UNIT -1

Fundamentals

Symbol – An atomic unit, such as a digit, character, lower-case letter, etc. Sometimesa word.*[Formal language does not deal with the "meaning" of thesymbols.]*

Alphabet – A finite set of symbols, usually denoted byΣ.
Σ ={0, 1}
Σ = {0, a,9, 4}
Σ = {a, b, c,d}

String – A finite length sequence of symbols, presumably from some alphabet. w=0110
y=0aa
x=aabcaa
z = 111

Special string: ε (also denoted by λ)

Concatenation:	wz = 0110111		
Length:	$\|w\| = 4$	$\|\varepsilon\| = 0$	$\|x\| = 6$
Reversal:	y^R = aa0		

Some special sets ofstrings:

Σ^*	All strings of symbols fromΣ
Σ^+	Σ^* -{ε}

Example: Σ = {0,1}
Σ^* = {ε, 0, 1, 00, 01, 10, 11, 000, 001,…}
Σ^+ = {0, 1, 00, 01, 10, 11, 000, 001,…}

A **language**is:
A set of strings from some alphabet (finite or infinite). In otherwords,
Any subset L ofΣ^*

Some speciallanguages:
{}The empty set/language, containing nostring.
{ε}A language containing one string, the emptystring.

Examples:
Σ = {0,1}
L = {x | x is in Σ^* and x contains an even number of 0"s}

Σ = {0, 1, 2,…, 9, .}
L = {x | x is in Σ^* and x forms a finite length real number}

= {0, 1.5, 9.326,…}

Σ = {a, b, c,…, z, A, B,…, Z}
L = {x | x is in Σ^* and x is a Pascal reserved word}
= {BEGIN, END, IF,…}

Σ = {Pascal reserved words} U { (,), ., :, ;,…} U {Legal Pascal identifiers} L = {x | x is in Σ^* and x is a syntactically correct Pascal program}

Σ = {English words}
L = {x | x is in Σ^* and x is a syntactically correct English sentence}

Regular Expression

- A regular expression is used to specify a language, and it does soprecisely.
- Regular expressions are veryintuitive.
- Regular expressions are very useful in a variety ofcontexts.
- Given a regular expression, an NFA-ε can be constructed from itautomatically.
- Thus, so can an NFA, a DFA, and a corresponding program, allautomatically!

Definition:
Let Σ be an alphabet. The regular expressions over Σare:
Ø Represents the empty set {}
E Represents the set{ε}
Represents the set {a}, for any symbol a inΣ
Let r and s be regular expressions that represent the sets R and S, respectively.

r+sRepresents the set RUS	(precedence3)
rsRepresents thesetRS	(precedence2)
r^* Represents thesetR*	(highest precedence)
(r) Represents thesetR	(not an op, providesprecedence)

If r is a regular expression, then L(r) is used to denote the correspondinglanguage.

Examples:
Let Σ = {0,1}
(0 +1)* All strings of 0‟s and1‟s0(0 +1)* All strings of 0‟s and 1‟s, beginning with a0
(0 +1)*1 All strings of 0‟s and 1‟s, ending with a1
(0 + 1)*0(0+1)* All strings of 0‟s and 1‟s containing at least one 0 (0 + 1)*0(0 + 1)*0(0+1)* All strings of 0‟s and 1‟s containing at least two
0‟s (0+1)*01*01* All strings of 0‟s and 1‟s containing at least two
0‟s (101*0)* All strings of 0‟s and 1‟s containing an even number of 0‟s
1*(01*01*)* All strings of 0‟s and 1‟s containing an even number of0‟s
(1*01*0)*1* All strings of 0‟s and 1‟s containing an even number of0‟s

Identities:

1. Øu = uØ=Ø	Multiply by0
2. εu = uε=u	Multiply by1
3. Ø* =ε	
4. ε* =ε	
5. u+v =v+u	

6. u + Ø =u

7. u + u = u

8. u* =(u*)*

9. u(v+w) =uv+uw

10. (u+v)w =uw+vw

11. (uv)*u = u(vu)*

12. (u+v)* = (u*+v)*

=u*(u+v)*

=(u+vu*)*
= (u*v*)*

=u*(vu*)*

=(u*v)*u*

Finite State Machines

A finite state machine has a set of states and two functions called the next-state function and the outputfunction
The set of states correspond to all the possible combinations of the internal storage
If there are n bits of storage, there are 2^n possiblestates
The next state function is a combinational logic function that given the inputs and the current state, determines the next state of thesystem
The output function produces a set of outputs from the current state and theinputs

- There are two types of finite statemachines
- In a Moore machine, the output only depends on the currentstate
- While in a Mealy machine, the output depends both the current state and the currentinput
- We are only going to deal with the Mooremachine.
- These two types are equivalent incapabilities

A Finite State Machine consistsof:
Kstates:S = {s_1, s_2, … ,s_k}, s_1 is initial state Ninputs:I = {i_1, i_2, …,i_n}
Moutputs:O = {o_1, o_2, …,o_m}
Next-state function T(S, I) mapping each current state and input to next state Output Function P(S) specifies output

Finite Automata

- Two types – both describe what are called regularlanguages
    - Deterministic (DFA) – There is a fixed number of states and we can only bein one state at atime

- Nondeterministic (NFA) –There is a fixed number of states but wcan bein multiple states at onetime

- While NFA‟s are more expressive than DFA‟s, we will see that addingnondeterminism does not let us define any language that cannot be defined by aDFA.

- One way to think of this is we might write a program using a NFA, but then when it is "compiled" we turn the NFA into an equivalentDFA.

Formal Definition of a Finite Automaton

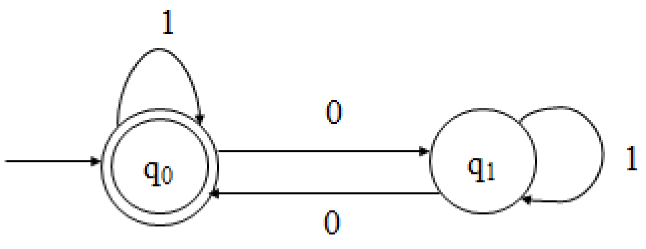
- Finite set of states, typicallyQ.
- Alphabet of input symbols, typically∑
- One state is the start/initial state, typically q0 // q0 ∈Q
- Zero or more final/accepting states; the set is typically F. // F⊆Q
- A transition function, typicallyδ. Thisfunction
- Takes a state and input symbol asarguments.

Deterministic Finite Automata (DFA)

- A DFA is a five-tuple: M = (Q, Σ, δ, q0, F)
  Q=A <u>finite</u> set ofstates
  Σ=A <u>finite</u> inputalphabet
  q0=The initial/starting state, q0 is inQ
  F=A set of final/accepting states, which is a subset ofQ
  Δ=A transition function, which is a total function from Q x Σ toQ
  δ: (Q x Σ)–>Q δ is defined for any q in Q and s in Σ, and δ(q,s)=q‟is equal to another state q‟ inQ.
  Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

- For Example #1:

  $Q = \{q_0, q_1\}$
  $\Sigma = \{0, 1\}$
  Start state is q_0
  $F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

- LetM=(Q,Σ,δ,q_0,F)beaDFAandletwbeinΣ*.Thenwis*accepted*byMiff

$\delta(q_0, w) = p$ for some state p in F.

- Let M = (Q, Σ, δ, q_0, F) be a DFA. Then the *language accepted* by M is theset:

$L(M) = \{w \mid w$ is in Σ^* and $\delta(q_0, w)$ is in $F\}$

- Another equivalentdefinition:

$L(M) = \{w \mid w$ is in Σ^* and w is accepted by $M\}$

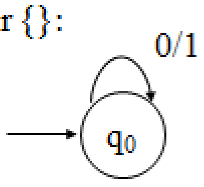
- Let L be a language. Then L is a *regular language* iff there exists a DFA M such that L =L(M).

Notes:

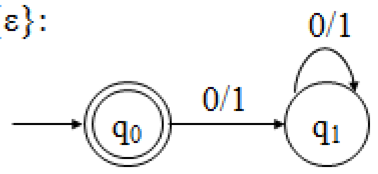
- A DFA M = (Q, Σ, δ,q0,F) partitions the set Σ* into two sets: L(M)and Σ* -L(M).
- If L = L(M) then L is a subset of L(M) and L(M) is a subset ofL.
- Similarly, if L(M1) = L(M2) then L(M1) is a subset of L(M2) and L(M2) is a subset of L(M1).
- Some languages are regular, others are not. For example,if L1 = {x | x is a string of 0's and 1's containing an even number of 1's} and L2 = {x | x = 0^n1^n for some n >= 0}then L1 is regular but L2 is not.

- Let Σ = {0, 1}. Give DFAs for {}, {ε}, Σ^*, and Σ^+.

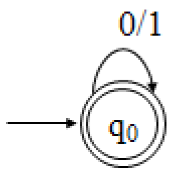
For {}:



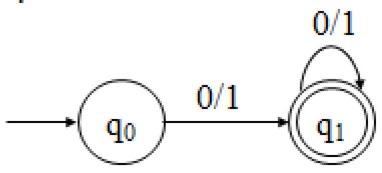
For {ε}:



For Σ^*:



For Σ^+:



<u>Nondeterministic Finite Automata (NFA)</u>

An NFA is a five-tuple: M = (Q, Σ, δ, q_0,F)

Q	A finite set ofstates
Σ	A finite inputalphabet
q0	The initial/starting state, q0 is inQ
F	A set of final/accepting states, which is a subset ofQ
δ	A transition function, which is a total function from Q x Σ to2^Q

δ: (Q x Σ)->$\mathbf{2^Q}$ -2^Q is the power set of Q, the set of all subsets of Q δ(q,s) -The set of all states p such that there is atransition

labeled s from q to p δ(q,s) is a function from Q x S to 2^Q (but not to Q)
Let M = (Q, Σ, δ,q_0,F) be an NFA and let w be in Σ*. Then w is *accepted* by
M iffδ({q_0}, w) contains at least one state inF.

Let M = (Q, Σ, δ,q_0,F) be an NFA. Then the *language accepted* by M is
theset: L(M) = {w | w is in Σ* and δ({q_0},w) contains at least one state inF}
Another equivalentdefinition:
L(M) = {w | w is in Σ* and w is accepted by M}

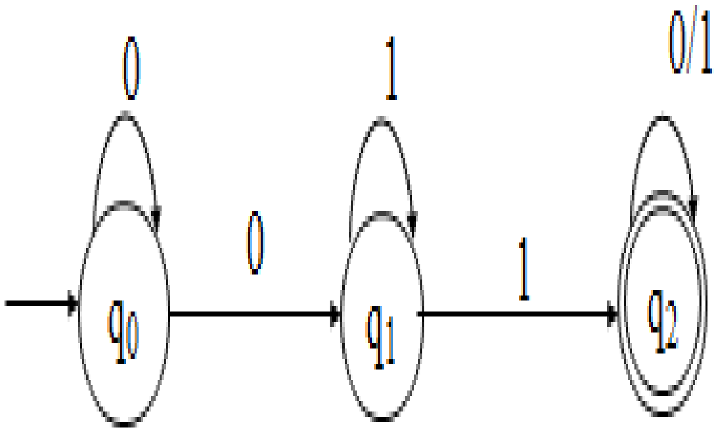
- Example: some 0's followed by some 1's

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_2\}$



δ:	0	1
q_0	$\{q_0, q_1\}$	$\{\}$
q_1	$\{\}$	$\{q_1, q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

$=\{\in, 0, 00, 1, 11, 111, 01, 10, \ldots\ldots\}$

$=\{\in$, any combination of 0's, any combination of 1's, any combination of 0 and 1 $\}$

Hence, L. H. S. = R. H. S. is proved.

3.4 RELATIONSHIP BETWEEN FA AND RE

There is a close relationship between a finite automata and the regular expression we can show this relation in below figure.

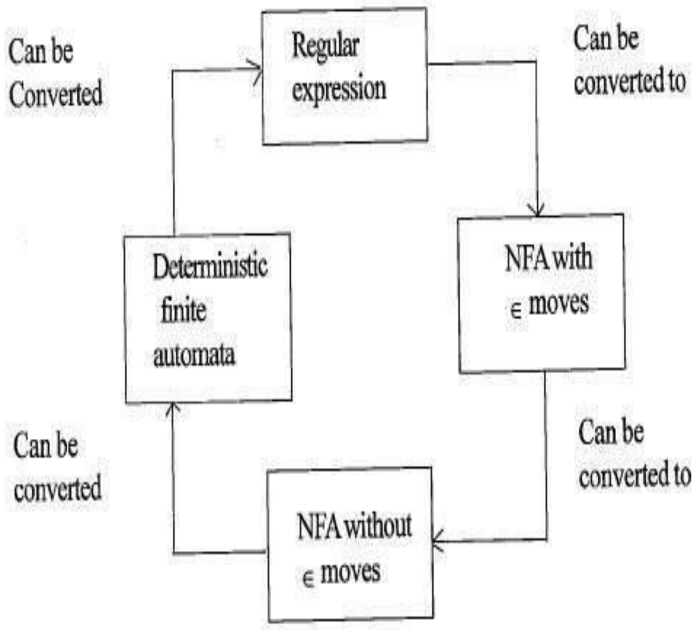


FIGURE : Relationship between FA and regular expression

The above figure shows that it is convenient to convert the regular expression to NFA with $\in$ moves. Let us see the theorem based on this conversion.

3.5 CONSTRUCTING FA FOR A GIVEN REs

Theorem : If r be a regular expression then there exists a NFA with $\in$ - moves, which accepts $L(r)$.

Proof : First we will discuss the construction of NFA M with $\in$ - moves for regular expression r and then we prove that $L(M) = L(r)$.

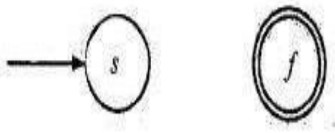
Let r be the regular expression over the alphabet Σ.

Construction of NFA with $\in$ - moves

Case 1 :

(i) $r = \phi$

NFA $M = (\{s, f\}, \{\ \}\delta, s, \{f\})$ as shown in Figure1 (a)

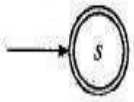


(No path from initial state s to reach the final state f.)

Figure 1 (a)

(ii) $r = \in$

NFA $M = (\{s\}, \{\ \}, \delta, s, \{s\})$ as shown in Figure 1 (b)

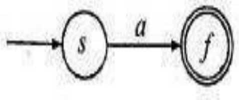


(The initial state s is the final state)

Figure 1 (b)

(iii) $r = a$, for all $a \in \Sigma$,

NFA $M = (\{s, f\}, \Sigma, \delta, s, \{f\})$



(One path is there from initial state s to reach the final state f with label a.)

Figure 1 (c)

Case 2 : $|r| \geq 1$

Let r_1 and r_2 be the two regular expressions over Σ_1, Σ_2 and N_1 and N_2 are two NFA for r_1 and r_2 respectively as shown in Figure 2 (a).

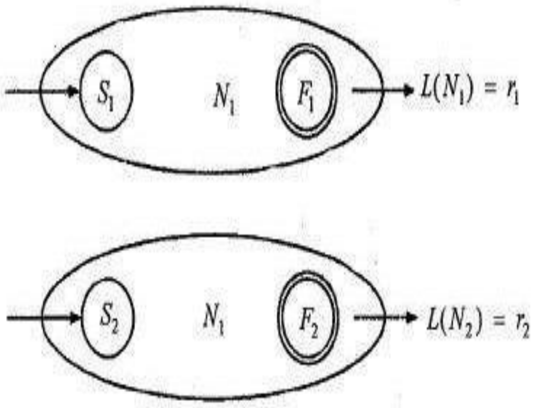
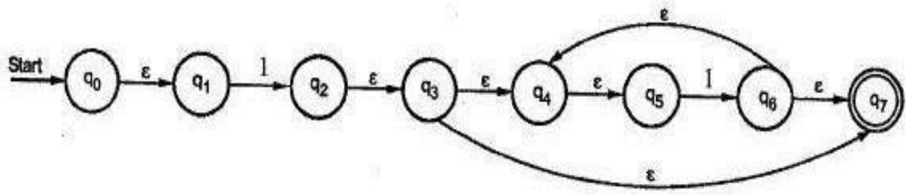


Figure 2 (a) NFA for regular expression r_1 and r_2

The final NFA is



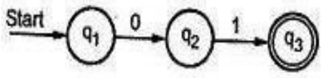
Example 4 : Construct NFA for the r. e. $(01+2^*)0$.

Solution : Let us design NFA for the regular expression by dividing the expression into smaller units

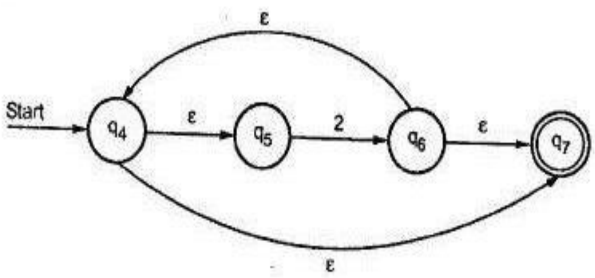
$$r=(r_1+r_2)r_3$$

where $r_1=01$, $r_2=2^*$ and $r_3=0$

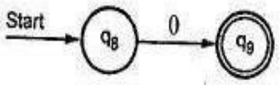
The NFA for r_1 will be



The NFA for r_2 will be



The NFA for r_3 will be



Conversion from NFA to DFA

Suppose there is an NFA N < Q, ∑, q0, δ, F > which recognizes a language L. Then the DFA D < Q'', ∑, q0, δ'', F'' > can be constructed for language L as:

Step 1: Initially Q'' = ϕ.

Step 2: Add q0 to Q''.

Step 3: For each state in Q'', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q'', add it to Q''.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

Example
Consider the following NFA shown in Figure 1.

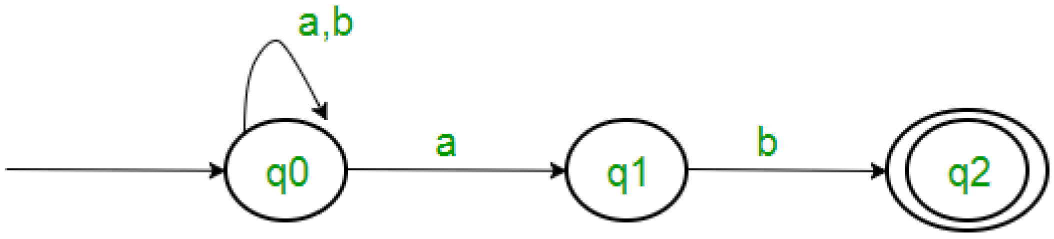


Figure 1

Following are the various parameters for NFA.
Q = { q0, q1, q2 }
∑ = (a, b)
F = { q2 }
δ (Transition Function of NFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q‟ = ϕ
Step 2: Q‟ = {q0}
Step 3: For each state in Q‟, find the states for each input symbol.
Currently, state in Q‟ is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA

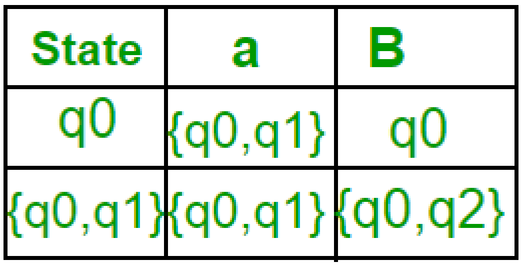
δ‟ (Transition Function of DFA)

State	a	b
q0	{q0,q1}	q0

Now { q0, q1 } will be considered as a single state. As its entry is not in Q‟, add it to Q‟.
So Q‟ = { q0, { q0, q1 } }

Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like:
δ‟ ({ q0, q1 }, a) = δ (q0, a) ∪ δ (q1, a) = { q0, q1 }
δ‟ ({ q0, q1 }, b) = δ (q0, b) ∪ δ (q1, b) = { q0, q2 }
Now we will update the transition table of DFA.

δ‟ (Transition Function of DFA)

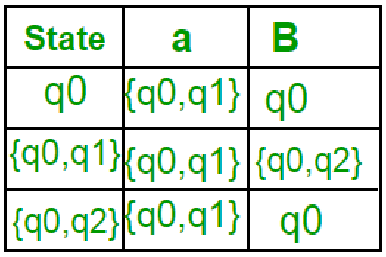


State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now { q0, q2 } will be considered as a single state. As its entry is not in Q‟, add it to Q‟.
So Q‟ = { q0, { q0, q1 }, { q0, q2 } }

Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like:
δ‟ ({ q0, q2 }, a) = δ (q0, a) ∪ δ (q2, a) = { q0, q1 }
δ‟ ({ q0, q2 }, b) = δ (q0, b) ∪ δ (q2, b) = { q0 }
Now we will update the transition table of DFA.

δ‟ (Transition Function of DFA)



State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 }

Following are the various parameters for DFA.
Q‟ = { q0, { q0, q1 }, { q0, q2 } }
∑ = (a, b)
F = { { q0, q2 } } and transition function δ‟ as shown above. The final DFA for above NFA has been shown in Figure 2.

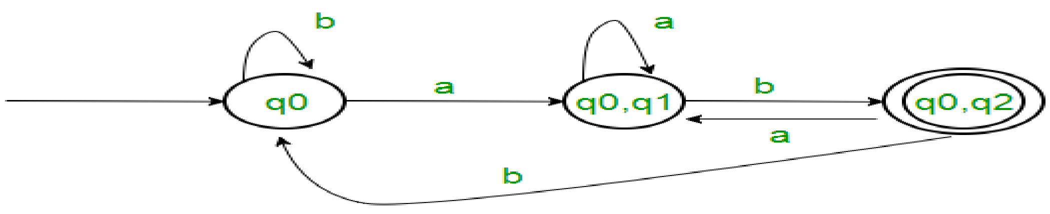


Figure 2

Note : Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA

Application of Finite state machine and regular expression in Lexical analysis: Lexical analysis is the process of reading the source text of a program and converting that source code into a sequence of tokens. The approach of design a finite state machine by using regular expression is so useful to generates token form a given source text program. Since the lexical structure of more or less every programming language can be specified by a regular language, a common way to implement a lexical analysis is to; 1. Specify regular expressions for all of the kinds of tokens in the language. The disjunction of all of the regular expressions thus describes any possible token in the language. 2. Convert the overall regular expression specifying all possible tokens into a deterministic finite automaton (DFA). 3. Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer. To recognize identifiers, numerals, operators, etc., implement a DFA in code. State is an integer variable, δ is a switch statement Upon recognizing a lexeme returns its lexeme, lexical class and restart DFA with next character in source code.

CONTEXT FREE-GRAMMAR

Definition: Context-Free Grammar (CFG) has 4-tuple: G = (V, T, P,S)

Where,

V -A finite set of variables or*non-terminals*
T -A finite set of *terminals* (V and T do not intersect)
P -A finite set of *productions*, each of the form A –>α,
Where A is in V and α is in (V U T)*
Note: that α may be ε.
S -A starting non-terminal (S is inV)

Example :CFG:

G = ({S}, {0, 1}, P, S) P:
S–>0S1 or just simply S –> 0S1 |ε
S –>ε

ExampleDerivations:

S	=> 0S1	(1)
S	=> ε	(2)
	=> 01	(2)
S	=> 0S1	(1)
	=> 00S11	(1)
	=> 000S111	(1)
	=> 000111	(2)

- Note that G “generates” the language $\{0^k 1^k \mid k>=0\}$

Derivation (or Parse) Tree

- **Definition:** Let G = (V, T, P, S) be a CFG. A tree is a derivation (or parse) treeif:
  - Every vertex has a label from V U T U{ε}
  - The label of the root isS
  - If a vertex with label A has children with labels X_1, X_2,…, X_n, from left to right, then

synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminals **<u>A –> X1, X2,…, Xn</u>**

<u>must be a production in P</u>

The first L stands for "Left-to-right scan of input". The second L stands for "Left-most derivation". The „1"

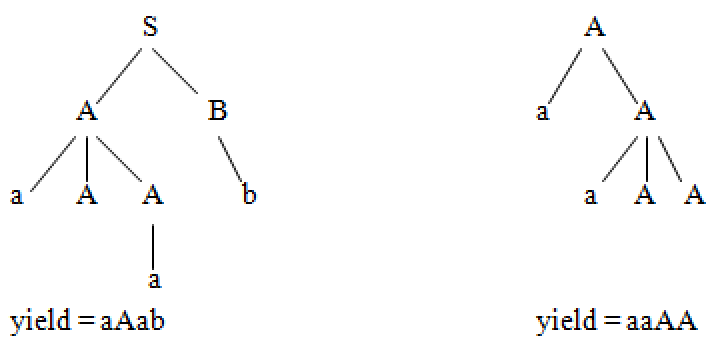
stands for "1 token of look ahead".

No LL (1) grammar can be ambiguous or left recursive.

LL (1) Grammar:

- If a vertex has label ε, then that vertex is a leaf and the only child of its"parent
- More Generally, a derivation tree can be defined with any non-terminal as theroot.
- **Example:**

S –> AB
A –> aAA
A –> aA
A –> a
B –> bB
B –> b



Notes:

- Root can be anynon-terminal
- Leaf nodes can be terminals ornon-terminals

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of

- A derivation tree with root S shows the productions used to obtain a sentential form.

LL(k)

LL(k) grammar performs a top-down, leftmost parse after reading the string from left-to-right
Here, kk is the number of look-aheads allowed.
With the knowledge of kk look-aheads, we
calculate FIRSTkFIRSTk and FOLLOWkFOLLOWk where:

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

- FIRSTkFIRSTk: kk terminals that can be at the beginning of a derived non-terminal
- FOLLOWkFOLLOWk: kk terminals that can come *after* a derived non-terminal

The basic idea is to create a lookup table using this information from which the parser can then simply go and check what derivation is to be made given a certain input token.

Now, the following text from here explains strong LL(k)LL(k):
In the general case, the LL(k)LL(k) grammars are quite difficult to parse directly. This is due to the fact that the left context of the parse must be remembered somehow.
Each parsing decision is based both on what is to come as well as on what has

already been seen of the input.

The class of LL(1)LL(1) grammars are so easily parsed because it is strong. The strong LL(kLL(k) grammars are a subset of the LL(k)LL(k) grammars that can be parsed *without* knowledge of the left-context of the parse. That is, each parsing decision is based only on the next k tokens of the input for the current nonterminal that is being expanded.
Formally,

A grammar (G=N,T,P,S)(G=N,T,P,S) is strong if for any two distinct A-productions in the grammar:
A→αA→α
A→βA→β
FIRSTk(αFOLLOWk(A))∩FIRSTk(βFOLLOWk(A))=∅FIRSTk(αFOLLOWk(A))∩FIRSTk(βFOLLOWk(A))=∅
That looks complicated so we"ll see it another way. Let"s take a textbook example to understand, instead, when is some grammar "weak" or when exactly would we need to know the left-context of the parse.

S→aAaS→aAa
S→bAbaS→bAba
A→bA→b
A→ϵA→ϵ
Here, you"ll notice that for an LL(2)LL(2) instance, baba could result from either of the SSproductions. So the parser needs some left-context to decide whether baba is produced by S→aAaS→aAa or S→bAbaS→bAba.
Such a grammar is therefore "weak" as opposed to being a strong LL(k)LL(k) grammar.

Unit-II

BOTTOM UPPARSING:
Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)
$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow - - - \Rightarrow r_{n-}$

$\longleftarrow$

$_1 \Rightarrow r_n \Rightarrow$ sentence Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce r_i r_{i-1} match some RHS β against r_i then replace β with its corresponding LHS, A.In terms of the parse tree, this is working from leaves to root.
Example – 1:

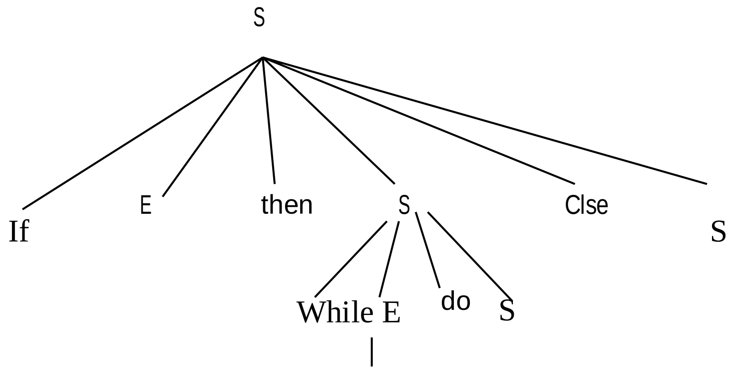
S→if E then S else S/while E do S/

print E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basicidea: Given input string a, "reduce" it to the goal (start) symbol, by looking for substring that match productionRHS.



true

$\underset{lm}{\Rightarrow}$ **if E then S elseS**

$\underset{lm}{\Rightarrow}$ **if id then S elseS**

$\underset{lm}{\Rightarrow}$ **if id then while E do S elseS**

$\underset{lm}{\Rightarrow}$ **if id then while true do S elseS**

$\underset{lm}{\Rightarrow}$ **if id then while true do print elseS**

$\underset{lm}{\Rightarrow}$ **if id then while true do print elseprint**

$\underset{rm}{\Leftarrow}$ **if E then while true do print elseprint**

$\underset{rm}{\Leftarrow}$ **if E then while E do print elseprint**

$\underset{rm}{\Leftarrow}$ **if E then while E do S elseprint**

$\underset{rm}{\Leftarrow}$ **if E then S elseprint**

$\underset{rm}{\Leftarrow}$ **if E then S elseS**

$\underset{rm}{\Leftarrow}$ **S**

HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

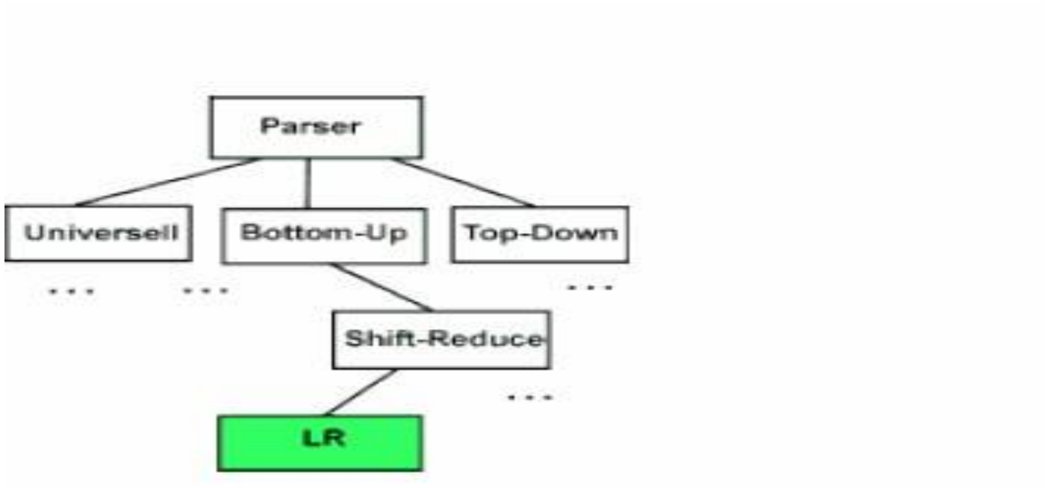
$E \rightarrow E+E/E*E/(E)/id$

Right-sentential form	Handle	Reducing production
a+b*c	A	$E \rightarrow id$
E+b*c	B	$E \rightarrow id$
E+E*C	C	$E \rightarrow id$
E+E*E	E*E	$E \rightarrow E*E$
E+E	E+E	$E \rightarrow E+E$
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us

LR PARSINGINTRODUCTION:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



WHY LR-PARSING:

1.LRparsers can be constructed to recognize virtually all programming-language constructs for which context-free grammarscan be written.

2.TheLRparsing method is the most general non-backtracking shift- reduce parsing method known, yetitcanbeimplementedas efficiently as other shift-reducemethods.

3.The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictiveparsers.

4,AnLR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of theinput.
The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this taskeasy.

LR-PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 and k=1. LR(1) is of practical relevance.

„L"stands for "Left-to-right" scan of input.

„R‟ stands for "Rightmost derivation (in reverse)".

K"standsfornumber ofinput symbolsoflook-a-head thatareusedin makingparsingdecisions.When (K) is omitted, „K"is assumed to be 1.
LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar.

A grammar is LR(1) if, given a right-most derivation

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 - - - r_{n-1} \Rightarrow r_n \Rightarrow$ sentence.

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol areleft.

LR(0)item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex– 1: $S^I \rightarrow .S\$$

$S \rightarrow .$

x $S \rightarrow .(L)$

Ex-2: $A \rightarrow XYZ$ generates 4LR(0) items

$A \rightarrow .XYZ$

$A \rightarrow X.$

YZ $A \rightarrow XY.$

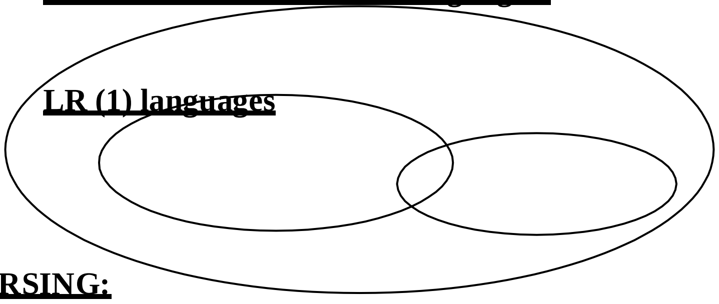
Z $A \rightarrow XYZ.$

$A \rightarrow XY.Z$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

→ **LR(0) items play a key role in the SLR(1) table constructionalgorithm.**

→ **LR(1) items play a key role in the LR(1) and LALR(1) table constructionalgorithms. LR parsers have more information available than LL parsers when choosing aproduction:**

* LR knowseverything derived fromRHS plus„K‟lookaheadsymbols.
* **LL just knows„K‟lookaheadsymbols into what‟sderived fromRHS.**
* **Deterministic context free languages:**
*
*
* **LR (1) languages**
*
*



LALR PARSING:

Example:

Construct C={I0,I1,… .... ,In} The collection of sets of LR(1)items

For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a singleterm)

I_0 $\rightarrow$same asprevious
I_1 $\rightarrow$ “
I_2 $\rightarrow$ “
I_{36} – Clubbing item I3 and I6 into one I36 item.
C $\rightarrow$cC,c/d/$
C$\rightarrow$cC,c/d/$
C$\rightarrow$d,c/d/$
$I_5$$\rightarrow$some as previous
$I_{47}$$\rightarrow$C$\rightarrow$d,c/d/$
$I_{89}$$\rightarrow$C$\rightarrow$cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d	$	S	C
I_o	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		

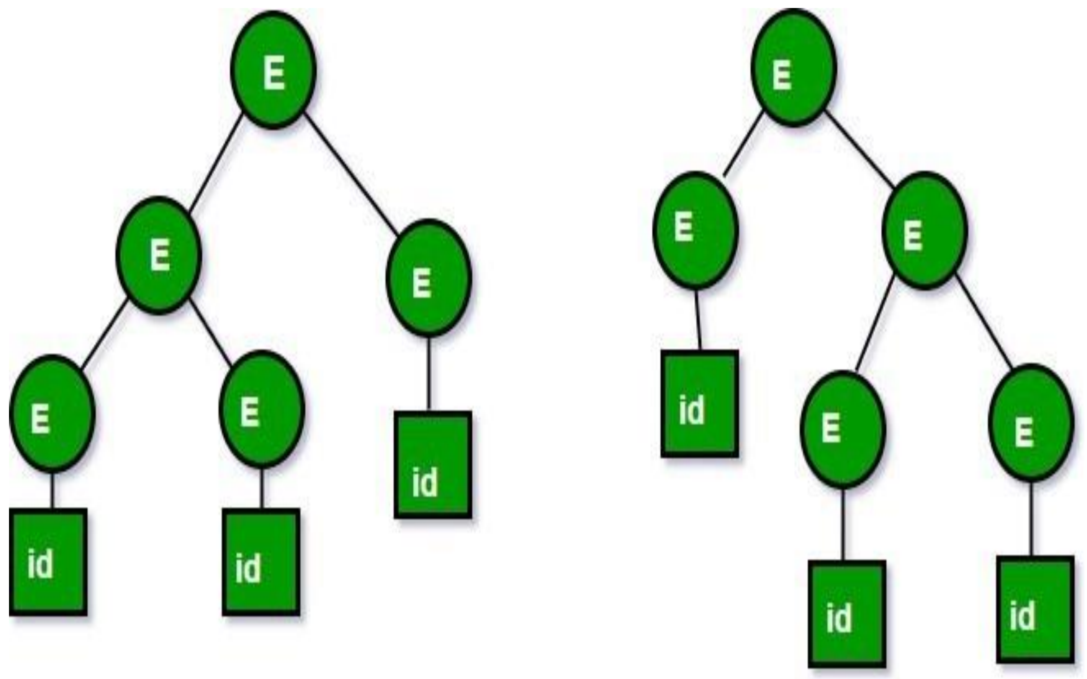
Ambiguous grammar:

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft**M**ost **D**erivation **T**ree (LMDT) or **R**ight**M**ost **D**erivation **T**ree (RMDT).

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than on parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

For Example:

1. Let us consider this grammar : **E ->E+E|id**
We can create 2 parse tree from this grammar to obtain a string **id+id+id** :
The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

YACC PROGRAMMING

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:
YACC input file is divided in three parts.

```
/* definitions */

 ....


%%

/* rules */

....

%%


/* auxiliary routines */

....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

```
%token ID
```

- Yacc automatically assigns numbers for tokens, but it can be overridden by

```
%token NUMBER 621
```

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{**and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:

```
%start nonterminal
```

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

- YACC input file generally finishes with:

```
 .y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Semantics

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semanticrules

Definition of (syntax Directed definition) SDD :

- SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

A syntax-directed definition is a generalization of a context-free grammar inwhich:

- Each grammar symbol is associated with a set ofattributes.
- Thissetofattributesforagrammarsymbolispartitionedintotwosubsetscalledsynthesized and inherited attributes of that grammar symbol.
- Each production rule is associated with a set of semanticrules.
- Semantic rules set up dependencies between attributes which can be represented by a dependencygraph.
- This dependency graph determines the evaluation order of these semanticrules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing avalue.

<u>The two attributes for non terminalare :</u>

The two attributes for non terminalare :

Synthesized attribute (S-attribute) :(↑)

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

Inherited attribute:(↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of thatnode.

- The attribute can be string, a number, a type, a, memory location or anythingelse.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.Terminals can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

• A parse tree showing the values of attributes at each node is called an Annotated parsetree.

• The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.

• Of course, the order of these computations depends on the dependency graph induced by the semanticrules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :

S→ EN

E→E+T

E→E-T

E→ T

T→ T*F

T→T/F

T→F

F→(E)

F→digit N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production

Productionrule	**Semanticactions**
S→EN	S.val=E.val
E→E1+T	E.val =E1.val +T.val
E→E1-T	E.val = E1.val –T.val
E→T	E.val=T.val
T→T*F	T.val = T.val *F.val
T→T|F	T.val =T.val | F.val
F→ (E)	F.val=E.val
T→F	T.val=F.val
F→digit	F.val =digit.lexval
N→;	can be ignored by lexical Analyzer as;I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute “Val”.

The taken digit has synthesized attribute “lexval”.

In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

Write the SDD using the appropriate semantic actions for corresponding production rule of the givenGrammar.

The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom upmanner.

The value obtained at the node is supposed to be final output.

L-attributed SDT

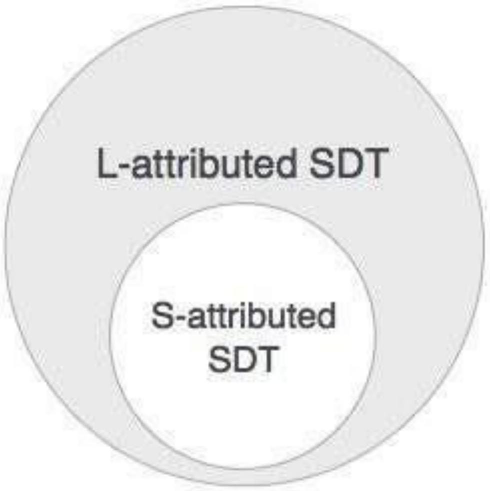
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

```
S → ABC
```

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

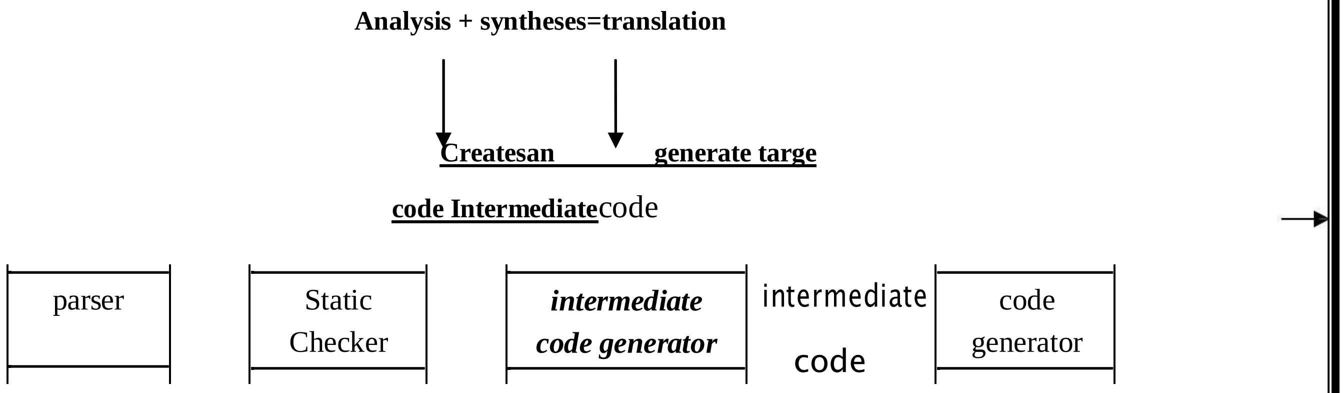
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions

Intermediate Code

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an objectmodule.



In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.

We assume that the source program has already been parsed and statically checked..the various intermediate code forms are:

a) Polishnotation
b) Abstract syntax trees(or)syntaxtrees
c) Quadruples
d) Triples three address code
e) Indirecttriples
f) Abstract machinecode(or)pseudocopde

postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, asab+.

In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfixexpression.

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semanticrules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)
Where a is an attributes obtained from the function f.

• A syntax-directed definition is a generalization of a context-free grammar inwhich:
• Each grammar symbol is associated with a set ofattributes.

Thissetofattributesforagrammarsymbolispartitionedintotwosubsetscalledsynthesized
and inherited attributes of that grammar symbol.

• Each production rule is associated with a set of semanticrules.

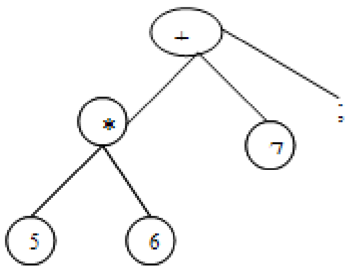
• Semantic rules set up dependencies between attributes which can be represented by a dependencygraph.

<u>Annotated Parse Tree</u>

• A parse tree showing the values of attributes at each node is called an Annotated parsetree.

• The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.Of course, the order of these computations depends on the dependency graph induced by the

Syntax tree:



Annotated parse tree :

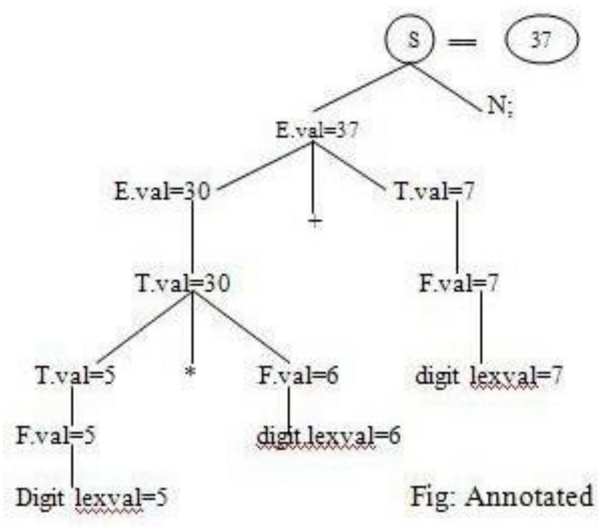


Fig: Annotated parse tree

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar. P →M D

M → ε
D →D ; D | **id** : T | **proc id** ; N D ; S
N → ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

```
S →id := E        { p : = lookup ( id.name);
                    if p ≠ nil then
                    emit( p ' : =' E.place)
                    else error }

E → E1 + E2       { E.place : = newtemp;
```

emit(E.place ': =' E_1.place ' + ' E_2.place) }

E → E_1 * E_2 { E.place : = newtemp;
emit(E.place ': =' E_1.place ' * ' E_2.place) }

E →- E_1 { E.place : = newtemp;
emit (E.place ': =' 'uminus' E_1.place) }

E →(E_1) { E.place : = E_1.place }

E → id { p : = lookup (**id**.name);
ifp≠nil **then**
E.place : = p
elseerror }

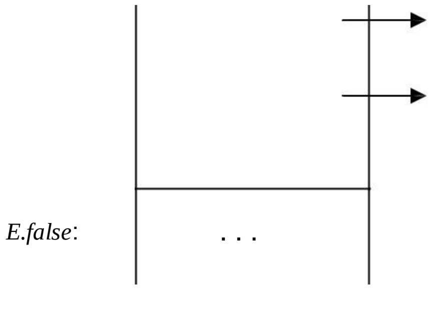
Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

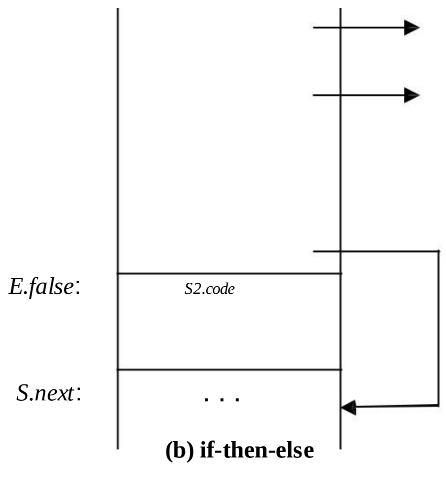
S **if** E **then** S1
if E **then** S1 **else**
| S2
| **while** E **do** S1

In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel*returns a new symbolic label each time it is called.

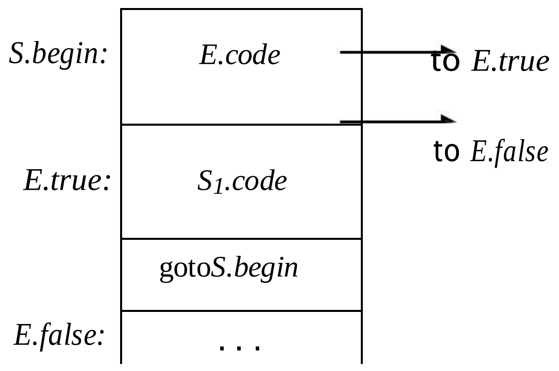
- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.nextis a label that is attached to the first three-address instruction to be executed after the code for **Code for if-then , if-then-else, and while-do statements**



(a) if-then



(b) if-then-else



(c) while-do

PRODUCTION	SEMANTIC RULES

$S \rightarrow$ **if** *E* **then** S_1	*E.true : = newlabel;*
	E.false : = S.next;
	S_1*.next : = S.next;*
	S.code : = E.code || gen(E.true „:") || S_1*.code*

$S \rightarrow$ **if** *E* **then** S_1**else** S_2	*E.true : = newlabel;* *E.false : = newlabel;* S_1*.next : = S.next;* S_2*.next : = S.next;* *S.code : = E.code || gen(E.true „:") ||* S_1*.code ||* *gen(„***goto***" S.next) ||* *gen(E.false „:") ||* S_2*.code*

$S \rightarrow$ **while***E* **do** S_1	*S.begin : = newlabel;*
	E.true : = newlabel;
	E.false : = S.next;
	S_1*.next : = S.begin;*
	S.code : = gen(S.begin „:")|| E.code ||
	gen(E.true „:") || S_1*.code ||*
	*gen(„***goto***" S.begin)*