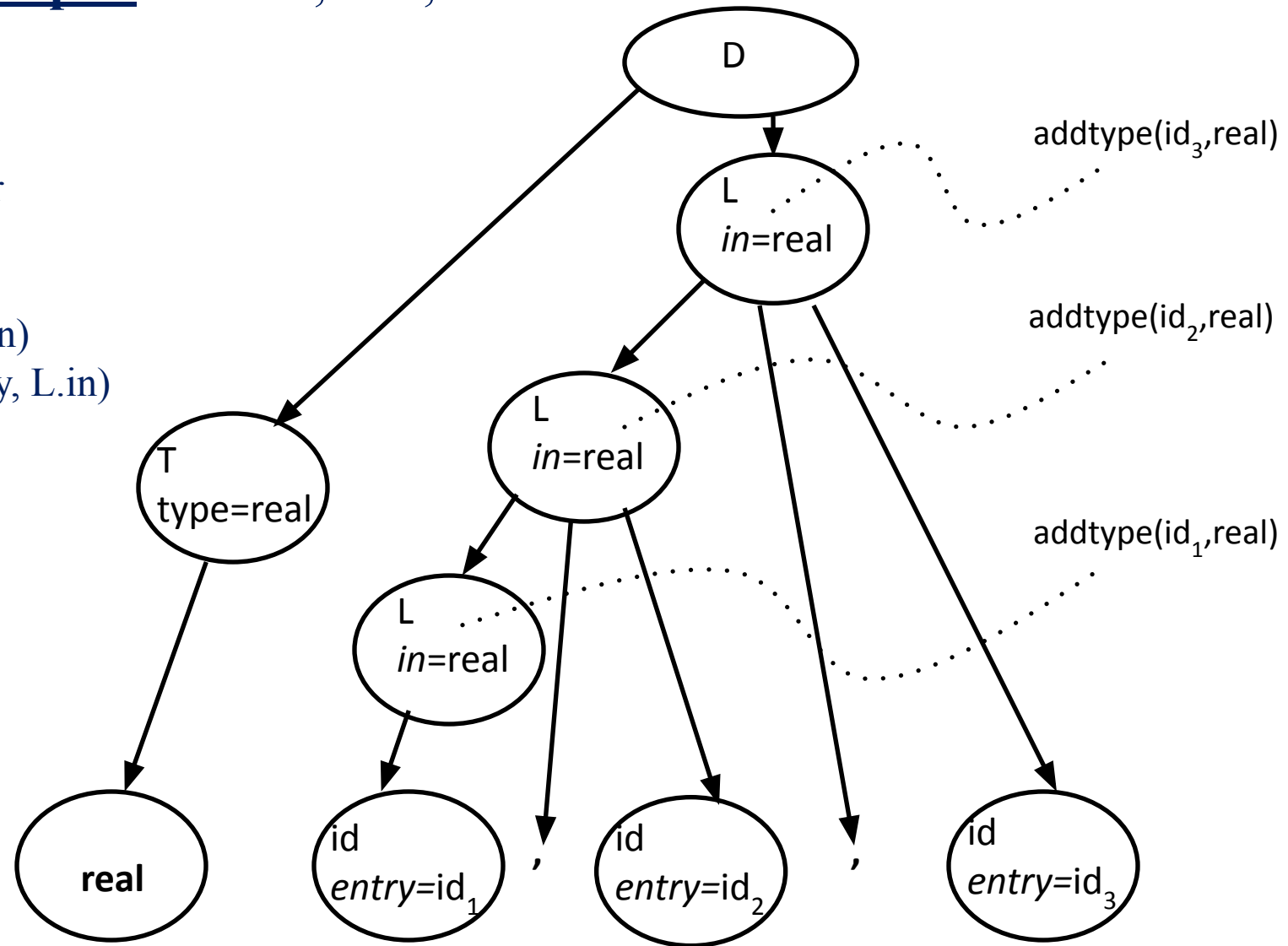


**Annotated parse tree** is a parse tree which contains values at each node

### Annotated parse tree for $3*5+4n$

**Example:** real id1, id2 , id3


<u>Production</u>	<u>Semantic Rule</u>
$D \rightarrow T L$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$
	$addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$



### Annotated pars tree for real $id_1, id_2, id_3$

# Syntax Directed Translation:

- It is a CFG together with semantic rules (CFG + semantic rules)
- Semantic actions must be written in {} and appear anywhere on the RHS of the production
- Semantic actions should be performed based on the corresponding production in bottom up parsing
- Semantic actions should be performed when it appears in top down parsing and semantic action is also considered as children of LHS of the production

Ex:      $E \rightarrow E + T \quad \{\text{printf}("+");\}$   semantic action

$E \rightarrow T \quad \{\}$

$T \rightarrow T * F \quad \{\text{printf}("*");\}$

$T \rightarrow F \quad \{\}$

$F \rightarrow \text{num} \quad \{\text{printf}(\text{num.val});\}$

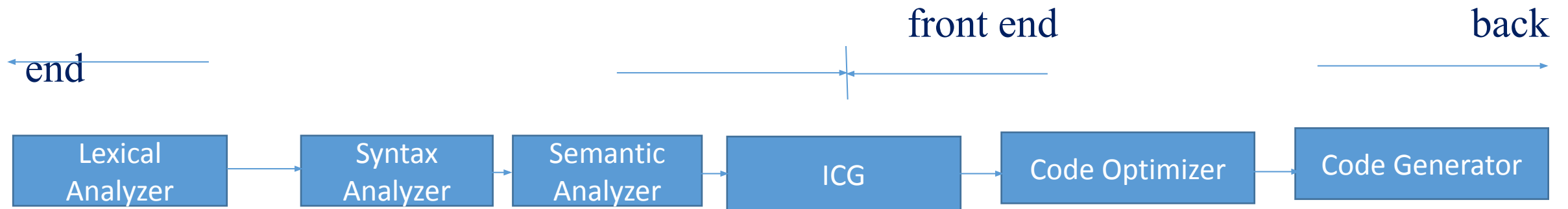


# Intermediate Code Generation:

- It is used to translate the source code into machine code
- It lies between high level language and low level language
- It receives the input from its predecessor phases and input is in the form of annotated syntax tree



- If the compiler directly translates source code into machine code without using intermediate code then a full native compiler is needed for each new machine



- It can be represented in two ways
  - **High Level Intermediate Code:** It can be represented as source code
  - **Low Level Intermediate Code:** It is close to the target machine code

# Intermediate Code Generation (Cont...):

## Forms of intermediate code:

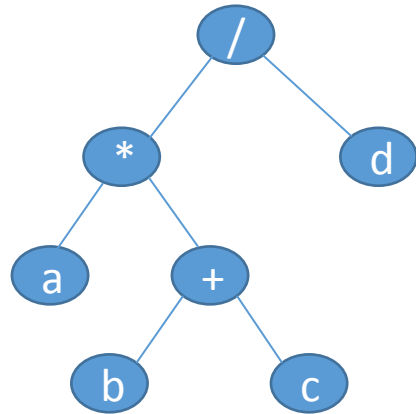
1. Syntax Tree/Abstract Syntax Tree   2. Polish/Postfix Notation   3. Three Address Code

### 1. Syntax Tree / Abstract Syntax Tree

□ Each internal node represents an operator

□ Leaf nodes represents an operands

**Ex:**  $a*(b+c)/d$



### 2. Polish / Suffix / Postfix Notation

□ In this, the operator appears only after operands

**Ex:**  $a+(b*c)$    □    $abc*+$     $(a+b)*c$    □    $ab+c*$     $(a-b)*(c/d)$    □    $ab-cd/*$

# Intermediate Code Generation (Cont...):

## 3. Three Address Code

- Each instruction should contain at most 3 addresses
- Each instruction should contain at most 1 operator in the RHS
- It is represented in 3 ways 1. Quadruple 2. Triple 3. Indirect Triple

**Ex:**  $x+y*z$  □  $t1 = y*z$   $t2 = x+t1$

**(i) Quadruple** □ It contains 4 fields i.e., operator, input1/source1/argument1, input2/source2/argument2, output/result/ destination **Ex:**  $a = b * -c + b * -c$

$t1 = -c$   $t2 = b*t1$   $t3 = -c$   $t4 = b*t3$   $t5 = t2 + t4$

**Drawback:** Large amount of memory is required for storing all the temporary variables

Represent 3 address code in the form of **Quadruple**

address	Operator	Source1	Source2	Result
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5

# Intermediate Code Generation (Cont...):

## 3. Three Address Code

**(ii) Triple** □ It contains 3 fields operator, source1, source2

**Advantage:** Temporary variables are not required so with less amount of memory we can execute the instructions

Represent 3 address code in the form of **Triple**

address	Operator	Source1	Source2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)

Represent 3 address code in the form of **Indirect Triple**

Pointer	Triples
100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

**(iii) Indirect Triple** □ This is also same as triple but it requires an extra table which contains a pointer  
This pointer is pointing to the triples and triple information is available in triple  
table

# Intermediate Code Generation (Cont...):

## Three Address Code instruction forms

1. Assignment instructions are of the form  $x = y \text{ op } z$  where op is a binary operator  
□ **Ex:**  $x = y + z$  ,  $x = y > 3$  ,  $x = y \&\& z$
2. Assignment instructions are of the form  $x = \text{op } z$  where op is a unary operator  
□ **Ex:**  $x = -y$  ,  $x = ++y$  (increment, decrement, logical not etc...)
3. Copy instructions are of the form  $x = y$  where value of y is assigned to x  
□ **Ex:**  $y = 10$  ,  $x = y$
4. Unconditional jump goto L                      **Ex:** goto 100;      100: statements (to be executed)
5. i. Conditional jumps are of the form if x **relop** y goto L (condition true the L)  
ii. Conditional jumps are of the form if x goto L1 else goto L2 (x is true then L1 else L2)
6. Procedure / function calls and returns are implemented using following program  
□ **Ex:**  $x = y + z$  ,  $x = y > 3$  ,  $x = y \&\& z$
7. Address & pointer assignments are of the form      **Ex:**  $x = \&y$  ,  $x = *y$  ,  $*x = y$
8. Indexed copy instructions are of the form      **Ex:**  $x = y[i]$  ,  $x[i] = y$

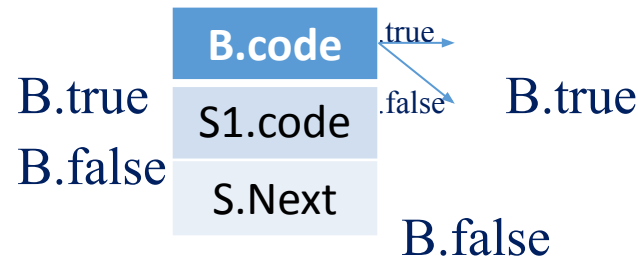
# Intermediate Code generation (Cont...):

Intermediate code for flow of control statements / SDT or SDD of flow of control statements into three address code

## Productions for simple if

$S \rightarrow \text{if}(B) \text{ then } S1$

## Code for simple if



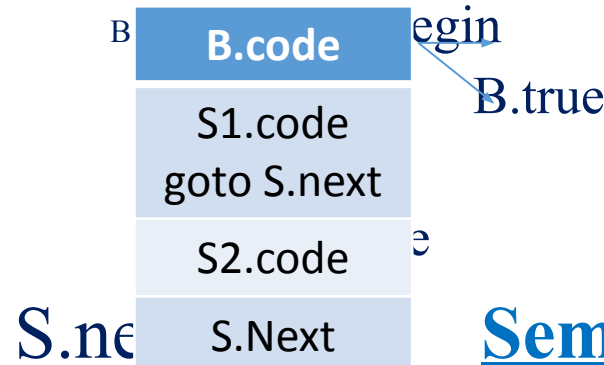
## Semantic rule

$B.true = \text{newlabel}()$   
 $S1.next = S.next$   
 $B.false = S.next$   
 $S.code = B.code \parallel \text{label}(B.true) \parallel B.false = \text{newlabel}()$   
 $\quad S1.code$   
 $S.code = B.code \parallel \text{label}(B.true) \parallel S1.code \parallel$

## Productions for if else

$S \rightarrow \text{if}(B) \text{ then } S1 \text{ else } S2$

## Code for if else



## Semantic rule

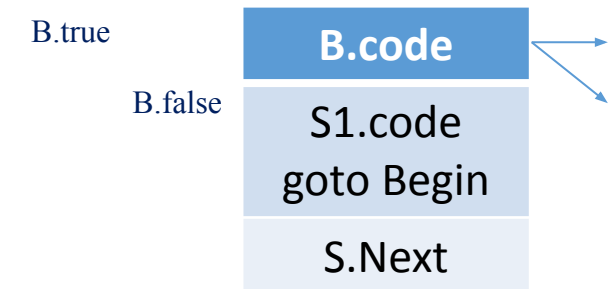
$B.true = \text{newlabel}()$   
 $S1.next = S.next$   
 $S2.next = S.next$   
 $S.code = B.code \parallel \text{label}(B.true) \parallel S1.code \parallel$   
 $\quad \text{gen('goto' S.next)} \parallel \text{label}(B.false) \parallel S2.code$

## Productions for while

$S \rightarrow \text{while}(B) \text{ then } S1$

## Semantic rule

$Begin = \text{newlabel}()$   
 $B.true = \text{newlabel}()$   
 $S1.next = Begin$   
 $B.false = S.next$   
 $S.code = \text{label}(Begin) \parallel B.code \parallel \text{label}(B.true)$   
 $\quad \parallel S1.code \parallel \text{gen('goto' Begin)}$



# Intermediate Code generation (Cont...):

## Intermediate code for switch statement / Translation of switch statement

### Switch statement syntax   Translation of switch statement

switch(E)	code to evaluate E into t	test:   if t = v <sub>1</sub> goto L1
{	goto test	if t = v <sub>2</sub> goto L2
case v <sub>1</sub> : s1	L <sub>1</sub> : code for S1	.
case v <sub>2</sub> : s1	goto next	.
.	L <sub>2</sub> : code for S2	.
.	goto next	if t = v <sub>n-1</sub> goto Ln-1
.	.	goto Ln
case v <sub>n-1</sub> : sn-1	.	next:
default: sn	L <sub>n-1</sub> : code for Sn-1	
}	goto next	
	L <sub>n</sub> : code for Sn	
	goto next	

# Intermediate Code generation (Cont...):

## Intermediate code for procedures

$D \sqsubseteq \text{define } T \text{ id } (F) \{S\}$

$F \sqsubseteq \epsilon / T \text{ id } , S$

$S \sqsubseteq \text{return } E;$

$E \sqsubseteq \text{id } (A);$

$A \sqsubseteq \epsilon / E , A$



# Intermediate Code generation (Cont...):

## Directed Acyclic Graph (DAG) or DAG Representation of a basic block

□ DAG represents the structure of a basic block

1. Internal node represents operator
2. Leaf node represents identifiers, constants
3. Internal node also represents result of expressions

### □ Applications of DAG

1. Determining the common sub expression
2. Determining which names are used inside the block & computed outside the block
3. Determining which statements of the block could have their computed value outside the block
4. Simplifying the list of quadruples by eliminating common sub expression

### □ Examples

1. Construct DAG for the expression  $a + a*(b-c) + (b-c)*d$
2. Construct DAG for the  $a = b+c$   $b = a-d$   $c = b+c$   $d = a-d$
3. Construct DAG for the following  $a = b+c$   $b = b-d$   $c = c+d$   $e = b+c$
4. Construct DAG for the following  $d = b*c$   $e = a+b$   $b = b*c$   $a = e-d$
5. Construct DAG for the expression  $a = b*-c + b*-c$
6. Construct DAG for the expression  $a = (a*b+c) - (a*b+c)$

# Intermediate Code generation (Cont...):

## Dependency Graph

- Represents the flow of information among the attributes in a parse tree
- Useful for determining the evaluation order for attributes in a parse tree
- Annotated parse tree shows the values of attribute whereas dependency graph determines how those values can be computed

## □ Examples

### productions

1.  $E \rightarrow E+T$

2.  $E \rightarrow T$

3.  $T \rightarrow T * F$

4.  $T \rightarrow F$

5.  $F \rightarrow \text{digit}$

### semantic rules

$E.val \rightarrow E.val + T.val$

$E.val \rightarrow T.val$

$T.val \rightarrow T.val * F.val$

$T.val \rightarrow F.val$

$F.val \rightarrow \text{digit.lexval}$

**Construct dependency graph for the given expression:**

**5+3\*4**

## **Semantic Analysis**

- 1. Semantic Errors  
recognizers**
- 2. Chomsky hierarchy of languages and**
- 3. Type checking**
- 4. Type conversions**
- 5. Equivalence of type expressions**
- 6. Polymorphic functions**
- 7. Overloading of functions and operators**

# Introduction

- Errors are of two types. They are
  1. Compile time
  2. Run time
- Most of the times errors will occurred at front end of the phases of the compiler
  1. Lexical Error
  2. Syntactic Error
  3. Semantic Error

# Introduction

## Lexical Error:

- Errors which occurred in lexical phase is called lexical errors
- It is a sequence of characters that does not match the pattern of any token
- lexical phase error is found during the execution of the program
  - spelling error (identifier / keyword / operator)
  - exceeding length of identifier or numeric constant
  - appearance of illegal characters
  - To remove the character that should be present
  - To replace a character with an incorrect character
  - transposition of characters

**Ex:** void main()

```
{  
    int x=10,y=20; char *a; printf(“%d”,a1); x=4*ab;  
}
```

(number / identifier)

# Introduction

## Syntactical Error:

- Errors which occurred in syntax phase is called syntactic errors
- It occurs during the parsing of input code, and are caused by grammatically incorrect statements
- syntax phase error is found during the execution of the program
  - error in structure
  - missing operators
  - missing semicolons
  - unbalanced parenthesis

while parse tree construction

**Ex:** if (number=100)  
    print(“number is equal to 100”);  
else  
    printf(“number is not equal to 100”);

## Semantic Error:

- Errors which occurred in syntax phase is called syntactic errors
- These are detected at compile time
  - incompatible type of operands
  - undeclared variable
  - not matching of actual argument with formal argument
  - return type mismatch

**Ex:** `int i;      int a="hello";      int main() {return 'c';}`  
`void f(int m)`  
`{`  
`m=t;`  
`}`

**Non initialized var' incompatible**

**return type mismatch**

## **Error Handling:**

- Find errors
- Diagnosis
  - Viable prefix property of parser allows early detection of syntax errors
- Error recovery
  1. Panic mode
  2. Phrase level recovery
  3. Error production
  4. Global correction



# Grammar

Formally, a generative grammar  $G$  is a quad-tuple  $G = (V, T, P, S)$

Where,  $V$  is a finite set of non-terminals

$T$  is a finite set of terminals

$P$  is the finite set of production rules

$S$  is the start symbol

Two main categories of formal grammar are:

- **Generative grammars** – are set of rules for generation of strings in a language
- **Analytic grammars** – are set of rules to determine whether a string is a member of the language

In 1956, *Noam Chomsky* classified the generative grammars into four types known as the Chomsky hierarchy

# Types of Grammars

## Type-3 (Regular Grammar)

- Language which is generated by this grammar is regular language
- This grammar is accepted by finite automata
- Linear grammar (1 non-terminal  $\Rightarrow$  at most 1 non-terminal)
- Types – LLG, MLG, RLG
- Regular grammar is either LLG or RLG but not both
- Production rule is in the form of  $\alpha \Rightarrow \beta$

Where  $\alpha \in V$  and  $\beta \in VT^*$  or  $T^*V$

Example:  $S \Rightarrow aA$

$A \Rightarrow bB$

$B \Rightarrow \epsilon$

## Type-2 (Context Free Grammar)

- Language which is generated by this grammar is CFG
- This grammar is accepted by push down automata (PDA)
- Production rule is in the form of  $\alpha \rightarrow \beta$

Where  $\alpha \in V$  and  $\beta \in (V+T)^*$

**Example:**  $S \rightarrow aBb$

$B \rightarrow bBB$

$B \rightarrow aa$

$B \rightarrow b$

# Type-1 (Context Sensitive Grammar)

- It is also called as non contracting grammars
- Language which is generated by this grammar is CSL
- This grammar is accepted by linear bounded automata (LBA)
- Production rule is in the form of  $\alpha \rightarrow \beta$  such that  $|\alpha| \leq |\beta|$

where  $\alpha, \beta \in (V + T)^+$

i.e., epsilon ( $\epsilon$ ) on the LHS and RHS of productions is not allowed

$S \rightarrow \epsilon$  can be allowed but S must not appear in RHS and the grammar must not satisfy with type-3 or type-2

**Example:**  $S \rightarrow abc \mid aAbc$   
 $Ab \rightarrow bA$   
 $Ac \rightarrow Bbcc$   
 $bB \rightarrow Bb$

## Type-0 (Unrestricted Grammar)

Language which is generated by this grammar is recursively enumerable language (REL)

This grammar is accepted by Turing machine

Production rule is in the form of  $\alpha \rightarrow \beta$

where  $\alpha \in (V \cup T)^+$  and  $\beta \in (V \cup T)^*$

i.e., epsilon ( $\epsilon$ ) on the left-hand side of any productions is not allowed

$S \rightarrow \epsilon$  can be allowed but S must not come in RHS

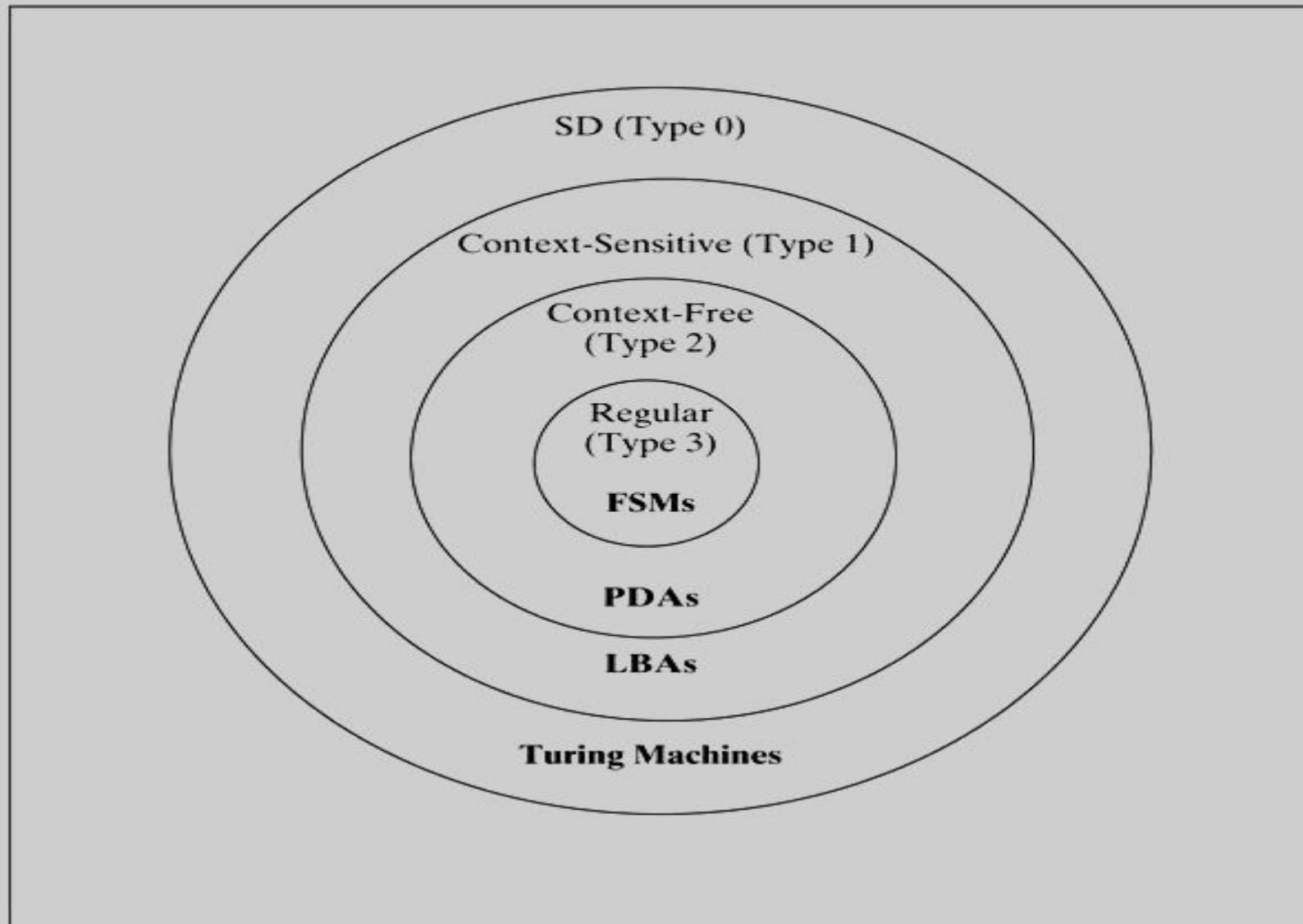
**Example:**  $S \rightarrow aBc \mid \epsilon$   
 $aB \rightarrow cA \mid a$   
 $Ac \rightarrow d$

# Types of Grammars Comparison

**Type-3  $\subseteq$  Type-2  $\subseteq$  Type-1  $\subseteq$  Type-0**

Class	Grammar	Language	Automaton	Restrictions on productions
Type-3	Regular	Regular	FA	$\alpha \rightarrow \beta$ Where $\alpha \in V$ and $\beta \in VT^*$ or $T^*V$
Type-2	Context Free	Context Free	PDA	$\alpha \rightarrow \beta$ Where $\alpha \in V$ and $\beta \in (V+T)^*$
Type-1	Context Sensitive	Context Sensitive	LBA	$\alpha \rightarrow \beta$ where $\alpha, \beta \in (V+T)^+$ and $ \alpha  \leq  \beta $
Type-0	Unrestricted	Recursively Enumerable	TM	$\alpha \rightarrow \beta$ where $\alpha \in (VUT)^+$ and $\beta \in (VUT)^*$

# The Chomsky Hierarchy



# Symbol Table

- **Symbol table** is a data structure used by compiler to keep track of semantics of variable. i.e. symbol table stores the information about scope and binding information about names.
- Symbol table is built in lexical and syntax analysis phases.
- It is used by various phases as follows, **semantic analysis** phase refers symbol table for type conflict issues. **Code generation** refers symbol table to know how much run-time space is allocated? What type of space allocated?



# Use of symbol table

- To achieve compile time efficiency compiler makes use of symbol table
- It associates lexical names with their attributes.
- The items to be stored in symbol table are,
  - Variable name
  - Constants
  - Procedure names
  - Literal constants & strings
  - Compiler generated temporaries
  - Labels in source program

**Contd..**

Compiler uses following types of information from symbol table, i.e. attributes.

- Data type
- Name
- Procedure declarations
- Offset in storage
- In case of structure or record, a pointer to structure table
- For parameters, whether pass by value or reference
- Number and type of arguments passed
- Base address

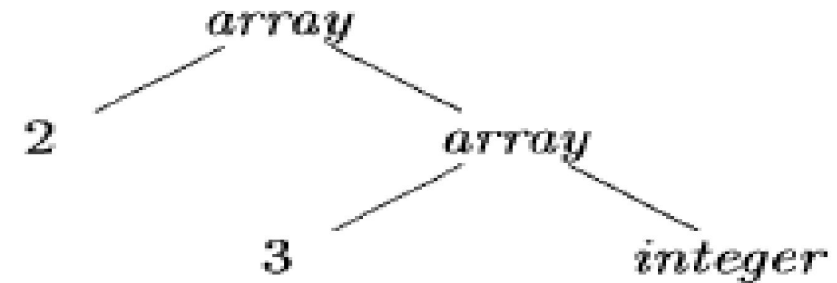
# Type

**Type:** is a property of program constructs such as expressions. It defines a set of values (range of variables) and a set of operations on those values.

- Compiler must check that the source program follows both the syntactic and semantic conventions of the source language.
- Static checks - reported during the compilation phase.
- Dynamic checks - occur during the execution of the program.

# Type Expressions

Example:     `int array[2][3]`  
              `array(2,array(3,integer))`



- A basic type is a type expression
- A type name is a type expression
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named field
- A type expression can be formed by using the type constructor  $\rightarrow$  for function types
- Type expressions may contain variables whose values are type expressions

# Type Equivalence

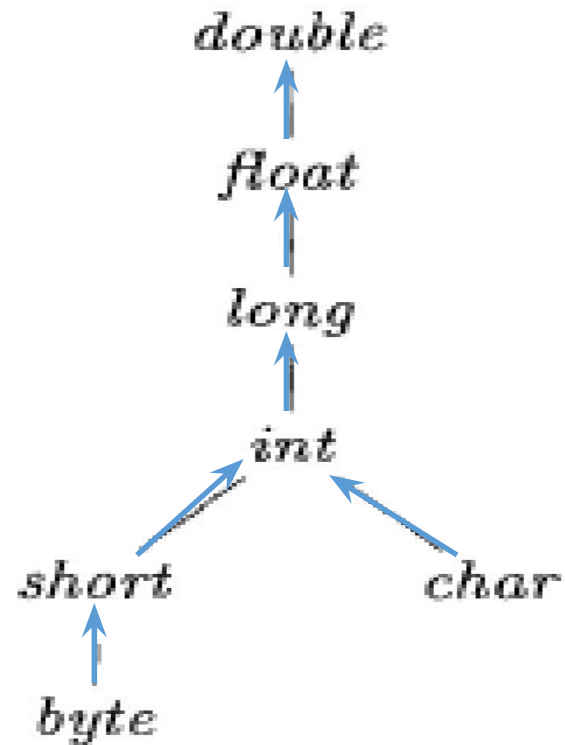
- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

```

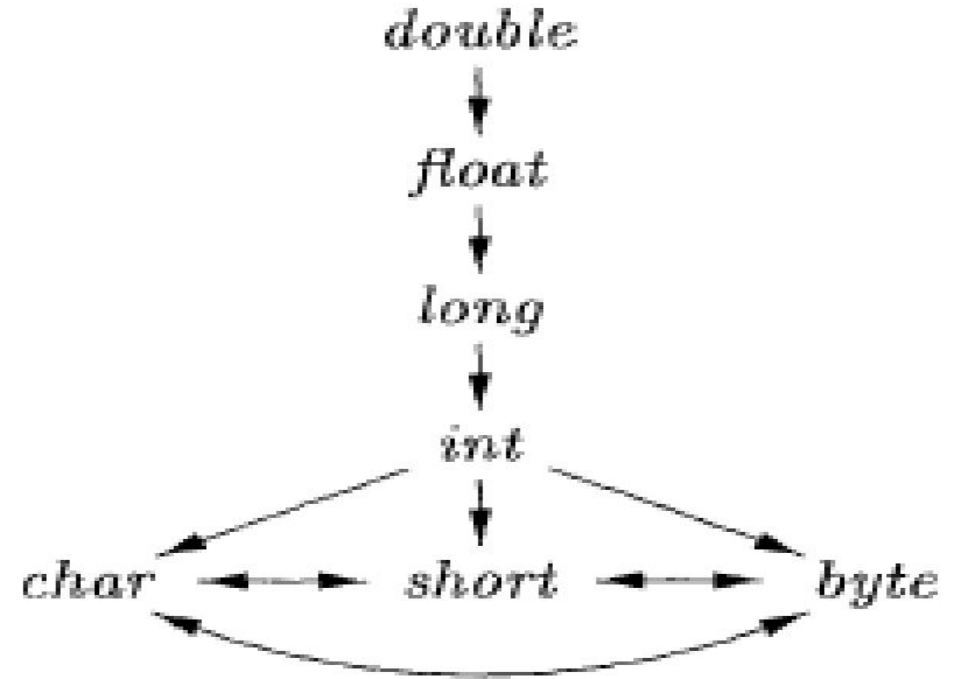
(1) function sequiv(s, t): boolean;
    begin
(2)         if s and t are the same basic type then
(3)             return true
(4)         else if s = array(s1, s2) and t = array(t1, t2) then
(5)             return sequiv(s1, t1) and sequiv(s2, t2)
(6)         else if s = s1 × s2 and t = t1 × t2 then
(7)             return sequiv(s1, t1) and sequiv(s2, t2)
(8)         else if s = pointer(s1) and t = pointer(t1) then
(9)             return sequiv(s1, t1)
(10)        else if s = s1 → s2 and t = t1 → t2 then
(11)            return sequiv(s1, t1) and sequiv(s2, t2)
        else
(12)            return false
    end

```

# Conversions between primitive types



(a) Widening conversions



(b) Narrowing conversions

# Type conversions into expression evaluation

$E \rightarrow E_1 + E_2$     {  $E.type = \max(E_1.type, E_2.type);$   
                           $a_1 = \text{widen}(E_1.addr, E_1.type, E.type);$   
                           $a_2 = \text{widen}(E_2.addr, E_2.type, E.type);$   
                           $E.addr = \text{new Temp}();$   
                           $\text{gen}(E.addr '=' a_1 '+' a_2);$  }



# Type Checking?

- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
- This generally means that all operands in any expression are of appropriate types and number.
- Much of what we do in the semantic analysis phase is type checking.

# Designing a Type Checker

- A language is considered strongly - typed if each and every type error is detected during compilation.
- Type checking can be done compilation, during execution, or divided across both.

When designing a type checker for a compiler, here's the process:

1. Identify the types that are available in the language
2. Identify the language constructs that have types associated with them
3. Identify the semantic rules for the language

# Static type checking

- Static type checking is done at compile-time. The information the type checker needs is obtained via declarations and stored in a master symbol table.
- After this information is collected, the types involved in each operation are checked.

## Cont...

- For example, if `a` and `b` are of type `int` and we assign very large values to them, `a * b` may not be in the acceptable range of `int`'s, or an attempt to compute the ratio between two integers may raise a division by zero. These kinds of type errors usually cannot be detected at compiler time.

# Dynamic type checking

- Dynamic type checking is implemented by including type information for each data location at runtime.
- For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type".
- The execution of any operation begins by first checking these type tags. The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution.

# Implicit type conversion

- Many compilers have built-in functionality for correcting the simplest of type errors.
- **Implicit type conversion, or coercion**, is when a compiler finds a type error and then changes the type of the variable to an appropriate type.
- Ada and Pascal, for example, provide almost no automatic coercions, requiring the programmer to take explicit actions to convert between various numeric types.

# Error Recovery

- Since type checking has the potential for catching errors in programs, it is important to do something when an error is discovered.
- The inclusion of error handling may result in a type system that goes beyond the one needed to specify correct programs.

# Specifications of a simple type checker

The type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the type of its sub-expressions.



# Type Checking of Expressions

- $E \rightarrow \text{literal} \{ E.\text{type} = \text{char}; \}$
- $E \rightarrow \text{num} \{ E.\text{type} = \text{integer}; \}$
- $E \rightarrow \text{id} \{ E.\text{type} = \text{lookup}(\text{id}.\text{entry}); \}$
- $E \rightarrow E_1 \text{ mod } E_2 \{ E.\text{type} = \text{if } (E_1.\text{type} == \text{integer}) \text{ if } (E_2.\text{Type} == \text{integer}) \text{ integer; else type-error;}$
- $E \rightarrow E_1[E_2] \{ E.\text{type} = \text{if } ((E_2.\text{type} == \text{integer}) \&\& (E_1.\text{type} == \text{array}(s, t))) t; \text{ else type-error;}$
- $E \rightarrow *E_1 \{ E.\text{type} = \text{if } (E_1.\text{type} == \text{pointer}(t)) t \text{ else type-error;}$

# Type Checking for Statements

- $S \rightarrow id = E \{ \text{if } (id.type == E.type) \text{ void; else type-error;} \}$
- $S \rightarrow \text{if } E \text{ then } S \{ \text{if } (E.type == \text{boolean}) S_1.type; \text{else type-error;} \}$
- $S \rightarrow \text{While } E \text{ do } S \{ \text{if } (E.type == \text{boolean}) S_1.type; \text{else type-error;} \}$
- $S \rightarrow S_1; S_2; \{ \text{if } (S_1.type == \text{void}) \text{ if } (S_2.type == \text{void}) \text{ void; else type-error;} \}$

# Type Checking of Functions

- $T \rightarrow T \rightarrow T \{ T.type = T1.type \rightarrow T2.type \}$
- $E \rightarrow E(E) \{ E.type = f ((E\_2.type == s) \ \&\& \ (E\_1.type == s \rightarrow t)) \ t; \text{ else type-error};$

# Polymorphism

The *polymorphism* refers to 'one name having many forms', i.e. 'different behavior of an instance depending upon the situation'.

C++ implements *polymorphism* through *overloaded functions* and *overloaded operators*.

The term 'overloading' means a name having two or more distinct meanings.

Overloading occurs when the *same* operator or function *name* is used with *different signatures*.

## Contd...

- Overloading occurs when the *same* operator or function *name* is used with *different signatures*.
- Both operators and functions can be overloaded.
- Operators except below, are possible to overload.
  - . (dot)
  - :: (qualifier)
  - ?: (ternary operator)
  - sizeof
- Different definitions must be distinguished by their signatures (otherwise which to call is ambiguous).
  - Reminder: signature is the operator/function name and the ordered list of its argument types.
  - E.g., **add(int, long)** and **add(long, int)** have different signatures.

# Assignment

1. Explain Chomsky hierarchy of languages and recognizers with examples
2. Define Type Checker. Write down the specification of a simple Type Checker
3. a) What are the different types of errors may have at various stages of program execution? Explain in-detail with examples
4. b) What is the need of type conversion? And discuss the different types of conversion methods
5. Define polymorphism? Explain about operator overloading and function overloading with an example
6. What is type expression? Discuss different types of type expressions with examples

# **Unit – 4**

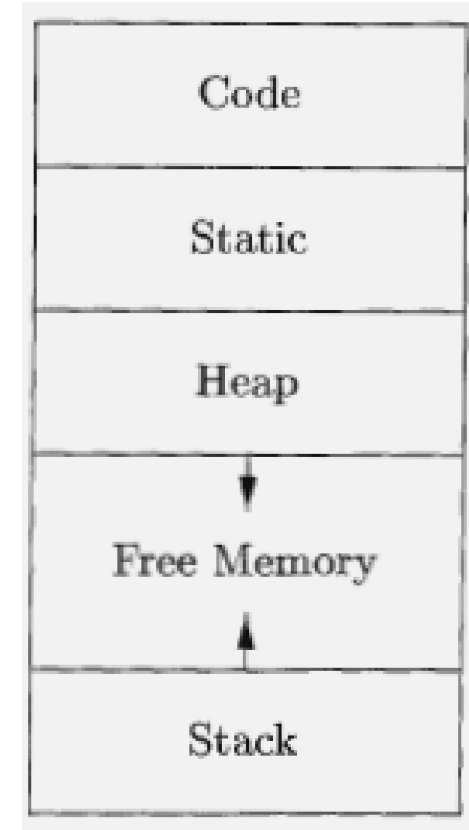
## **Storage Organization**

- 1. Storage language Issues**
- 2. Storage Allocation**
- 3. Storage Allocation Strategies**
- 4. Scope**
- 5. Access to Nonlocal Names**
- 6. Parameter Passing**
- 7. Dynamics Storage Allocation Techniques**

# Storage language Issues:

## Runtime Environment / Runtime Storage Management:

- Always all the programs are stored in hard disk until compilation but CPU / processor executes the program only if it is available in main memory
- OS allocates free memory to the program then compiler uses this in order to store the compiled program
- Main memory can be divided into 4 parts
- Code contains the target executable code(.obj + library files= .exe)
- Static data area stores the static and global variables
- For better utilization of free space we used heap and stack memory
- When a function is called then activation record gets created
- Stack is used to store the activation record information and it is pushed on to the top of the stack and used more often if there are many functions
- Heap is a dynamic data structure which allocates memory at run time and used more often if there are many dynamic, pointer variables
  - In C we used malloc(), calloc(), realloc(), free()
  - In java we used new(), delete()





# Activation Record:

- ✓ Used to manage the information needed by a single execution of a procedure
- ✓ Procedure calls & returns are usually managed by runtime stack called control stack
- ✓ When the activation begins then the procedure name will push on to the stack and when the activation ends / returns then it will be popped
- ✓ The root of all activation records will be at the bottom of the control stack
- ✓ Activation tree represents the sequence & relation between caller & called function

## Model of Activation Record / Fields of Activation Record:

Actual Parameters: Holds the actual parameters of the calling function

Returned Values: Stores the result of function call

Control / Dynamic Link: Points to the activation record of calling function

Access / Static Link: Refers to the local data of called function but found in another AR

Saved Machine status: Stores the address of next instruction to be executed (prg' countr)

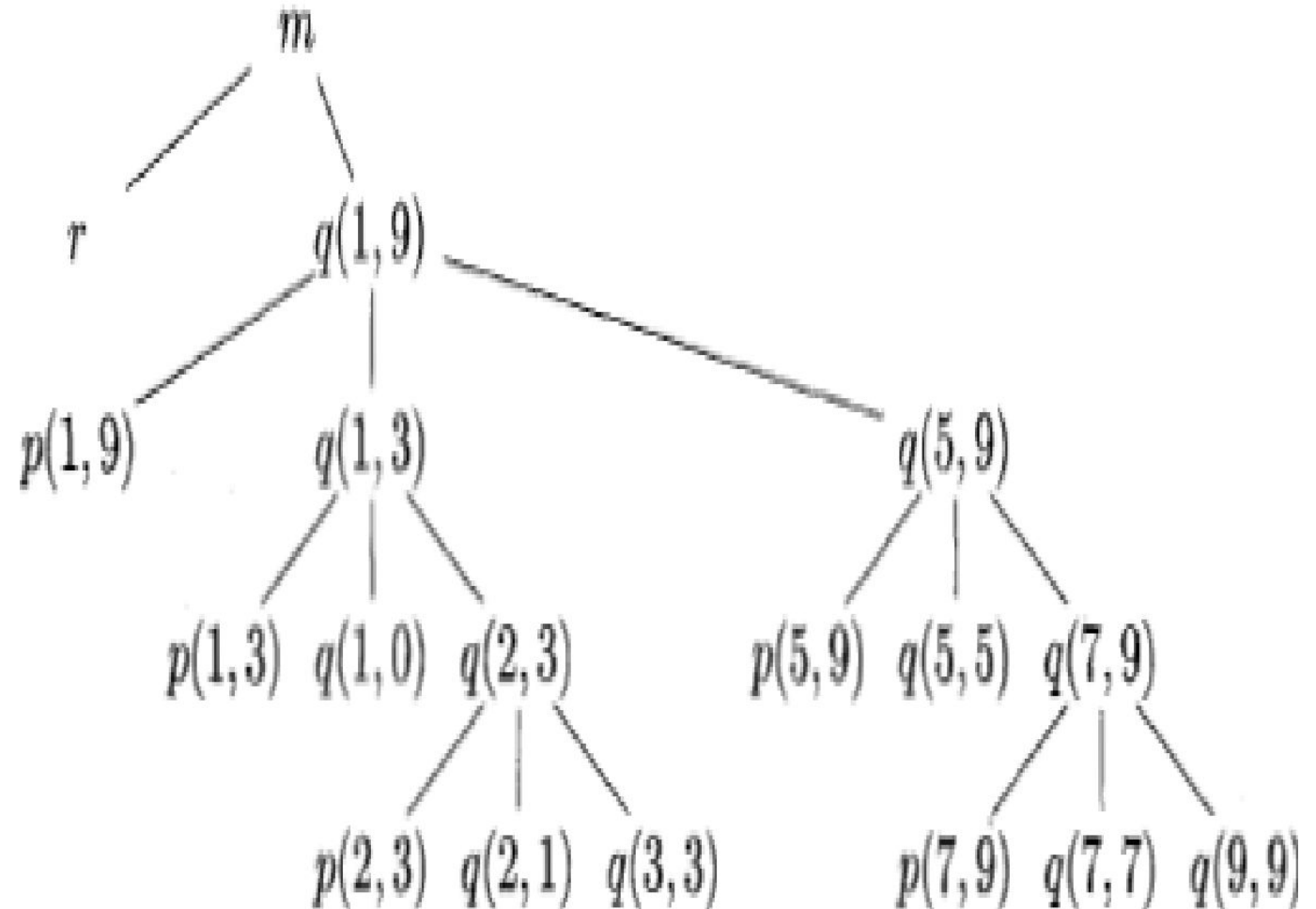
Local Variables: Holds the data that is local to the execution of corresponding function

Temporary Variables: Holds the data that arises during expression evaluation

## Activation for QS:

```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
    ...
    leave quicksort(1,3)
    enter quicksort(5,9)
    ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

## Activation tree representing calls during an execution of QS:



# Storage Allocation Strategies / Storage Organization:

## Static Allocation:

- ✓ The allocation of memory can be done at compile time (variables, arrays)
- ✓ We know how much memory is required for the declared variables before program written
- ✓ Supports the dynamic data structure i.e., memory is created at compile time and deallocated after program completion

### □ Drawbacks:

1. Can't change the size of the memory once you fixed
2. Memory wastage if more memory allocated than the required memory
3. Causes problem if less memory allocated than the required memory
4. Insertions and deletions are very expensive (more no. of elements to be shifted)

## Dynamic Allocation:

- ✓ The allocation of memory / size can be done at run time
- ✓ Many times we don't know how much memory is needed for the variables / arrays
- ✓ Memory will be allocated / deleted by using new / delete operators

**new operator:** It creates a memory & returns memory address to the pointer      new datatype;

int \*p;    p=new int;    \*p=10;      cout<<\*p;

int \*p;    p=new int[5]; //p points to the address of memory location of int type

# Storage Allocation Strategies / Storage Organization:

**delete operator:** It creates a memory & returns memory address to the pointer new datatype;

```
int *p;    p=new int;    *p=10;    cout<<*p;  
int *p;    p=new int[5]; //p points to the address of memory location of int type
```

## **Stack:**

- When a function is called then activation record gets created for the corresponding function and will be push down to the stack (for every function there is an AR)
- Activation record contains the locals so that they are bound to fresh storage in each AR
- Size of the memory(var's) is static and variables resides in stack can be accessed directly

## □ **Drawbacks:**

1. Supports dynamic memory allocation but slower than static memory allocation
2. Supports recursion but references to non-local variables after AR can't be retained

## **Heap:**

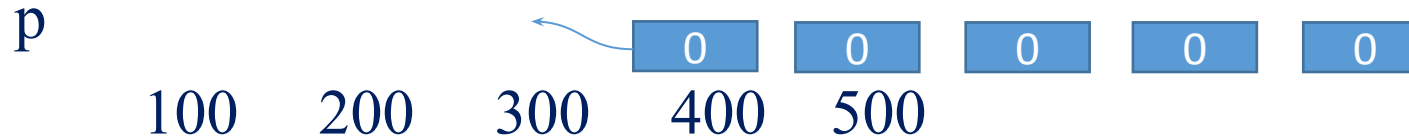
- Implements dynamic memory allocation (at any time allocation & deallocation done)
- Size of the memory(store pointer variables & used by malloc()) is dynamic and variables resides in heap can be accessed indirectly

# Dynamic Memory Allocation:

- ✓ The allocation of memory can be done at run time / execution
- ✓ All these functions are available in `alloc.h` / `stdlib.h`

## □ malloc() – memory allocation

- used to allocate memory for a pointer variable during execution
- Allocates requested size in bytes & returns a pointer to the starting address of memory block
- Allocates only one block of memory
- `datatype *ptr = (datatype *) malloc (bytesize);`    `int *p = (int *) malloc (n*sizeof(int));`
- Accepts only one argument and return type of `malloc()` is void so type casting is required
- Default initial values are garbage (unpredicted) values



## □ calloc() – contiguous memory allocation

- used to allocate memory for a pointer variable during execution
- Allocates multiple blocks of memory & returns a pointer to the starting / 1<sup>st</sup> block
- `datatype *ptr = (datatype *) calloc (no. of blocks, sizeofblock);`    `int *p = (int *) calloc (n, sizeof(int));`
- Requires 2 arguments and default values are 0 (zero)

# Dynamic Memory Allocation:

## □ realloc() – modification in memory

- Used to modify the size of memory which is already allocated

- `realloc(p, bytesize);`

- `int *p = (int *) malloc (3*sizeof(int));` □ `realloc(p, 5*sizeof(int));`

## □ free() – delete memory

- Used to destroy / release the memory which is allocated for the corresponding pointer variable

- `free(p);`

# Parameter Passing:

- ✓ Mechanism which is used to pass parameters to a procedure (subroutine) or function
- ✓ The communication medium among procedures is known as parameter passing
- ✓ The values of the variables from a calling procedure are transferred to the called procedure by some mechanism

## r-value

- The value of an expression is called its r-value and r-values can always be assigned to some other variable
- The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator

## l-value

- The location of memory (address) where an expression is stored is known as the l-value of that expression

It always appears at the left hand side of an assignment operator

## Ex:

**day = 1;   week = day \* 7;   month = 1;   year = month \* 12;**

- Here the constant values like 1, 7, 12 and variables like day, week, month and year are r-values
- variables have l-values as they also represent the memory location assigned to them

**7 = x + y;**

- is an l-value error, as the constant 7 does not represent any memory location



# Parameter Passing (Cont...):

## Formal Parameter:

1. Variables that take the information passed by the caller procedure are called formal parameters
2. These variables are declared in the definition of the called function

## Actual Parameter:

1. Variables whose values and functions are passed to the called function are called actual parameters
2. These variables are specified in the function call as arguments

## **□Parameter passing techniques**

1. call by value
2. call by reference
3. call by value result
4. call by name

## call by value

- The calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record
- Formal parameters then hold the values passed by the calling procedure
- If the values held by the formal parameters are changed, it should have no impact on the actual parameters

## call by reference

- The l-value of the actual parameter is copied to the activation record of the called procedure
- The called procedure has the address of actual parameter & formal parameter refers to the same address
- If the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value



# Parameter Passing (Cont...):

## call by value result

- It also works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends
- On function call, the values of actual parameters are copied in the activation record of the called procedure
- Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters

**int a=1, b=2;**

**procedure(a,b);**

**Procedure(x,y) { x = x + 2; a = x \* y; x = x + 1; }**

- Value of a = 7 when we use call by reference
- Value of a = 4 when we use call by value result

## call by name

- The name of the procedure being called is replaced by its actual body
- Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so it can work on actual parameters, much like pass-by-reference

**void procedure(int x, int y){for(int k=0; k<10; k++) {y = 0; x++;}}**

**main() { int j = 0, A[10]; procedure(j, A[j]);}**

# **Unit – 4( Part 2)**

## **Code Optimization**

1. Issues in the design of code optimization
2. Principal sources of optimization
3. Optimization of basic blocks
4. Loop optimization
5. Peephole optimization
6. Flow graphs
7. Data flow analysis of flow graphs

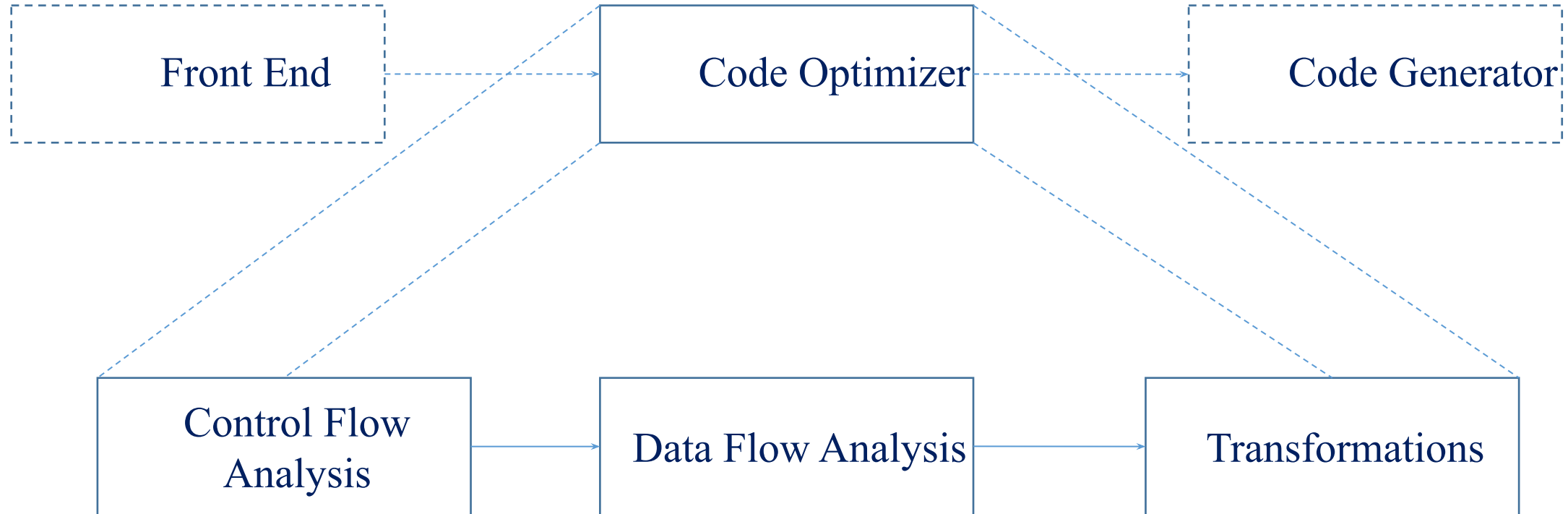
## **UNIT -5**

## **Code Generation:**

1. Issues in the design of code Generation
2. Machine Dependent Code Generation
3. object code forms
4. generic code generation algorithm
5. Register allocation and assignment
6. DAG representation of basic Blocks
7. Generating code from DAGs

# Code Optimizer:

## 1.Position of code optimizer



## 2.Purpose of code optimizer

- ✓ To get better efficiency
  - Run faster
  - Take less

# Code Optimizer (Cont...):

## 3. Places for potential improvements by the user and the compiler

### ✓Source code

□User can profile program, change algorithm or transform loops.

### ✓Intermediate code

□Compiler can improve loops, procedure calls or address calculations

### ✓Target code

□Compiler can use registers, select instructions or do peephole transformations

## code optimization Techniques

1. Compile Time Evaluation
2. Variable Propagation
3. Dead Code Elimination
4. Code Motion
5. Induction Variable & Strength Reduction

# Principal Sources of Optimization:

- ✓ It optimizes the code by removing unnecessary lines of code so we can use less amount of memory to store it and execute it in a fast manner

1. Common sub expression elimination
2. Code Motion / Code movement
3. Compile Time Evaluation
  - a. Constant folding b. Constant propagation
4. Dead Code Elimination
5. Induction Variable & Strength Reduction

## 1. Common Subexpression Elimination

- ✓ It is an expression which appears repeatedly in the program, which is computed previously but the values of variables in expression doesn't changed
- ✓ It replaces the redundant expression each time it is encountered

### unoptimized code

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

### optimized code

$a = b + c$

$b = a - d$

$c = b + c$

$d = b$

# Principal Sources of Optimization (Cont...):

## 2. Compile Time Evaluation

- ✓ Evaluation is done at compile time instead of run time & It can be done in 2 ways
  - a. **Constant folding**
    - Evaluate the expression and submit the result
    - $\text{area} = (22/7) * r * r$  here 22/7 is calculated and result 3.14 is replaced so  $\text{area} = 3.14 * r * r$
  - b. **Constant propagation**
    - Constant replaces a variable
    - $\text{pi} = 3.14, r = 5, \text{area} = \text{pi} * r * r$  □  $\text{area} = 3.14 * 5 * 5$

## 3. Code Motion / Code Movement

- ✓ It is a technique which moves the code outside the loop if it won't have any difference, if it executes inside or outside the loop

### unoptimized code

```
for(i=0; i<n; i++) {  
    x=y+z;  
    a[i] = 6*i; }  
}
```

### optimized code

```
x = y + z;  
for(i=0; i<n; i++) {  
    a[i] = 6*i; }  
}
```

# Principal Sources of Optimization (Cont...):

## 4. Dead Code Elimination

- ✓ It eliminates the statements which are never executed or if executed its output is never used

<u>unoptimized code</u>	<u>optimized code</u>	<u>unoptimized code</u>	<u>optimized code</u>
i = 0; if(i==1) { a = x + i; }  	i = 0;      	int add(int x, int y) { int z; z=x+y; return z; printf("%d",z); }	int add(int x, int y) { int z; z=x+y; return z; }

## 5. Induction Variable & Strength Reduction

Replacement of expensive operator with cheaper operator (\* >>>> +)

# Issues in the design of code optimization:

- ✓ Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed
- ✓ In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.
- ✓ A code optimizing process must follow the three rules given below:
  - The output code must not, in any way, change the meaning of the program
  - Optimization should increase the speed of the program and if possible, the program should demand less number of resources
  - Optimization should itself be fast and should not delay the overall compiling process



# Optimization of Basic Blocks:

- ✓ Reduce the no. of lines in a block by optimization
- ✓ In order to optimize a block, we use the below 5 approaches
  1. Common Sub Expression Elimination
  2. Dead Code Elimination
  3. Renaming Temporary Variables
  4. Interchange of Statements
  5. Algebraic Transformations

**1. Common Sub Expression Elimination**      □ refer slide-133

**2. Dead Code Elimination**      □ refer slide-135

**3. Renaming Temporary Variables**

- Same variables contains multiple values but the recent value will override previous value

**unoptimized code**

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

**optimized code**

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

# Optimization of Basic Blocks (Cont...):

## 4. Interchange of Statements

- Based on the operation we can interchange the statements

### unoptimized code

$t1 = b + c$

$t2 = a - t1$

$t3 = t1 * d$

$d = t2 + t3$

### optimized code

$t1 = b + c$

$t3 = t1 * d$

$t2 = a - t1$

$d = t2 + t3$

## 5. Algebraic Transformations

- Use this approach when you get direct value instead of performing operation

### unoptimized code

$t1 = a - a$

$t2 = b - t1$

$t3 = t2 * 2$

### optimized code

$t1 = 0$

$t2 = b$

$t3 = t2 \ll 1$  ( $\equiv$  multiply with 2)

# Loop Optimization:

- ✓ We can apply optimization techniques on loops also. These are
  1. Code Motion / Code Movement
  2. Loop Invariant Computations
  3. Loop Unrolling
  4. Loop Fusion

## 1. Code Motion / Code Movement

- ✓ It is a technique which moves the code outside the loop if it won't have any difference, if it executes inside or outside the loop

### unoptimized code

```
for(i=0; i<n; i++) {  
    x=y+z;  
    a[i] = 6*i;    }
```

### optimized code

```
x = y + z;  
for(i=0; i<n; i++) {  
    a[i] = 6*i;    }
```

## 2. Loop Invariant Computations

- The statements in the loop whose result of the computations do not change over the iterations

**Ex:**    a=10; b=20; c=30;

```
for(i=0;i<10;i++) { c=a+b; d=a-b; c=a*b; s=s+i;    }
```

Here 1<sup>st</sup> 3 statements in for loop is loop invariant computations i.e., these 3 statements are not dependent on for loop so we can eliminate these statements

# Loop Optimization (Cont...):

## 3. Loop Unrolling

- Loop overhead can be reduced by reducing the no. of iterations & replacing body of the loop

**Ex:**     for(i=0;i<100;i++) {   add();   }

**reduced to**

for(i=0;i<50;i++) {   add();   add();   }

## 4. Loop Fusion

- Adjacent loops can be merged into one loop to reduce loop overhead & improve performance

**Ex:**     for(i=0;i<10;i++) {   a[i]=a[i]+10;   }

for(i=0;i<10;i++) {   b[i]=b[i]+10;   }

**reduced to**

for(i=0;i<10;i++) {   a[i]=a[i]+10;   b[i]=b[i]+10;   }

# Peephole Optimization:

- ✓ This is applied to improve the performance of program by examining a short sequence of instructions in a window (peephole) & replace the instructions by a faster or short sequence of instructions
- ✓ we use the below 5 approaches
  1. Redundant Instruction Elimination
  2. Removal of Unreachable Code
  3. Flow of Control Optimizations
  4. Algebraic Simplifications
  5. Machine Idioms

## 1. Redundant Instruction Elimination

Consider the instructions

mov R0, a	} (we can eliminate 2 <sup>nd</sup> statement since a value is already in R0 )	
mov R0, a		
		mov a , R0

## 2. Removal of Unreachable Code

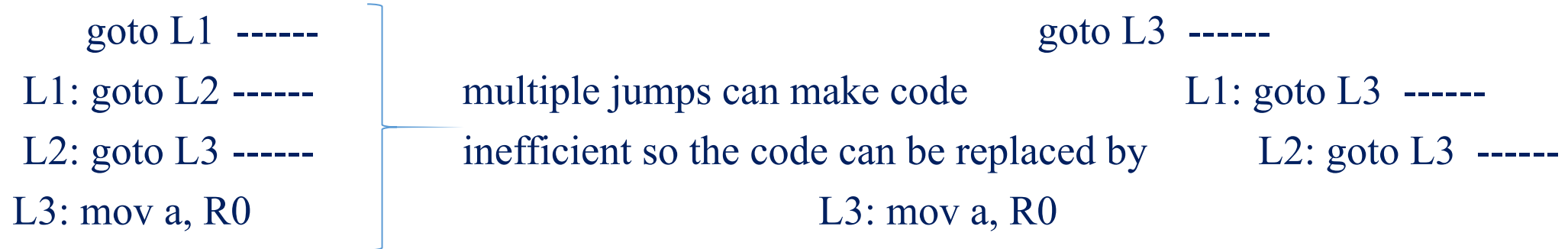
- Eliminates the statements which are unreachable i.e., never executed

i = 0;	i = 0;	int add(int x, int y) {	int add(int x, int y) {
if(i==1) {		int z=x+y;	int z=x+y;
sum = 0; }	} Eliminate	return z;	return z; }
printf("d",z); }			

# Peephole Optimization (Cont...):

## 3. Flow of Control Optimizations

- Unnecessary jumps can be eliminated by using this



## 4. Algebraic Simplifications

$i = i+0;$      $i = i*1;$     // These can be eliminated because the result won't change by executing these statements

$a=x^2$  can be replaced with  $a=x*x$

$b=y/8$  can be replaced with  $b= y>>3$

## 5. Machine Idioms

- It is the process of using powerful features of CPU instructions

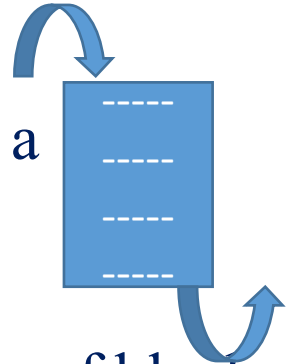
**Ex:** Auto increment / decrement features can be used to increment / decrement variables

$a=a+1$  can be replaced with `inc a` similarly  $a=a-1$  can be replaced with `dec a`

# Flow graphs:

## Basic Block

- ✓ A connection of sequence of consecutive statements which always executed in a sequential manner
- ✓ It should consists of exactly one entry and one exit point in each block
- ✓ It shouldn't contain conditional / unconditional control statements in the middle of block



Ex:  $x=a+b+c$  is not in 3 address code so convert it into  $(t1=a+b \quad t2=t1+c \quad x=t2)$  □ block

Algorithm for partitioning a 3 address code into basic block

Rule1: Determining the leaders

- 1<sup>st</sup> statement is a leader
- The target of the conditional or unconditional jump is a leader
- The statement following conditional or unconditional jump is a leader

Rule2: Determining the basic blocks

- We can determining the basic block beginning at leaders block and ends before the next leader

# Flow graphs (Cont...):

1. PROD=0 Here the leaders are 1, 5, 7, 10, 13

2. I=1 basic blocks are

3. T2=addr(A)-4

4. T4=addr(B)-4

5. T1=4\*I

6. T3=T2[T1]

7. PROD=PROD+T3

8. I=I+1

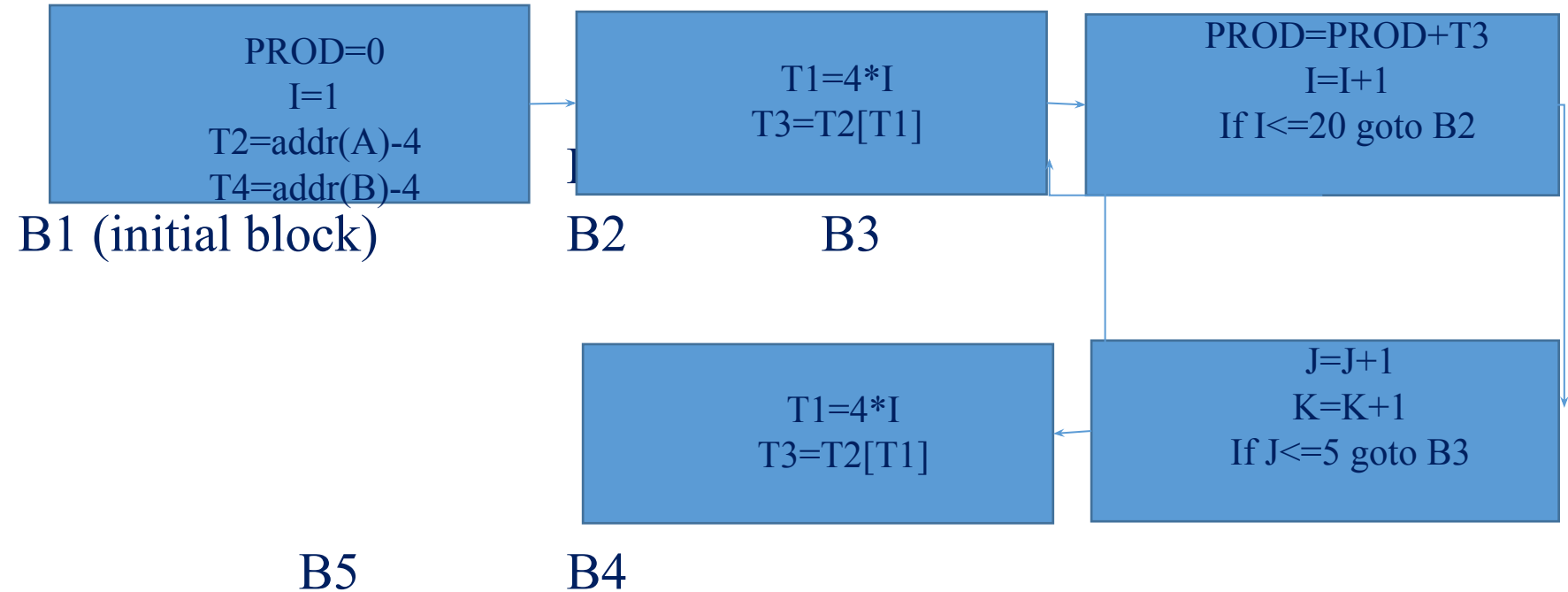
9. If I<=20 goto(5)

10. J=J+1

11. K=K+1

12. If J<=5 goto(7)

13. I=I+J



**Flow Graph:** It is a directed graph in which flow control information is added to the graph (basic blocks)

It contains collection of nodes where the edges are connected from one node to another node

In this basic blocks are nodes of the graph and edges are connected for flow of information



# Issues in the design of code Generator:

✓ while designing the target code efficiently we need to check the following issues

## 1. Input to Code Generator

- Accepts input from code optimizer/intermediate code generator and produces optimized intermediate code
- Intermediate code is represented in several ways like postfix notation, three address code (quadruple, triple, indirect triple), DAG / syntax tree

## 2. Target Program

- Code generator produces the target program in any one of the below given

### □ Absolute machine language

- The program will be stored in fixed memory location (MM) i.e., memory location won't be changed irrespective of any program
- It is mainly suitable if the program is very small so that it will be compiled and executed in a faster manner

### □ Relocatable machine language (object code)

- It requires linker and loader
- Linker links the object code of several programs into a single program (printf() in stdio.h, string functions in string.h etc into one)
- Loader will load the executable file into main memory and if free space is available in MM then it stores this program into MM
- It is suitable for any program

### □ Assembly language

- For code generation this is very better and by using this we can generate the code in a very easy manner

# Issues in the design of code Generator (Cont...):

## 3. Memory Management

□ By using the symbol table memory management can be done(token information)

## 4. Instruction Selection

□ In order to produce efficient code, instruction selection & the speed of the instruction is very important

□ **Ex:**  $x=y+z$  □ `MOV R0, y   ADD R0, z   MOV x, R0`

$a=a+1$  □ `MOV R0, a   ADD R0, #1   MOV a, R0` □ `INC a`

$a=b+c$   $d=a+e$  □ `MOV R0, b   ADD R0, c   MOV a, R0   MOV R0, a   ADD R0, e   MOV d, R0`

□ `MOV R0, b   ADD R0, c   ADD R0, e   MOV d, R0`

## 5. Register Allocation

□ We can perform more no. of operations with a limited no. of registers & which can be done in 2 ways

□ **Register allocation** □ specifies which register contains which variable( $R0 \sqsupset a$  means R0 contains var' a)

□ **Register assignment** □ specifies which variable contains which register( $a \sqsupset R0$  means a contains var' R0)

□ **Ex:**  $t=a+b$   $t=t*c$   $t=t/d$  □ `MOV R0, a   ADD R0, b   MUL R0, c   DIV R0, d   MOV t, R0`

## 6. Evaluation Order

□ The order in which the instructions are executed as well as the operations are performed

# Object Code Forms / target Machine / Simple target Machine Model:

✓ Let us assume that target computer uses the following instructions

## 1. Load Operation:

□ Load memory word to register i.e., LD R1, X (load the value in location X to register R1)

## 2. Store Operation:

□ Store register to memory word i.e., ST X, R1 (store the value in register R1 into location X)

## 3. Computational Operation:

□ This is represented in the form of OP dest, src1, src2 where OP is operator & src's, dest are loc'n/reg'r

□ We can do ADD, SUB,...etc... **Ex:** ADD R1, R2, R3 □  $R1 = R2 + R3$  and Inc X

## 4. Conditional Jump:

□ By checking the condition the control goes to the corresponding location like if, for, switch

**Ex:** BLTZ r, L □ branch to L if the content of r is  $< 0$  otherwise it executes the next statement

## 5. Unconditional Jump:

□ Without checking any condition the control will be transferred to the corresponding location like break, continue, goto, exit **Ex:** BR L here L is a label

So these instructions are used by target machine & will see the addressing modes used by target machine

# Object Code Forms (Cont...):

- ✓ Let us assume that our target machine has a variety of addressing modes
- ✓ By using addressing modes we can get the effective address & it contains corresponding operand

<u>Addressing Mode</u>	<u>Form</u>	<u>Address</u>
------------------------	-------------	----------------

Absolute	M	M
----------	---	---

Register	R	R
----------	---	---

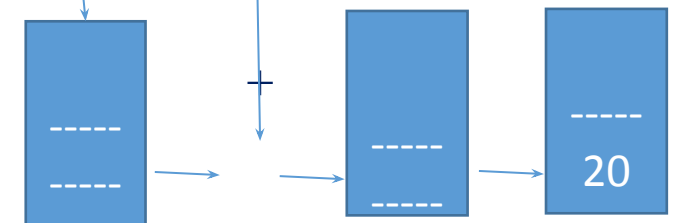
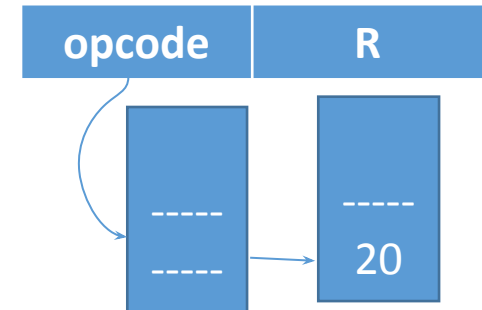
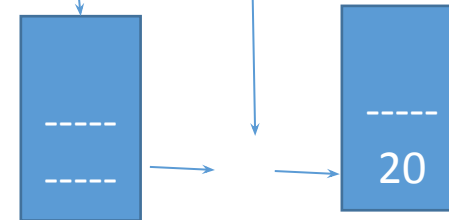
Indexed	C(R)	C+contentes(R)
---------	------	----------------

Indirect register	*R	Contentes(R)
-------------------	----	--------------

Indirect indexed	*C(R)	Contentes(C+contentes(R))
------------------	-------	---------------------------

Immediate / Literal	#	N/A
---------------------	---	-----

- Instead of address we can give value directly & no need of address so we write N/A under address and we are not representing in any form so we write #



# Generic code generation algorithm :

- ✓ It generates target code for sequence of instructions
- ✓ It uses a function getReg() to assign registers to variables
- ✓ It uses two data structures
  1. Register Descriptor
    - Used to keep track of which variable is stored in which register. Initially all registers are empty
  2. Address Descriptor
    - Used to keep track of location where variable is stored. Location may be register, memory address, stack,...
- ✓ The following actions are performed by code generator for an instruction  $x = y \text{ op } z$
- ✓ Assume that L is a location where the output of  $y \text{ op } z$  is to be stored
  1. Call the function getReg() to get the location of L
  2. Determine the location of y by consulting address descriptor of y. If y is not present in any location 'L' then it generate the instruction `mov y', L` (copy value of y to L)
  3. Determine the location of z by using step2 & the instruction is generated as `op z', L`
  4. Now L contains the value of  $y \text{ op } z$  i.e., assigned to x. So, if L is a register then update its descriptor that it contains value of x. as well as update address descriptor of x to indicate that it is stored in L
  5. If y and z have no further use then update the descriptor to remove y and z

# Generic code generation algorithm (Cont...):

**Ex:**  $d = (a-b) + (a-c) + (a-c)$

Three address code for the expression is  $t1=a-b$        $t2=a-c$        $t3=t1+t2$        $d=t3+t2$

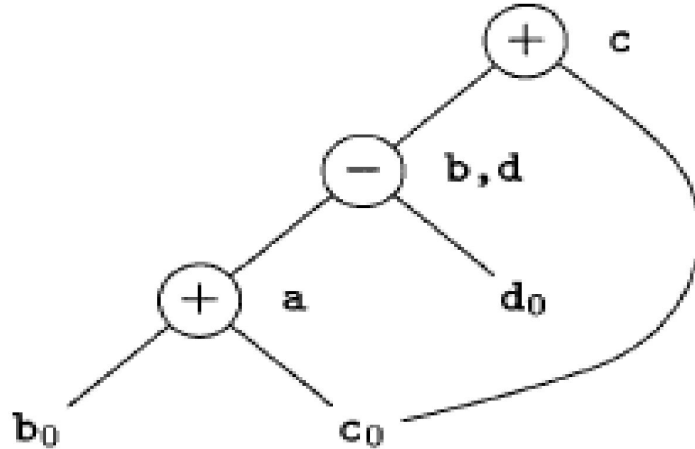
<u>Statement</u>	<u>Code Generator</u>	<u>Register Descriptor</u>	<u>Address Descriptor</u>
$t1=a-b$	mov R0, a sub R0,b	R0 contains t1	t1 in R0
$t2=a-c$	mov R1, a sub R1,b	R0 contains t1 R1 contains t2	t1 in R0 t2 in R1
$t3=t1+t2$	add R0, R1	R0 contains t3 R1 contains t2	t3 in R0 t2 in R1
$d=t3+t2$	add R0, R1	R0 contains d	d in R0

# DAG representation of basic blocks:

- ✓ There is a node for each of the initial values of the variables in the DAG that are appeared in the basic block
- ✓ There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s of the operands used by s
- ✓ Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block
- ✓ Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block

## Ex1:

```
a = b + c
b = a - d
c = b + c
d = a - d
```



```
a = b + c;
b = b - d
c = c + d
e = b + c
```

