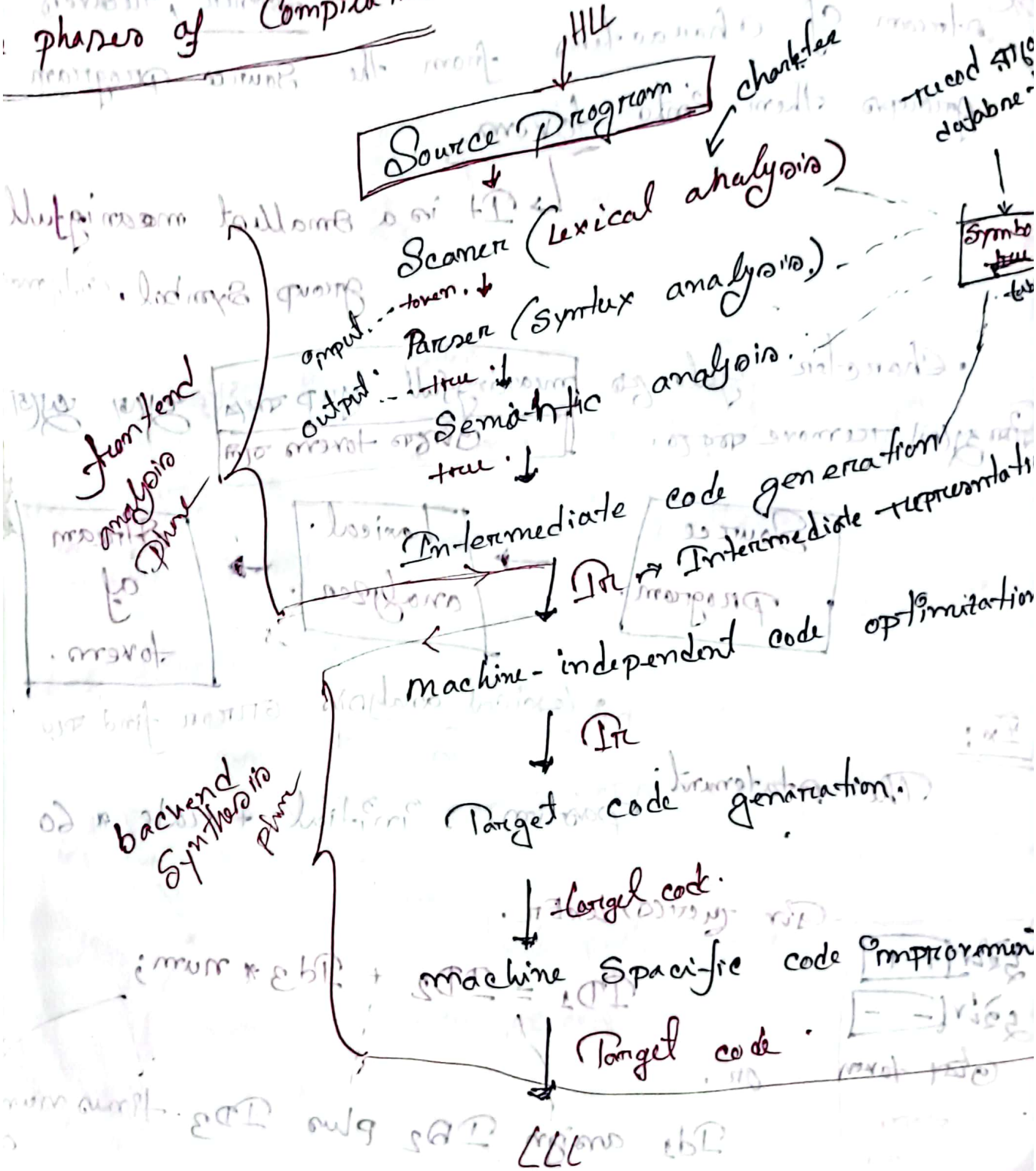
Bismillah hir Rahmanir Rahman.

Compiler >> Source program → Compiler → Target program (HLL → LLL)

phases of Compilation.



Scanner (Lexical analysis)

> " " ভিতরে যা থাকে তা একটা token.
> //, comment, whitespace বাদে all token

* It also called a Lexer or a Scanner, receives a stream of characters from the source program and groups them into tokens.
  ↳ It is a smallest meaningfull group symbol. (int, main, ...)

• Character গুলো কে meaningfull grp করে টোকেন tokens বলে — স্পেস গুলো [illegible]. ফাঁকা গুলো remove করে।

Source program → Lexical analyzer → Stream of tokens.

• Lexical analysis error find করে

Ex:

The statement: position = initial + rate * 60

এটা কে বলে, lexer.

$ID_1 = ID_2 + Id_3 * num;$

or,

Id_1 assign ID_2 plus ID_3 times num

> দুইটা + +
> দুইটা - -
> একটা token
> += একটাই token

Lexical analysis Tools.

- Lex → C Source Code. (UNIX)
- flex → " " "
- JLex → Produces Java code
- JFlex → " " " "

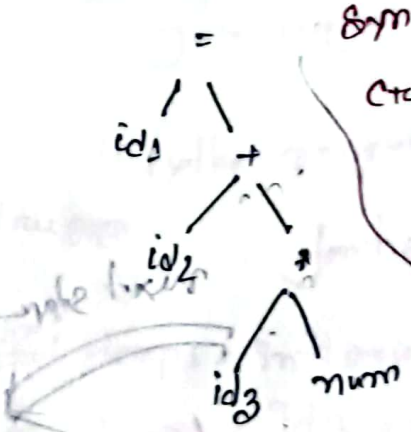
Syntax analyzer (parser)

A syntax analyzer also called parser, receives a stream of tokens from the lexer and groups them to phrases that match specified grammatical pattern.

described by context free grammar

$tx : id_1 = id_2 + id_3 * num$

↓ Syntax tree create করে Syntax analyzer



Stmt → id = expr ;

expr → expr + expr | expr * expr | id | num

[grammatical pattern গুলো গুলো find, তারপর tree generate করে].

Syntex analyzer Tools.

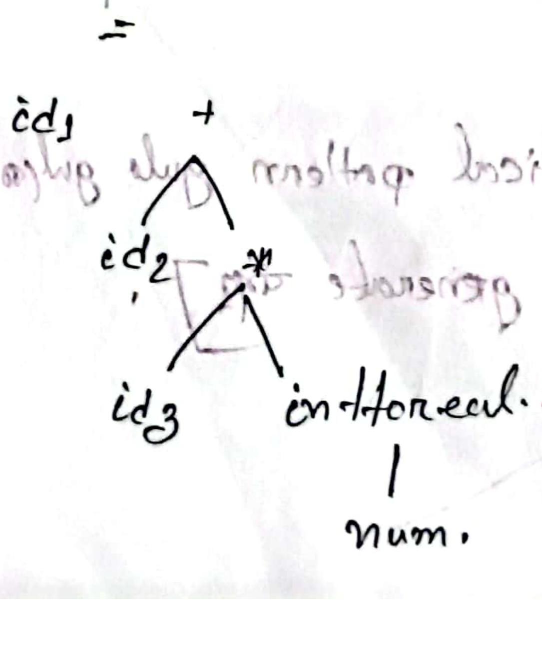
- Yacc → C Source code (UNIX)
- bison → C " " (gnu)
- CUP → Java Source code.

Semantic Analysis.

Semantic analyzer traverses the abstract (সারাংশ) tree, checking that each node is appropiate for context. It checks for Semantic errors. Its outputs refinded abstract syntex tree.

[By Syntax error and Semantic error remove হয় কেন]

Example: $id_1 = ID_2 + id_3 * Num$



← type casting opreator (real number convert)

next step

Intermediate code genaration.

Intermediate code is code that represents the semantics of a program but is machine-independent.

(সবাই use করতে পারবে Intermediate code use করলে
কিন্তু use ভিন্ন, কোন, যেকোনো যাইহোক use হবে].

Abstract Syntax Tree → intermediate code generator → intermediate code.

প্রথম Intermediate code genarate করি তারপর

C program →
Java program. → Intermediate code → x86 code.
Python program. → Intermediate code → MIPS 82 code.

front end (language specific)

Intermediate code ↑ তারপর এটার Language নিয়ে ভাবতে হবে না। তারপর যেভাবে দরকার তাই এটার genarate করলাম.

Back end (machine)

tree থেকে ↓

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

Code optimizer .

• optimize করার এক উদ্দেশ্য হচ্ছে টাইম বা, memory requirement reduce করা।

[একই Line code এর মধ্যে float না নিয়ে 60.0 নিলে 2 line হয়ে যায়।

• An optimizer reviews the code, looking for ways to reduce the number of operations and memory requirement

• optimized for speed or size .

Example: the intermediate code in may be optimized as.

$$temp1 = id3 * 60.0$$

$$id1 := id2 + temp1$$

Machine Code generation.

- The code generation receives the (optimized) intermediate code
- It produces
  - Machine code for specific machine
  - Assembly code for a specific assembler

If it produces assembly code, then assembler is used to produce the machine code.

Exp: the inter

$t_1 = ID_3 * 60.0$
$id_1 = id_2 + t_1$

↓

```
movf  id3, R2
mulf  #60.0, R2
movf  id2, R1
addf  R2, R1
```

} IR → assembly code

Machine code ≈ assembly code

intermediate code → Code generator

↳ Assembly code → Assembler

↳ Machine Code

Symbol table

Records <variable name, attributes>,

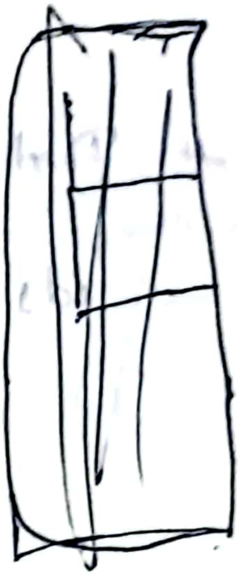
- তাহার মানে হচ্ছে সব কিছু Lexical, syntax, semantic এর ভেতর গুলোকে store করে রাখে, [records gula save রাখে].

Attributes:

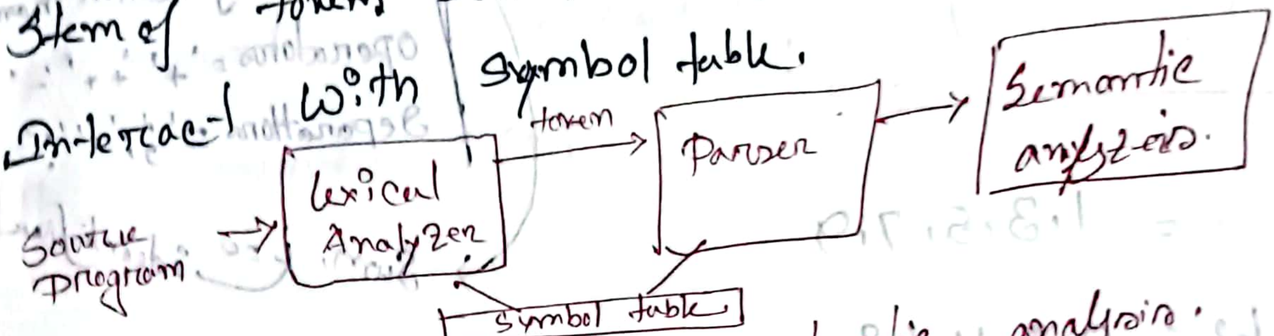
- Storage information.
- type.
- Scope
- Procedure names
- argument parsing.
- return type.

Design principle: find, store, retrieve, record quickly.

Tasks of Lexer / Scanner (কাজ)



- read the input char of the source program.
- group them into lexemes.
- produce as output a sequence of tokens for each lexeme in the source program.
- Stem of tokens sent to the parser.
- Interact with symbol table.



Lexical and syntactic analysis.

why Separating

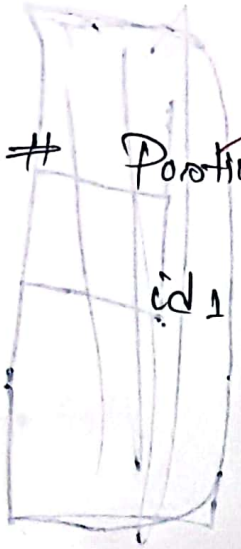
- Simplicity of design is the most important consideration.
- efficiency is improved.
- portability ~~enhaced~~ enhanced.

(for pattern o'
for store with O(N))

Lexeme : lexme কে pattern আসে Match হলে token.
" " না " invalid token.

Scaning : comment white space delete করে.

Lexical analysis : তার produces করে (token from scanner.



Symbol table stone 24 এগুলো.

Position = initial + rate * 60 // lexeme.

$id_1 = id_2 + id_3 * 60$ // token.

keywords = for, while, if
identifier = Variable name, function name
operators = '+' '++' '-'
separations = ';' ',' '('

Sum, i=0 ; identifier

Operations of Language:

$L_1 = 1, 3, 5, 7, 9$

$L_2 = 0, 2, 4, 6, 8$

- $L_1 \cup L_2 = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
- $L_1 L_2 = 10, 12, 14, 16, 18, 30, 32, 34, 36, 38$

concatation L_1 L_2 এর প্রথম digit

- $(L_1 \cup L_2)^3 = \{000, 012, 034, 123\}$ union এর length 3.
- $L_1{}^3 = 111, 333, 555, 135$ length 3 এর at least
- $(L_1 \cup L_2)^* L_2 = 0, 02, 04, 06, 08, 00, 02, 04, 06, 08, \ldots$

$= (0,1,2,3,4,5,6,7,8,9)^* (0,2,4,6,8)$ L_2 এর সাথে

Cont 0, 2, 4, 8, 6 আকারে

Regular Expression:

- r|s represents union.
- rs represents concatenation. L(r) L(s)
- r* represents more replaced 0 L(r)*
- r represents L(r)

Example;

r = A|B --- |Z a|b --- |z / letter

s = 0|1 --- |9. / digit.

t = r(r|s)*

r concatenation (r union s) ∈ replaced concatenation

id → letter (letter |digit)*

↑

t identifiers follows this.

digit+ means 0 or more digit.

- $\pi^+ = \pi \cdot \pi^*$ ← Zero or more instance (0 বার বা তার বেশি)
- $\pi? = \pi \mid E$ ← Zero or one instance (শূন্য বা একবার)

floating point number.

[? = optional বোঝাতে]

digit → [0-9]

digit ⇒ digit +

Number → digit (. digits)? E[+,-]? digits)?

digit এরপর . থাকলে digits থাকতেও পারে

10^ (+,-) থাকলে এরপর digits থাকতে হবে

7) class tokens

digit - 0-9

digits → digit+

number → digits (. digits)? (E [+-]? digits)?

letter → [A-Za-z].

id → letter (letter|digit)*

if → if.

then → then

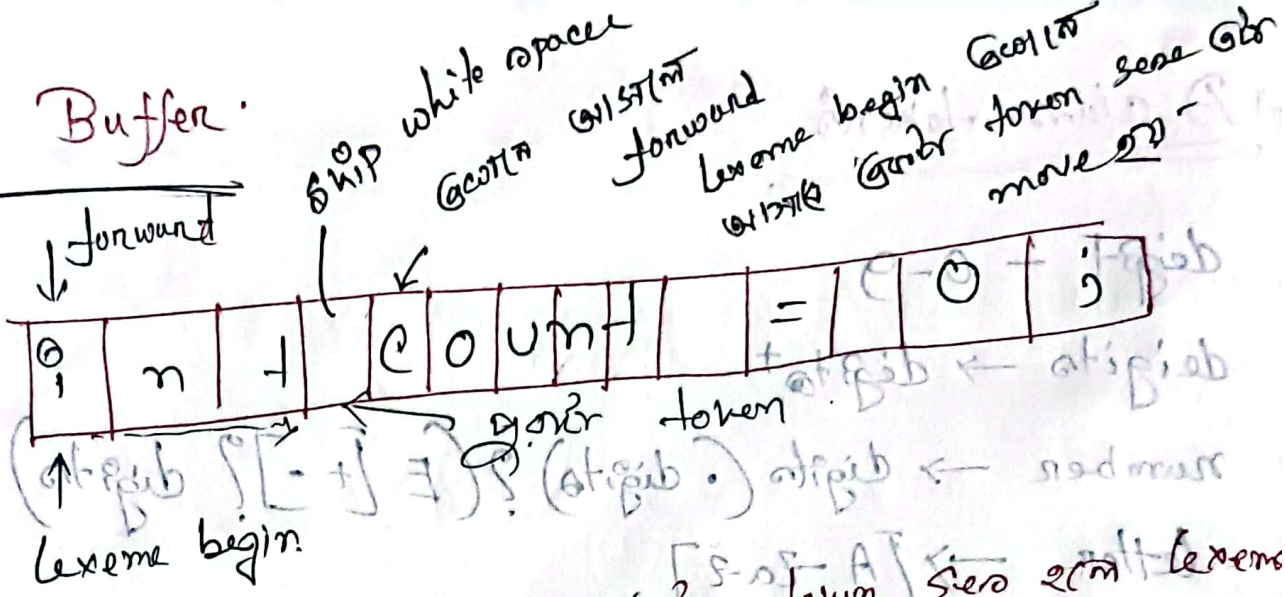
else → else

relop → <|>, <=|>=, =|<>.

white space : WS → (blank | tab | newline)+.

- whitespace is not a token, but it must be recognized by the lexer.
- lexer remove whitespace from the input

#Input Buffer.



Buffer pairs : a two buffer scheme that handles large lookaheads safely

Sentinels : Save time checking for the ends of Buffers. (Semicolon, eof)

We have two portion of Buffer.

- Lexeme begin : holds the starting position of the current token.

forward Pointer : Points to the current symbol.

Transition Diagrams.

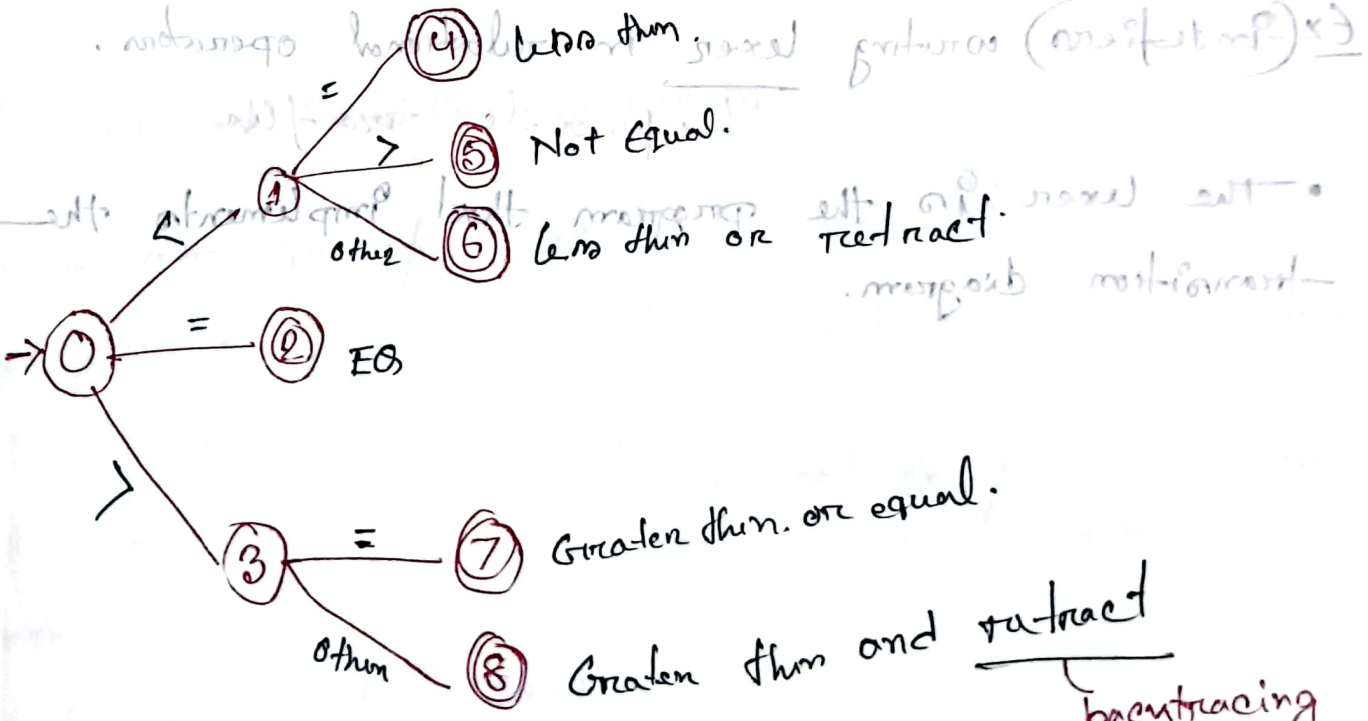
- Transition Diagram is a directed graph.
- It consists of a finite set of nodes called States. One of them start state
- Set of the states are called accepting states. The remaining states is rejecting states
- The directed edges between states are transitions

Example (Relational operators).

<, >, <=, >=, =, <>

if first symbol is < next state = 1, 2 or else.

" " " > " " = , < " " .

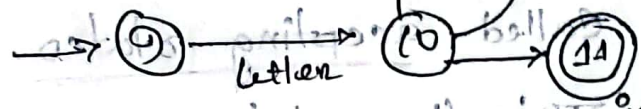


Transition Diagram.

~~Ex (relational operators)~~

Example (identifier)

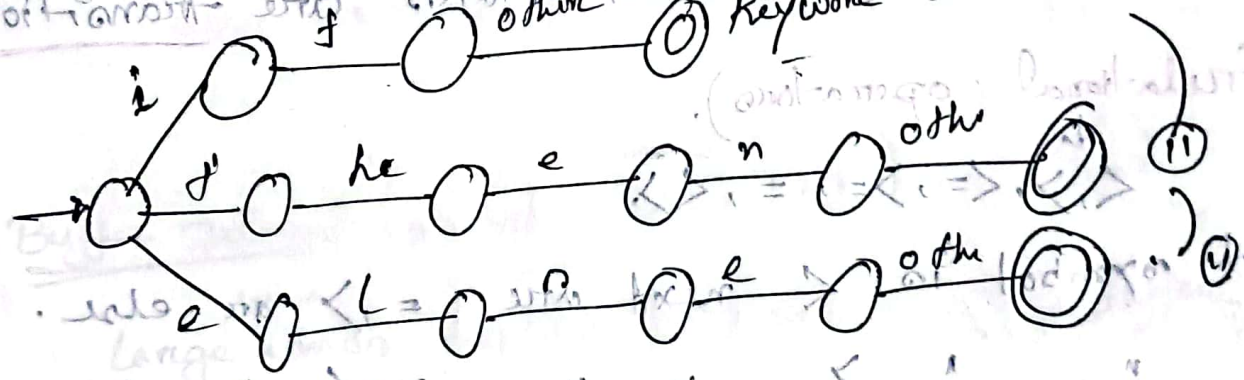
letter | digit



retract next char

identifier and retract

Example (Keywords)



~~Ex (identifier)~~ writing lexer for relational operators.

↳ generate the flex

- The lexer is the program that implements the transition diagram.

Grammar & Expressions (N. Draft Sig)

✓

Expr => Expression

Stmt => Statement

- Such rules is called <u>Production</u>.
- In a production lexical elements like the keyword if and the parentheses are called <u>terminals</u>
- <u>variables</u> like expr and stmt sequences of terminals called non terminals.

S => cAd.
A → λ/ab

} here a b, c, d are (terminals)
S, A are (non - terminals.)

~~CFG~~ Context Sensitive grammers (CSG):

CSG is more powerful than CFG, and It has 4 tuples

G = N, Σ, P, S

N = Set of non terminals

Σ : terminal.

P = production

S. = Start Symbol of the production.

CSG is Left sides and R8 hand of on any poduction rules may be Surrounded by context of terminal and non terminal Symbols.

CFG

4 components.

1. A set of terminals symbol referred as tokens.
2. A set of Non-terminals. " " syntactic variable
3. A set of Productions (P/R).
4. Start Symbol (S)

digits separated by (+, -).

• 9-5+2 ⇒ Syntax of Expressions:

List → List + digit
List → " - "
List → digit
digit → 0|1|2|3|4|5|6|7|8|9

non terminal.

terminals. (+,-) sep

all the Production $A \to Ba$ $A \to \alpha$ Left Linear gram

$A \Rightarrow Aa \mid Ba \mid b$

all the production $A \to \alpha B$ or $A \to \alpha$ then Right

$A \Rightarrow aA \mid aB \mid b$

[terminal ... RS ... Left]

Derivation Tree / Parse tree

- graphically represents the semantic information.

properties of parse tree:
- Root - start symbol.
- Interior nodes - non terminal symbol.
- leaves - terminal

- Top down approach to draw derivation tree.

Top down Approach:
- Starts with starting Symbol S.
- Goes down to tree leaves using Productions.

Bottom up: → terminal → start → draw

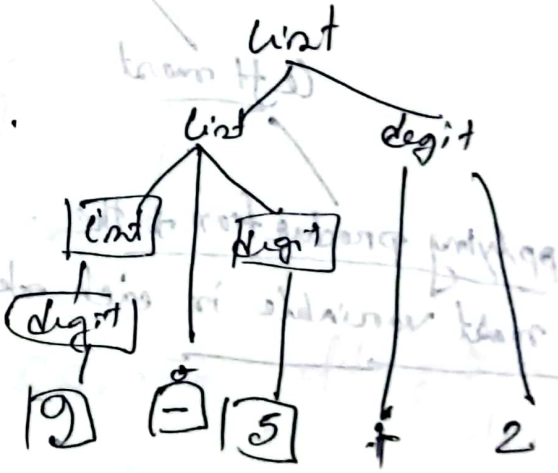
- Starts with leaves
- upward to the root which is the starting symbol S

upon (on)

* derivation of 9 - 5 + 2

list → list + digit / list - digit / digit.

list → list ± digit.
→ list ± digit + 2
→ list ± 5 + 2
→ digit ± 5 + 2
→ 9 + 5 + 2.



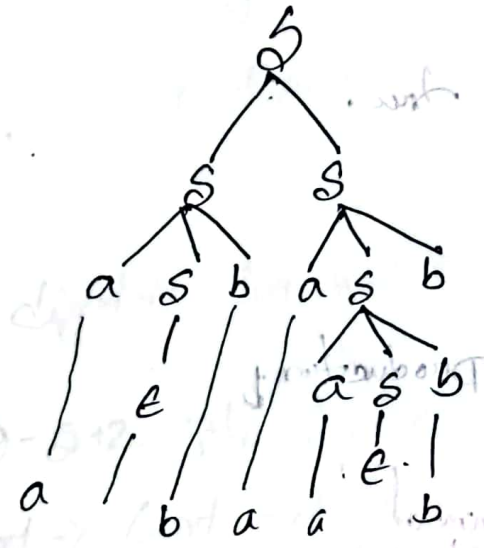
Example

CFG (N, T, P, S) where

$N = \{S\}$, $T = \{a, b\}$ $P = S \to SS \mid asb \mid \epsilon$ $S \Rightarrow S$

Non-terminal Terminal production

abaabb



$S \to SS \to$

2 Generation of tree

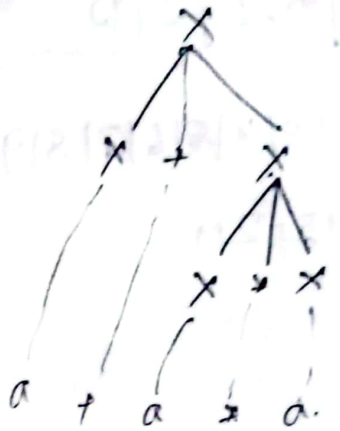
Left most

applying production to the left most variable in each step

Right most

by applying production to the Right most variable in each step.

$X \to X+X \mid X*X \mid X \mid a$.



Left most (a + a*a)

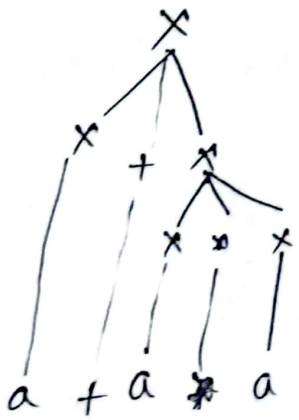
$X \to X+X$
$\to a+X$
$\to a + X*X$
$\to a + a*a$

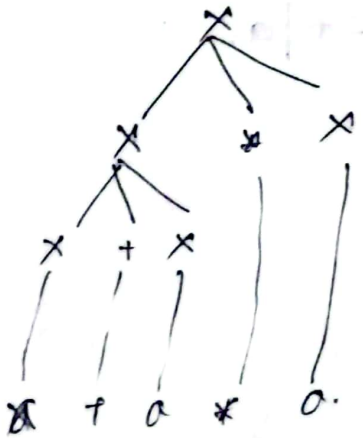
Ambiguity.

If CFG has more than one derivation tree for some string called Ambiguous grammar. (Ex. Left derivation etc.)

$X \to X+X \mid X*X \mid X \mid a$

(a + a*a)





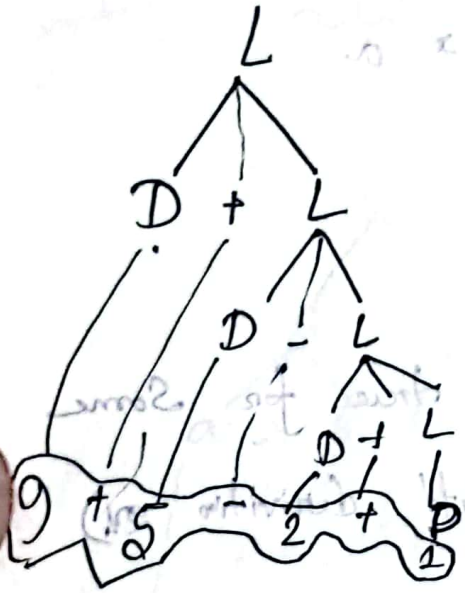
So, to define RE, and left most tree, this is ambiguous.

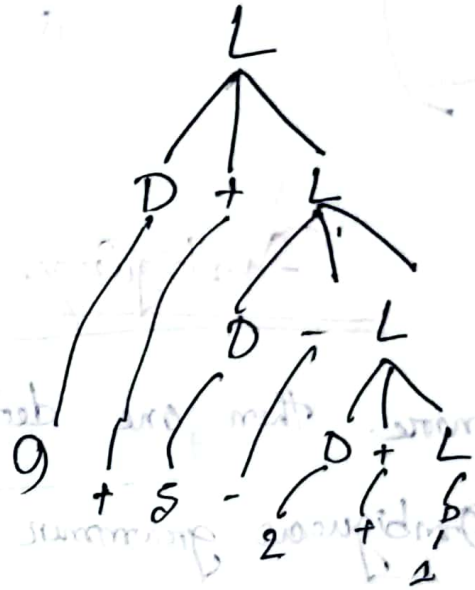
$L \to D+L \mid D-L \mid L \mid D$

$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$

String → 9+5-2+1

9+5-2+1





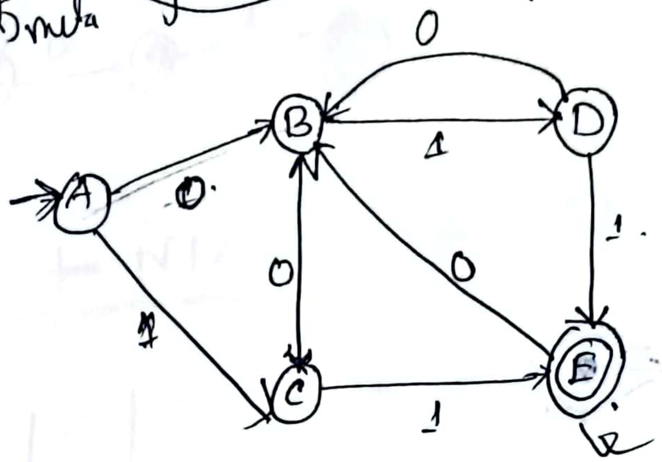
So this am = bigguo

~~$X \to X+a \mid X-a \mid a.$~~

Seg-8

Minimization of DFA

Automata follows a pattern



	0	1
→ A	B	C
B	B	D
C	B	C
D	B	(E)
(E)	B	C

0 equivalent : {A, B, C, D}, {E} ← final state আলাদা হবে

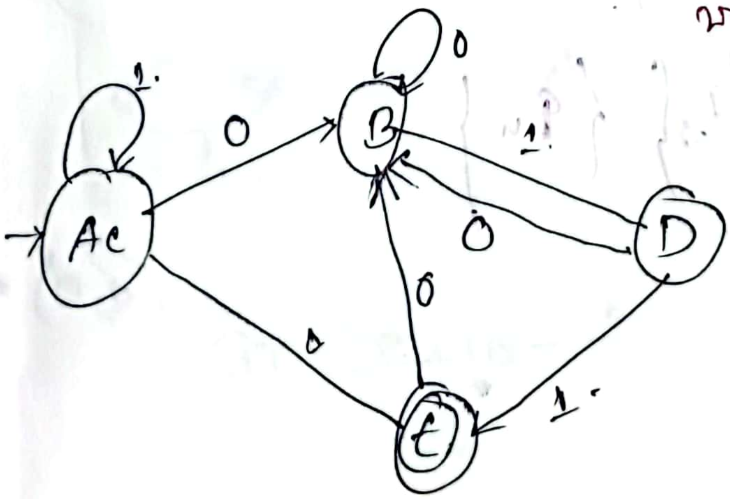
1 Equivalent : {A, B, C}, {D}, {E} ← A, B, C এর বেলায় B, B আলাদা

2 Equivalent : {A, C} {B} {D} {E} Same আসে 1 এর C, D থাকে

3 equivalent : {A, C} {B} {D} {E}

C D 3 থেকে আলাদা so এখন C, D মধ্যে D এর E থাকে, নাই so alone

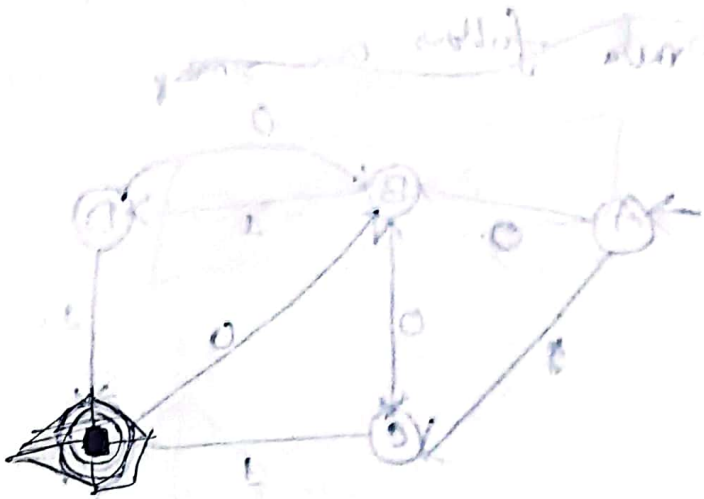
Same A, C এর মধ্যে B, C BC আলাদা আর এর মধ্যে আলাদা



	0	1
(AC)	B	C
B	B	D
D	B	E
(E)	B	C

Q. - AU 22 — 3(b)

	0	1
→ q_0	q_1	q_3
q_1	q_2	q_4
q_2	q_1	q_4
q_3	q_2	q_4
(q_4)	q_4	q_4



0 equivalent ⇒ $\{q_0, q_1, q_2, q_3\}\{q_4\}$

1 equivalent ⇒ $\{q_0, q_2, q_3\}\{q_1\}\{q_4\}$

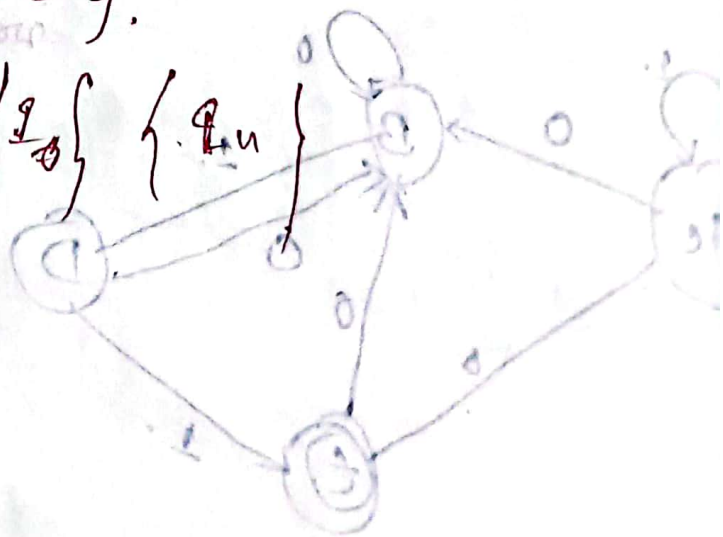
2 equivalent ⇒ $\{q_0, q_3\}\{q_2\}\{q_1\}\{q_4\}$

0 equivalent ⇒ $\{q_0, q_1, q_2, q_3\}\{q_4\}$

1 equivalent ⇒ ~~$\{q_0\}\{q_1, q_2, q_3\}$~~

⇒ $\{q_0, q_2, q_3\}\{q_1\}\{q_4\}$

	0	1
$q_0 q_2 q_3$	q_2	q_4
q_1		
q_4		



- Ignore number of states NFA
(Exact state which the machine moves can't determined)

NFA

$Q = \{a, b, c\}$ - ~~terminal~~ Non terminal

$\Sigma = \{0, 1\}$ - terminal.

Initial $q_1 = \{a\}$

Final $F = \{c\}$.

DFA	NFA
The transition from state is to single particular next state for each input symbol. [1 to one 1 to one]	The transition state can be to multiple next state for each input symbol.
* empty string transitions Not seen	* ~~NFA~~ Empty string transitions are seen here
• Backtracking is allowed	• Back tracking not possible
• requires More space	• Requires less space.