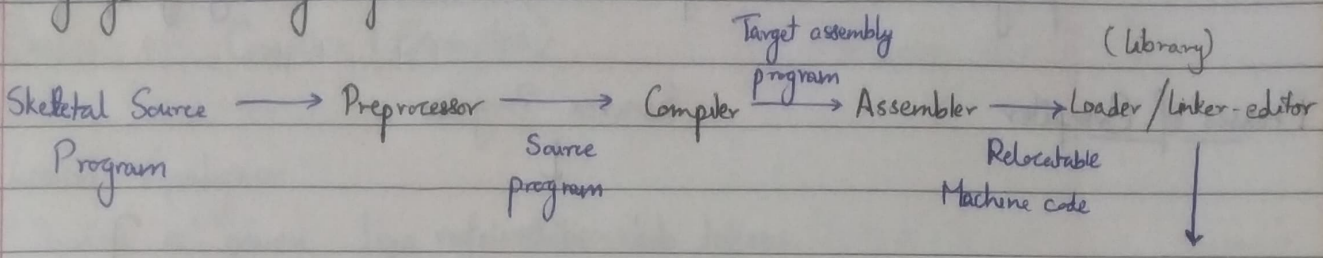
Date : __/__/__

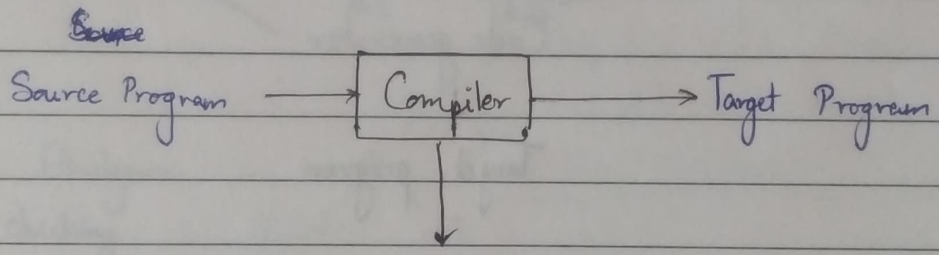
UNIT 1

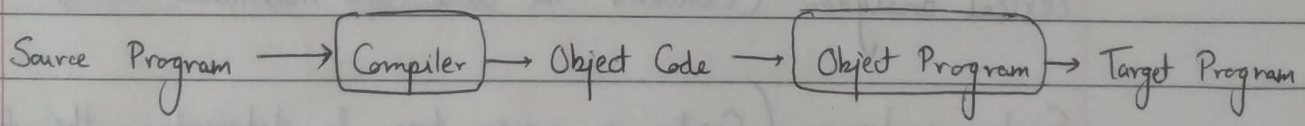
- Language Processing System.



1. Preprocessor
   a. Macro Preprocessing (work on macros #def)
   b. file inclusion Preprocessing (work on different file formats; image, video, link to others)
   c. Rational Preprocessing (synchronization of different older languages to modern control)
   d. Language extension (adding capabilities or functions to a language)

Compiler





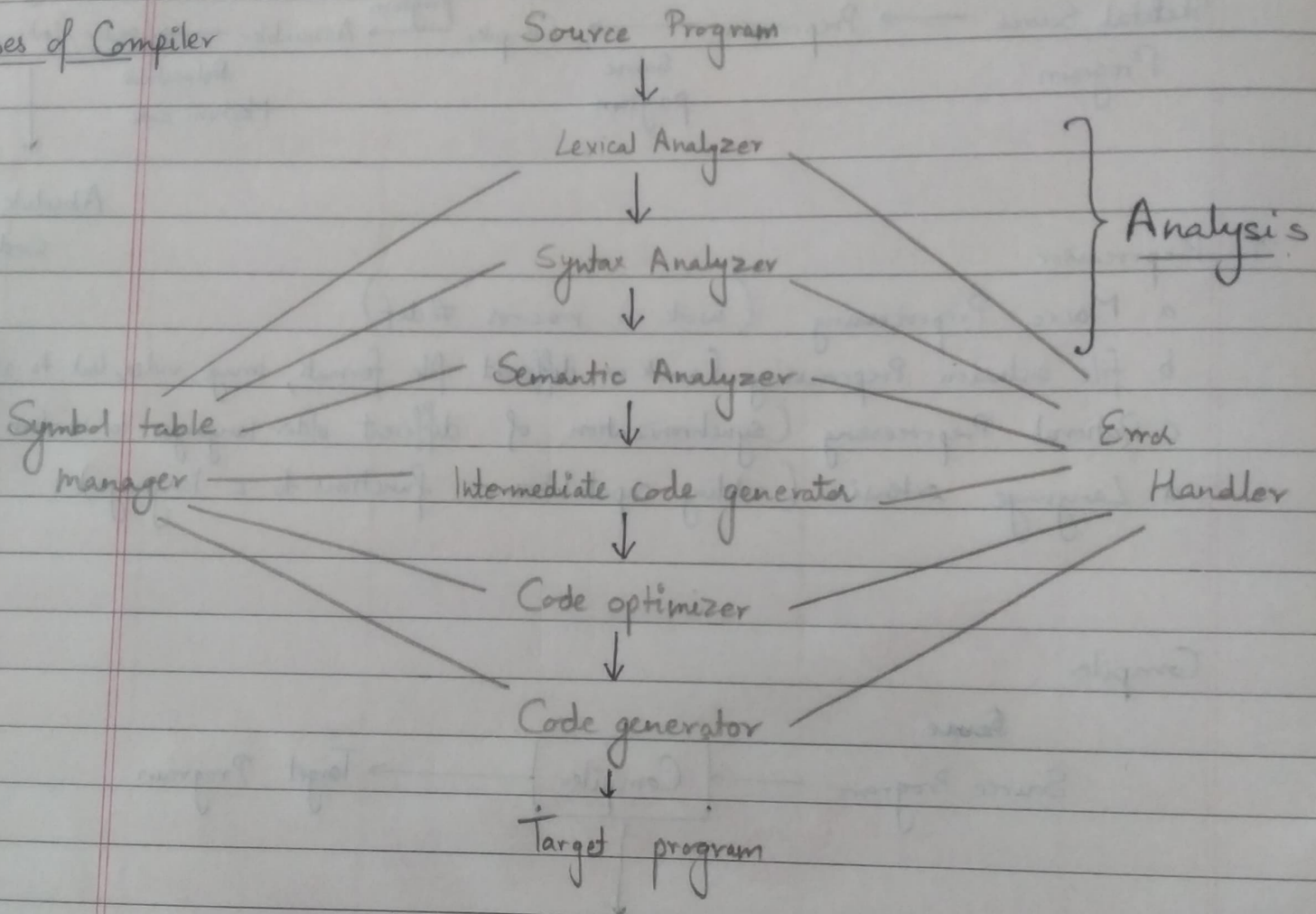
* Interpreter
- Checks the program line by line.
- Easily to modify & identify errors
- Makes the code machine independent
- ↑ execution time
- ↑ memory consumption

Date : __/__/____

Compiler

A translator program which converts a high-level language written program into an equivalent machine level language program.

Phases of Compiler



Lexical analyzer (Converts the code into individual tokens)

Syntax analyzer (Create a parser tree & determine the tokens to be grouped, to make a valid context)

~~Syntax~~ Semantic analyzer (relate syntactic structures to their language independent meanings)

Compilation process → Analysis → Synthesis

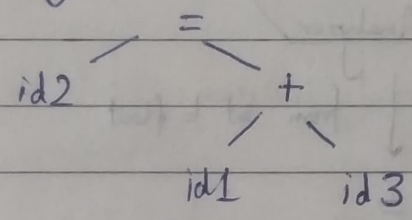
* Phases of Compiler (Examples)

1. Lexical Analyzer
   - Divide a given line of codes into tokens.

$$Z = A + B$$

identifier 2 = id1 + id3

2. Syntax Analyzer
   - Combine the tokens to generate a valid expression.
   - Create a parsing tree



3. Semantic Analyzer
   - Type checking

4. Intermediate Code Generation

5. Code optimization
   - Optional
   - Techniques: 1. Local optimization

if A>=6 goto L1
L1: goto L3
L3: print(" ");

→

if A>=6 goto L3

2. Subsequence optimization

A = B+C+D
E = B+C+F

→

$T_1 = B + C$
$A = T_1 + D$
$E = T_1 + F$

(Saves time)

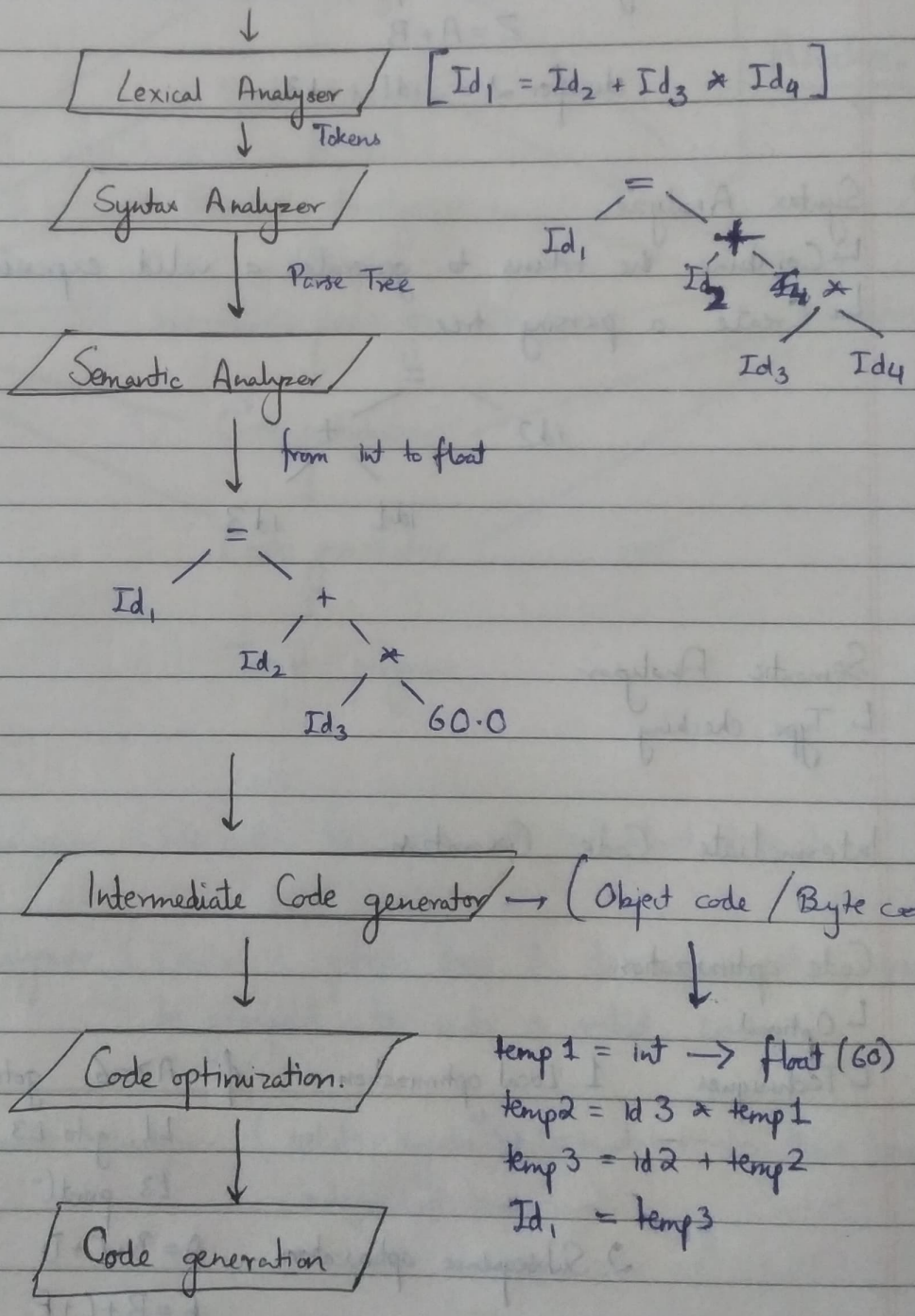
CP

Date : __/__/__

3. Loop optimization // Remove any unnecessary calculations or operations from within the loop

6. Code generation.

Example: $x = y + z * 60$



UPCP

Date : __/__/____

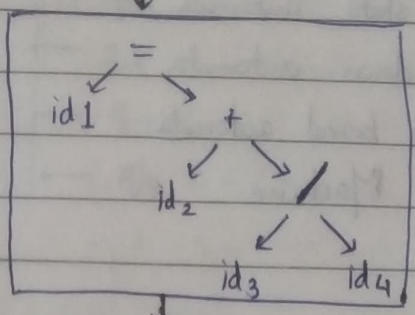
Ex2: $p = q + r / 100$

↓

Step 1: Lexical Analyzer → $id_1 = id_2 + id_3 / id_4$

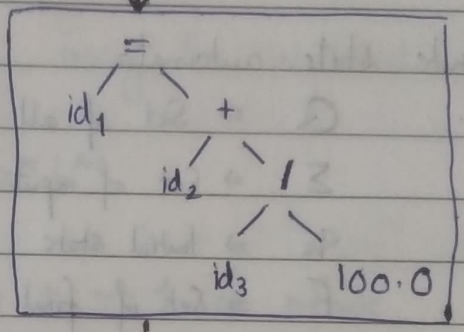
↕

Step 2:- Syntax Analyzer



↓

Step 3:- Semantic Analyzer



↓

Step 4:- Intermediate Code generator

```
temp 1 = int → float (id4)
temp 2 = id3 / temp 1
temp 3 = id 2 + temp 2
id1 = temp 3
```

↓

Step 5:- Code optimization

```
temp 1 = id3 / 100.0
id1 = id2 + temp 1
```

↓

Step 6:- Code generator → Corresponding machine language code

* finite state automata
* Grammar & its types
* Ambigious & unambigious grammar

Automata

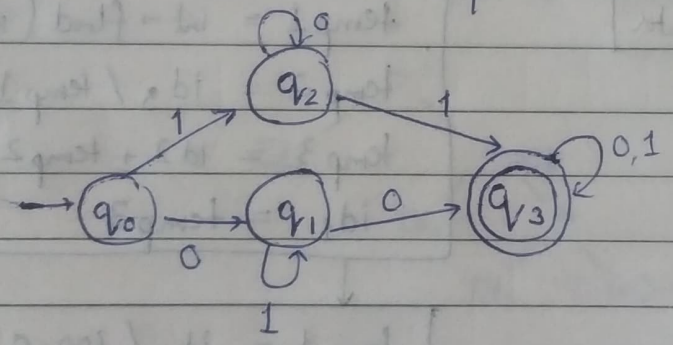
↳ A machine which takes input from user to produce an output

finite state automata	→	Regular Grammar
push down automata	→	Context free Grammar
Linear bound automata	→	Context sensitive Grammar
Turing Machine	→	Type 0 Grammar.

finite state automata

DfA ⇒

- Q ⇒ Set of all states
- Σ ⇒ Set of input alphabets/symbol
- q_0 ⇒ Initial state
- F ⇒ Set of final state(s)
- δ ⇒ Transition function $(Q \times \Sigma \to Q)$

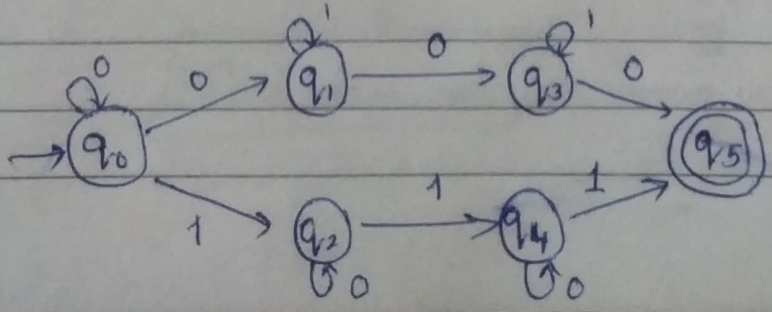


Transition Table

State	0	1
→ q_0	q_1	q_2
q_1	q_3	q_1
q_2	q_2	q_3
q_3	q_3	q_3

NfA ⇒

- Q ⇒ Set of all states
- Σ ⇒ Set of input symbols
- q_0 ⇒ Initial state
- F ⇒ Set of final state(s)
- δ ⇒ Transition function $(\delta : Q \times \Sigma \to 2^Q)$



Date : __/__/____

State	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	q_2
q_1	q_3	q_1
q_2	q_2	q_4
q_3	q_5	q_3
q_4	q_4	q_5
q_5	q_5	q_5

Regular Expression

Used to denote regular language, using the alphabets of the language.

$abc \Rightarrow a^*bc$, ab^+ , a^*b^*

$* \rightarrow$ 0 or infinite $+ \rightarrow$ 1 or infinite.

Grammar

N $\rightarrow$ Non-terminal symbols (A, B, S, D, K, etc.)

T $\rightarrow$ Terminal symbols {a, b, c---}

P $\rightarrow$ Set of production rules

S $\rightarrow$ Start symbol

- Terminal symbols can't be replaced but non-terminal symbols can be replaced.

Date : __/__/__

list → list + digit | list - digit | digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

* Left-most derivation

list → list + digit
list → list - digit + digit
list → digit - digit + digit
list →

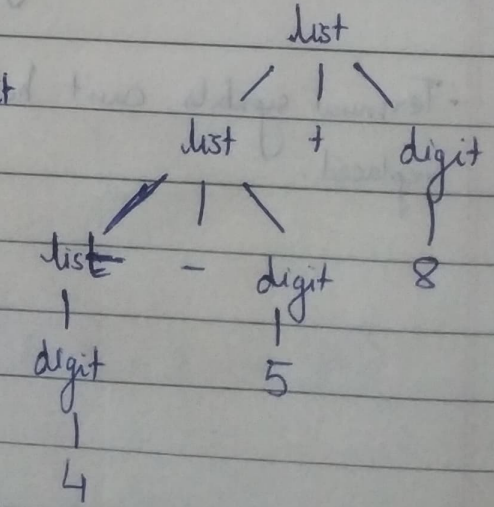
Parse Tree

list → list + digit | list - digit | digit

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4 - 5 + 8

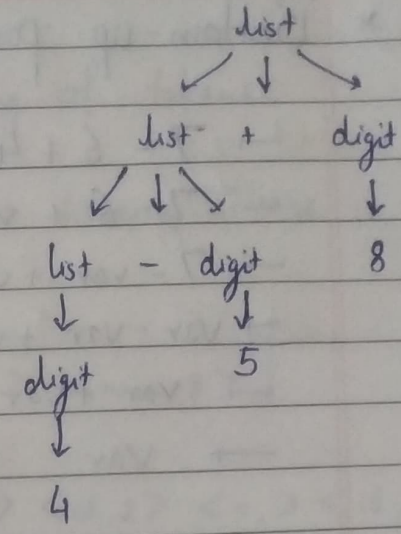
Left most derivation: list → list + digit
→ list - digit + digit
→ digit - digit + digit
→ 4 - digit + digit
→ 4 - 5 + digit
→ 4 - 5 + 8



Date : __/__/__

Right most derivation

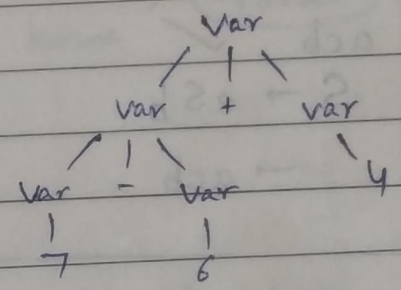
list → list + digit
→ list + 8
→ list - digit + 8
→ list - 5 + 8
→ digit - 5 + 8
→ 4 - 5 + 8



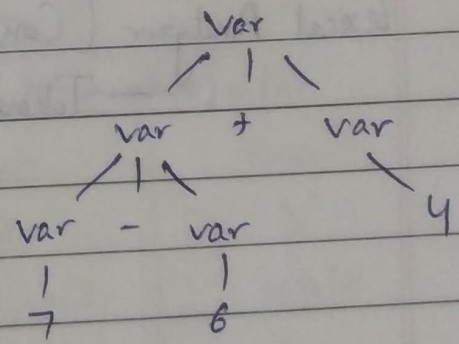
* Ambigious Grammar → More than 2 derivation / parse tree are present

* var → var + var | var - var | var | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Left most → var → var + var
→ var - var + var
→ 7 - 6 + 4



Right most → var → var + var
→ var + 4
→ var - var + 4
→ 7 - 6 + 4



~~var → var - var~~
~~→ var~~

var → var - var
→ var - var + var
→ 7 - 6 + 4

(Top-down parsing)

* Bottom-up parsing

→ 7 - 6 + 4
— 7 - 6 + var
— 7 - var + var
→ var - var + var
→ var + var
→ var

Q ⇒ S → aSb | C | ab

acb, abba

① acb ✓
S → aSb
→ acb

② abba ✗
S → aSb
→ a abb

Lexical Analyzer (Convert input into stream of tokens)
— Tokens

1. Keywords
2. Identifiers
3. Operators
4. Punctuations
5. Constants
6. Literals
7. Strings

Date : __/__/__

Lexeme

└ A sequence of characters comprising of tokens.

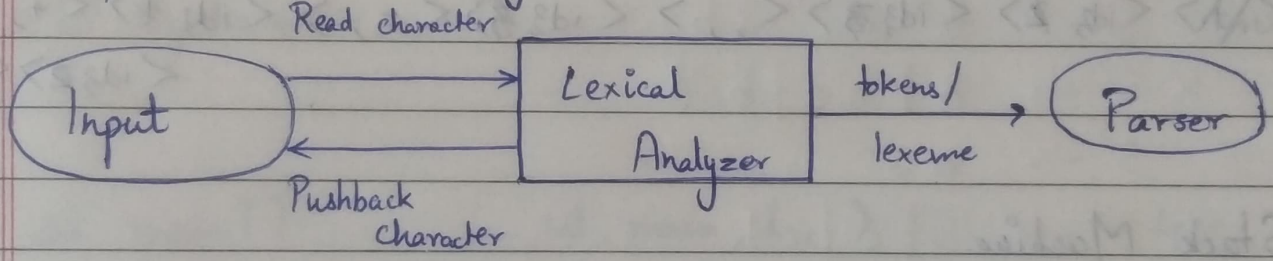
Input ⟶ Lexical Analyzer ⟶ Lexeme/Tokens

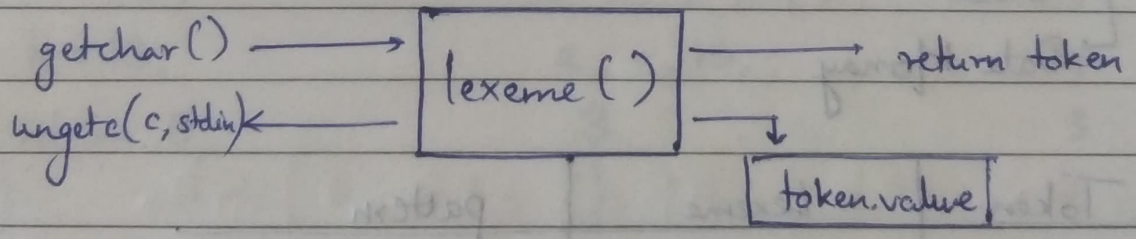
//Removal of whitespaces & comments

① Constants :- 7+8 ⟶ [<num, 7> <+, > <num, 8>]

② Identifiers :- p+q+r ⟶ [<id, 1> <=,> <id, 2> <+, > <id, 3>]

* Interface of Lexical Analyzer





Date : __/__/__

Q) int x;
x = 7;
int y, z;
z = x + y;

<id, 1> < = , > <num, 7>
<id, 3> < = , > <id, 1> <+, > <id, 2>

① int x ; <int> <id, 1> <;, >
② x = 7 ; <id, 1> <=, > <num, 7> <;, >
③ int y, z ; <int> <id, 2> <id, 3> <;, >
④ z = x + y ; <id, 3> <=, > <id, 1> <+, > <id, 2> <;, >

<int> will also be counted as a token.

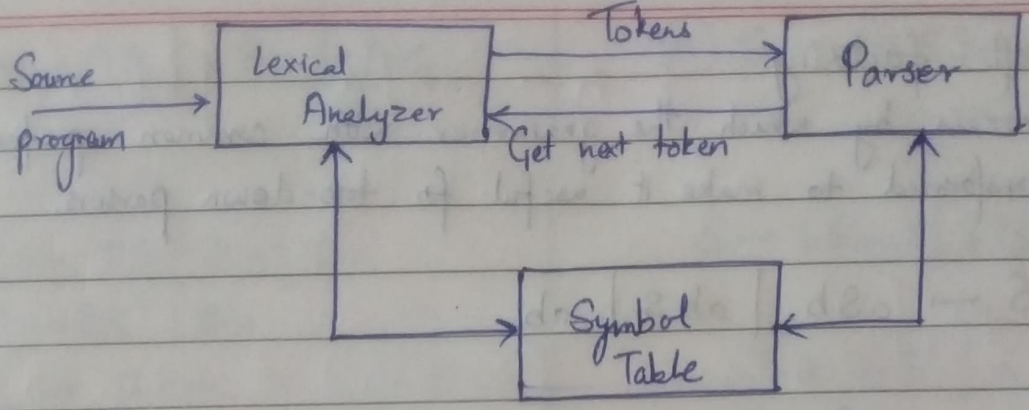
→ <int> <id1, 1> <;, > <id1, 1> <=, > <num, 7> <;, >
<int> <id2, 2> <id3, 3> <;, > <id3, 3> <=, > <id1, 1> <+, > <id2, 2> <;, >

Abstract Stack Machine

- Stack
- Data Memory

Token	Lexeme	pattern
const	const	const
if	if	If
relation	<, <=, >, >=, =	
i	pi	any numeric constant
num	3.14	any character b/w " and " except "
literal	"core"	pattern

Date : __/__/__



- To identify the token
- Regular expression should define what tokens are possible/allowed in grammar
- Recognize & categorize tokens

* Errors generated by lexical analyzers (lexical errors)
  1. Not able to identify the token
  2. Syntax error.

Q.

```
int main()
{
  int a, b;
  a = 10;
  return 0;
}
```

```
int, main, (, )
{
int a, , , b, ;
a, =, 10, ;
3
}
```

4
1
5 = 18
4
3
1

```
int main()
{
  int a = 10, b = 20;
  printf("sum is %d", a+b);
  return 0;
}
```

4
1
9 = 27
9
3
1

Left factoring

Process by which the grammar with common prefixes is transformed to make it useful for top-down parsers.

① $S \rightarrow aSb \mid abS \mid ab$

$S \rightarrow aS'$

$S' \rightarrow Sb \mid bS \mid b$

$S \rightarrow aS'$

$S' \rightarrow Sb \mid S''S \mid \epsilon$

$S'' \rightarrow b$

② $S \rightarrow ab \mid abc \mid abcd \mid b$

$S \rightarrow aS' \mid b$

$S' \rightarrow b \mid bc \mid bcd$

$S' \rightarrow bS''$

$S'' \rightarrow \epsilon \mid c \mid cd$

$S'' \rightarrow \epsilon \mid cS'''$

$S''' \rightarrow \epsilon \mid d$

③ $S \rightarrow aAb \mid aABc \mid aABcd \mid aAa$

$A \rightarrow a$

$B \rightarrow a$

$S \rightarrow aS'$

$S' \rightarrow Ab \mid ABc \mid ABcd \mid Aa$

$S' \rightarrow AS''$

$S'' \rightarrow b \mid Bc \mid Bcd \mid a$

$S'' \rightarrow b \mid BS''' \mid a$

$S''' \rightarrow c \mid cd$

$S''' \rightarrow cS''''$

$S'''' \rightarrow \epsilon \mid d$

$S \rightarrow aS'$

$S' \rightarrow AS''$

$S'' \rightarrow b \mid BS''' \mid a$

$S''' \rightarrow cS''''$

$S'''' \rightarrow \epsilon \mid d$

$A \rightarrow a$

$B \rightarrow a$

Date : __/__/____

④ $A \to aAB \mid aBc \mid aAc$

$A \to aA'$

× $A' \to AB \mid Bc \mid Ac$

$A' \to AA'' \mid Bc$

$A'' \to B \mid c$

→ $A \to aA'$

$A' \to AA'' \mid Bc$

$A'' \to B \mid c$

⑤ $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

$S \to bS' \mid a$

× $S' \to SSaaS \mid SSaSb \mid Sb$

$S' \to SS''$

× $S'' \to SaaS \mid SaSb \mid b$

$S'' \to SS''' \mid b$

× $S''' \to aaS \mid aSb$

$S''' \to aS''''$

$S'''' \to aS \mid Sb$

* Left Recursion / Recursion

$f(n)$
└ $f(n-1)$
└ $f(n-2)$
└ ------ } Recursion

If a grammar is recursive, the non-terminals are present on both LHS & RHS.

$S \to Sa$
$S \to aS$
$S \to SaS$
$S \to aSa$

} → i). Direct Recursion
ii). Indirect Recursion

$A \to aB \mid a$
$B \to aA$
} $A \to aaA \mid a$] Indirect Recursion.

* Types of Direct Recursion
  i). Left Recursion [Non-terminal at extreme left] $S \to Sa$
  ii) Right Recursion [Non-terminal at extreme right] $S \to aS$

Parser might left to infinity operations in case of left recursion

Convert left recursion into right recursion

Q). ~~$A \to \beta A'$~~ $A \to A\alpha \mid \beta$
~~A'~~

$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

Q) $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$

Date: 04-feb-2022

Aduet Dabral 190184010 B.T. CSE (AI-DS) A Date: / /

Practice Assignment

CS304 - Compiler Design

1). Consider the grammar $G = <\{S\}, \{do, while\}, P, S>$ with productions P,

$$S \rightarrow do\ S\ while\ S \mid while\ S\ do\ S \mid \epsilon$$

Show that this grammar is ambiguous.

Sol ⇒ for simplicity,

Let $a = do$, $b = while$

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

Considering string $w = abab$.

i). Left Parse Tree

$S \rightarrow aSbS$

$S \rightarrow a\epsilon bS$

$S \rightarrow abS$

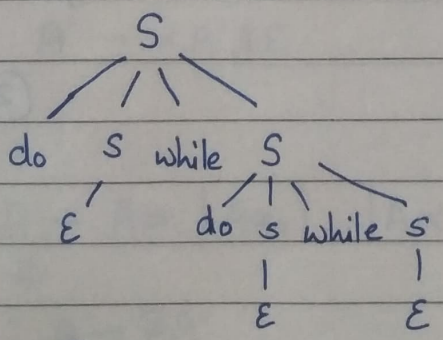
$S \rightarrow ab\,aSbS$

$S \rightarrow ab\,a\epsilon bS$

$S \rightarrow ab\,ab\epsilon$

$S \rightarrow ab\,ab$

$S \rightarrow do\ while\ do\ while$



ii). Right Parse Tree

$S \rightarrow aSbS$

$S \rightarrow abSaSbS$

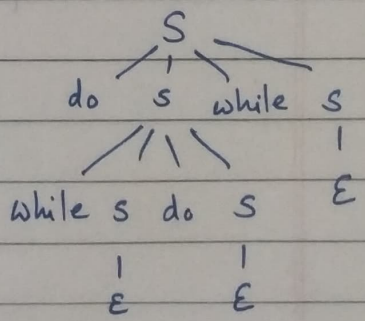
$S \rightarrow ab\epsilon\,aSbS$

$S \rightarrow aba\epsilon bS$

$S \rightarrow abab\epsilon$

$S \rightarrow abab$

$S \rightarrow do\ while\ do\ while$



⇒ The given grammar is ambigious.

Date : __/__/__

2) Consider the grammar $G = \langle \{S\}, \{a, if, then\}, P, S \rangle$ with productions P

$$S \rightarrow if\ S\ then\ S \mid if\ S \mid a$$

Left factor the productions.

Sol ⇒ For simplicity,

Let $a = a$

$if = b$

$then = c$

Thus,

$$S \rightarrow bScS \mid bS \mid a$$

* Left factoring the production rules

$$S \rightarrow bS' \mid a$$
$$S' \rightarrow ScS \mid S$$
$$S' \rightarrow SS''$$
$$S'' \rightarrow cS \mid \varepsilon$$

So,

The left factored grammar is

$$S \rightarrow bS' \mid a$$
$$S' \rightarrow SS''$$
$$S'' \rightarrow cS \mid \varepsilon$$

$$\Rightarrow \begin{cases} S \rightarrow if\ S' \mid a \\ S' \rightarrow SS'' \\ S'' \rightarrow then\ S \mid \varepsilon \end{cases}$$

Ans//

Date : __/__/__

* for left recursion
  - Try to find the terminating condition for the parser.

① $A \rightarrow A\alpha \mid \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

② $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2$

$\Downarrow$

$A \rightarrow \beta_1 A' \mid \beta_2 A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

③ $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$

④ $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

$\Downarrow$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

⑤ $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A'$

⑥ $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

$\Downarrow$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$

Date : __/__/__

Q) $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

⇓

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

Q) $S \to Sab \mid c$

⇓

$S \to cS'$

$S' \to abS' \mid \epsilon$

Q) $S \to SSS \mid \alpha$:

$S \to \alpha S'$

$S' \to SSS' \mid \epsilon$

Q) ~~$A \to A(A)A$~~

$A \to A(A) \mid a$

⇓

$A \to aA'$

$A' \to (A)A' \mid \epsilon$

Q) $A \to AaA \mid b$

$A \to bA'$

$A' \to aAA' \mid \epsilon$

Syntax Analyzer

- Parsing techniques - Syntax Analysis
  - Top-down
  - Bottom-up

① Top-down
- Root node to leaves
- Left most derivation
- Parse Tree decision based on production selected
- Types
  i). Top-down with backtracking (Recursive descent)
  ii). Top-down without backtracking (Predictive parsing / LL1)

② Bottom-up
- Leaves → root node
- Right most derivation
- Making a parsing tree is based on terminals (strings)
- Types
  i). Operator precedence
  ii). LR Parser
    - LR(0).
    - SLR(1)
    - LALR(1)
    - CLR(1)

* Recursive Descent

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

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

→ id + id * id

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

① id + id * id
↑

$E \to TE'$

$E \to FT'E'$

$\to (E)T'E'$ ← Backtrack

$E \to id\ T'E'$

② id + id * id
↑

$E \to id\ T'E'$

$\to id * FT'E'$ ← Backtrack

$E \to id\ \epsilon\ E'$

$E \to id\ E'$

$E \to id + TE'E'$

③ id + id * id
↑

$E \to id + T\ E'E'$

$\to id + FT'E'E'$

$\to id + (E)T'E'E'$

$\to id + id\ T'E'E'$

④ id + id * id
↑

$E \to id + id * FT'E'E'$

⑤ id + id * id
↑

$E \to id + id * (E)T'E'E'$

$\to id + id * id\ T'E'E'$

⑥ $E \to id + id * id\ \epsilon\ E'E'$

$\to id + id * id\ \epsilon E'$

$\to id + id * id\ \epsilon$

$\to id + id * id.$

Date : __/__/____

Q) $S \rightarrow pAy$
$A \rightarrow a \mid la$

String = "play"

① play
↑

$S \rightarrow pAy$
↑

② play
↑

$S \rightarrow pAy$
↑
Backtrack

$S \rightarrow pay$
↑

$S \rightarrow play$
↑

③ play
↑

$S \rightarrow play$
↑

④ play
↑

$S \rightarrow play$
↑

//

* first & follow

$S \rightarrow abc \mid d$ String :- aabb

:- d

If the parser knows in advance, what comes in the string, then the possibility of backtracking is eliminated.

* first()

→ If α is any string of grammar, then First(α) is the set of terminals that are in first position in the string derived from α.

1. If the terminal symbol occurs in the First(), then First(α) = {a}
2. If there is a rule $\alpha \rightarrow \epsilon$, then First($\alpha$) = {$\epsilon$}
3. ~~If there~~ for the rule,

$$\alpha \rightarrow x_1 x_2 x_3 \ldots x_n \mid \alpha \rightarrow x_1 \mid x_2 \mid x_3 \mid \ldots \mid x_n$$

$$First(\alpha) = First(x_1) \cup First(x_2) \cup First(x_3) \ldots \cup First(x_n)$$

4. If $X \rightarrow Ya$, then first(X) = first(Y), provided first(Y) shouldn't contain ϵ.

(1)

$S \rightarrow aAB$ First(S) = {a}

$A \rightarrow bA \mid c$ First(A) = {b, c}

$B \rightarrow c \mid d \mid \epsilon$ First(B) = {c, d, ϵ}

(2)

$S \rightarrow aAb$ First(S) = {a}

$A \rightarrow bA \mid Bd$ First(A) = {b} ∪ {c, d} = {b, c, d}

$B \rightarrow c \mid d$ First(B) = {c, d}

Date : __/__/__

③ $S \to (s) \mid \epsilon$ — $first(s) = \{ (, \epsilon \}$

④ $S \to (s) \mid a$ — $first(s) = \{ (, a \}$

⑤ $S \to AB$ — $first(s) = first(A) = \{a, b\}$
$A \to aA \mid b$ — $first(A) = \{a, b\}$
$B \to c$ — $first(B) = \{c\}$

⑥ $S \to AB$ — $first(S) = \{first(A) - \epsilon\} \cup first(B) = \{a, c\}$
$A \to a \mid \epsilon$ — $first(A) = \{a, \epsilon\}$
$B \to c$ — $first(B) = \{c\}$

⑦ $S \to ABCDE$ — $first(s) = \{a, b, c, d, e, \epsilon\}$
$A \to a \mid \epsilon$ — $first(A) = \{a, \epsilon\}$
$B \to b \mid \epsilon$ — $first(B) = \{b, \epsilon\}$
$C \to c \mid \epsilon$ — $first(C) = \{c, \epsilon\}$
$D \to d \mid \epsilon$ — $first(D) = \{d, \epsilon\}$
$E \to e \mid \epsilon$ — $first(E) = \{e, \epsilon\}$

---

*

If $X \to Y_1 Y_2 Y_3 \dots Y_n$

① $first(x) = first(Y_1)$

② $first(Y_1) = \{\epsilon\} \Rightarrow first(x) = \{first\{Y_1\} - \epsilon\} \cup first\{Y_2\}$

③ If $first(Y_i), i \in N$ contains ϵ
↳ $first(X)$ contains ϵ.

PCP