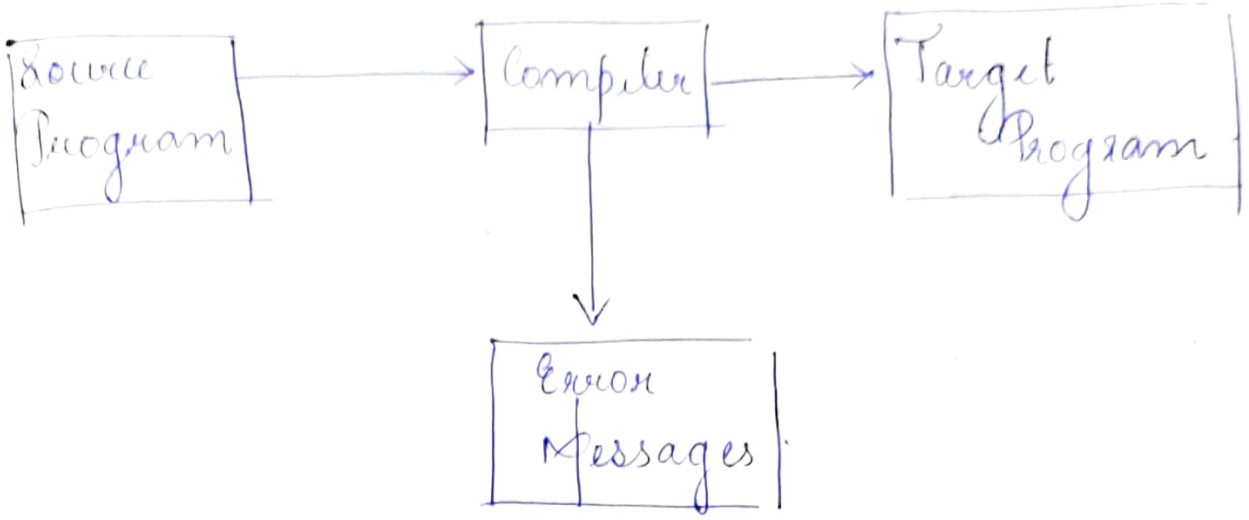
Introduction to Compilers



Phases of a Compiler

Analysis of the source program

1. Lexical analysis (or scanning or linear)
2. Hierarchical analysis (syntax/Parsing)
3. Semantic analysis.

Linear Analysis

sum = a + b ; (letter followed by (letter/digit)

Token =
- special symbol
- identifier
- operators
- keywords, . . .

The lexical analysis phase the given input/ source program is scanned from left to right thereby group into tokens.

Token

→ sequence of characters having collective meaning and moreover it must match with a pattern (regular expressions)

→ identifiers
→ keyword
→ operator
→ special symbol
→ constant.

lexeme

→ sequence of characters in the source program i.e matched by the pattern in order to identify a specific token eg: Position = initial + rate * 60

id1 = id2 + id3 * 60

eg: sum = a + b;

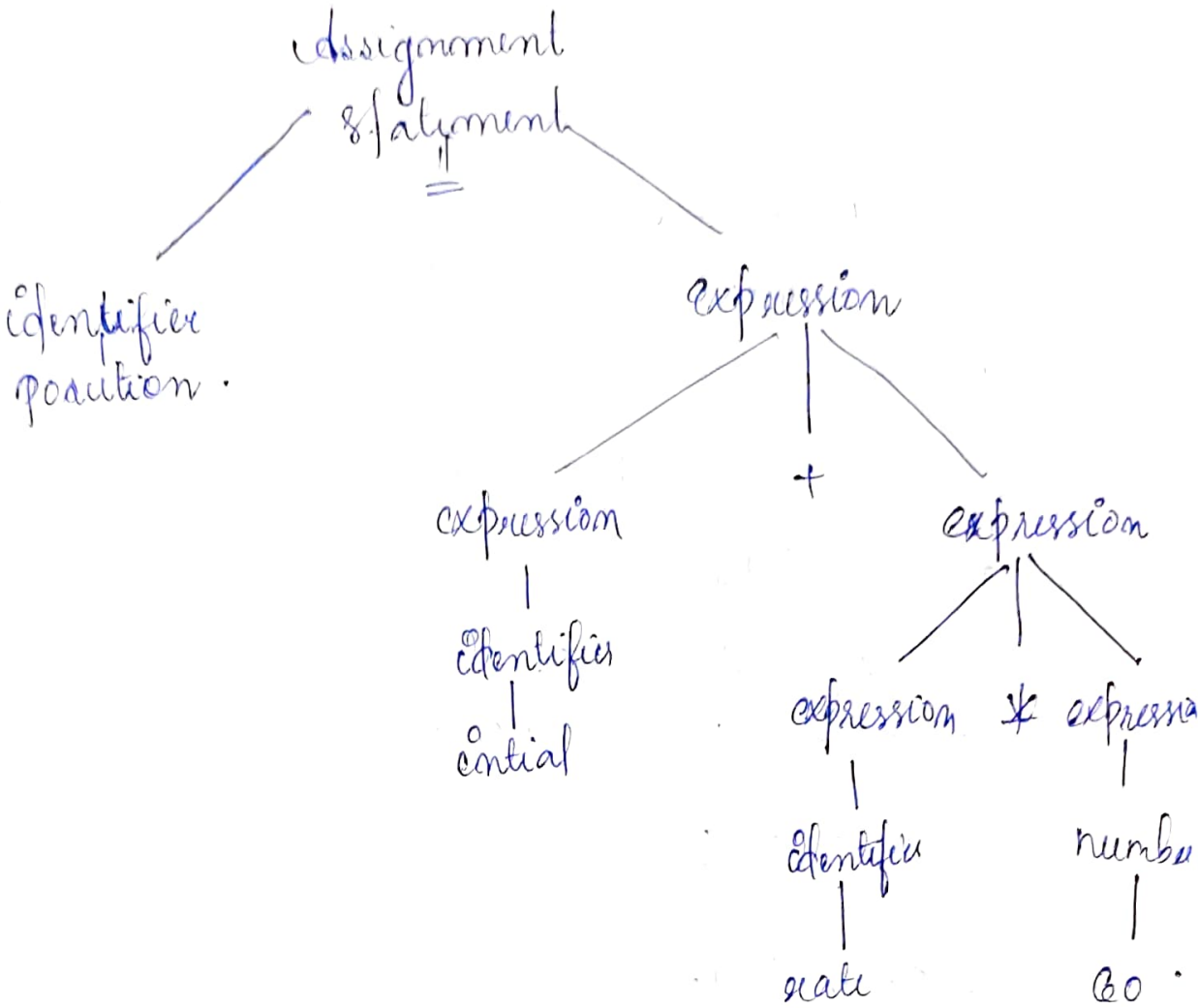
printf ("Enter the numbers");

Hierarchical Analysis (Parsing / Syntax)

CFG Rule

$E \rightarrow E+E \mid E*E \mid (E) \mid id \mid num$.

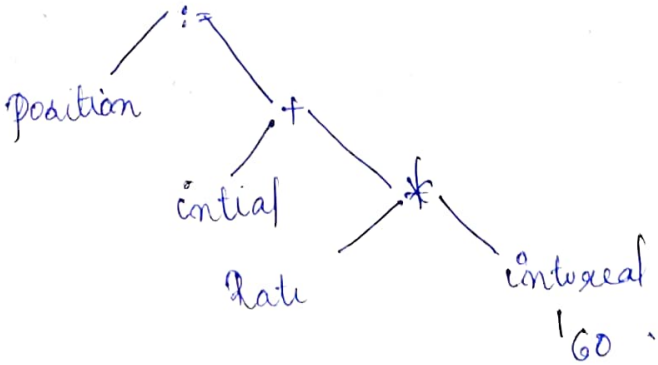
Position = initial + rate * 60



→ It involves grouping the tokens of source program (received during lexical analysis phase) into grammatical phases that are used by compiler to synthesis output (parse tree based on CFG) (based on the given source program))

Semantic analysis

eg: position = initial + rate * 60



Intermediate Code Generation

→ it should be easy to produce
→ easy to translate into target program.
→ three address code (representation)
→ position = initial + rate * 60

id1 = id2 + id3 * 60.

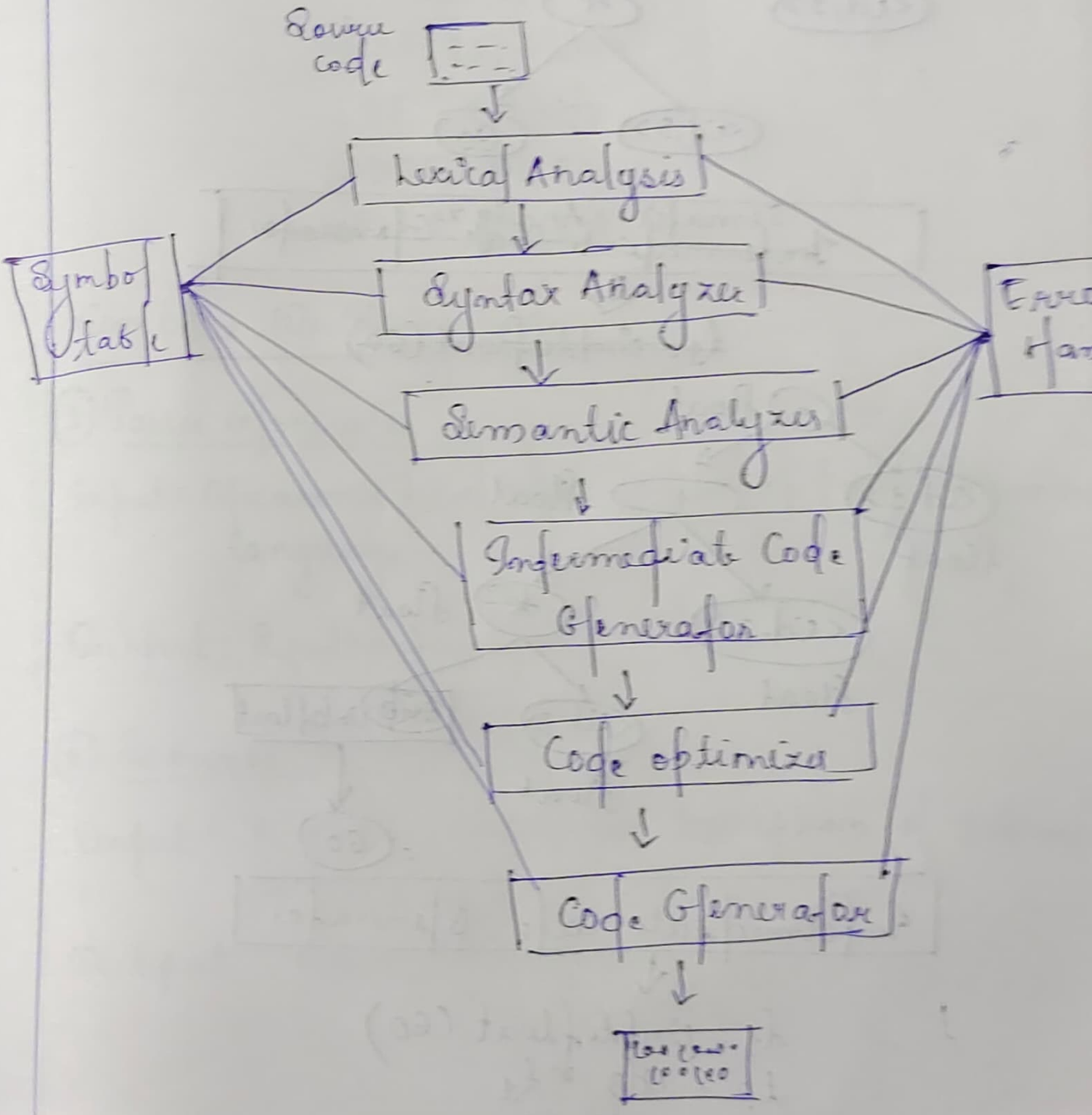
temp1 = inttoreal (60)
temp2 = id3 * temp1.
temp3 = id2 + temp2.
id1 = temp3.

Code Optimisation

Temp1 = id3 * inttoreal (60)
id1 = id2 + temp1.

Code Generation

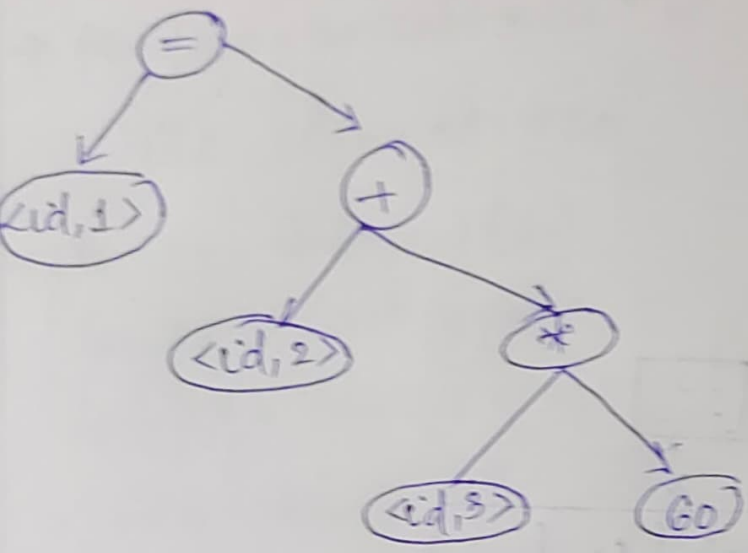
→ Final phase of the compiler is the generation of target code consisting of relocatable machine code.
→ Memory locations are selected for each of the variables used by the program.


Source code
Lexical Analysis
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code optimizer
Code Generator
Symbol table

Lexical Analyzer

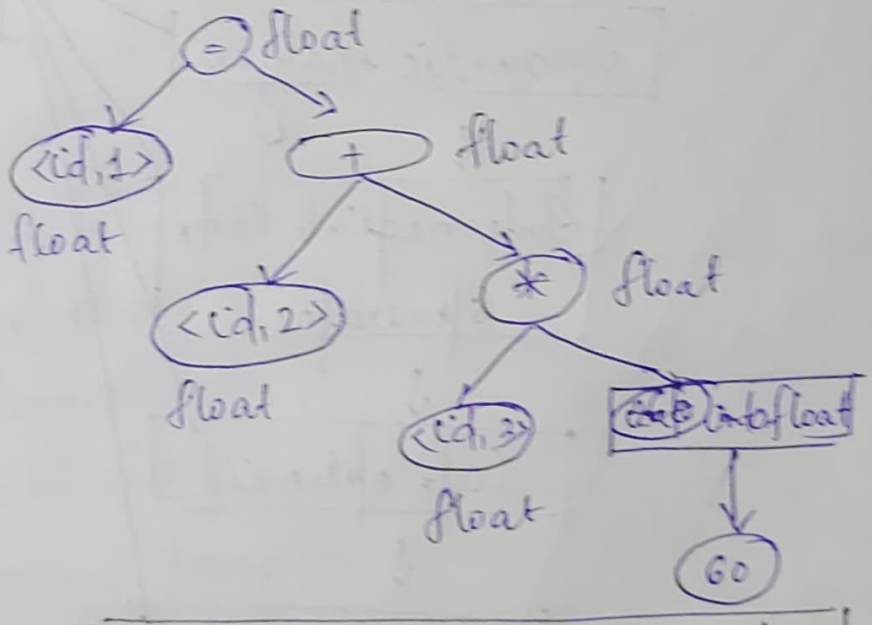
<id,1> <=> <id,2> <+> <id,3> <*> <60>

Syntax Analyzer



Semantic Analyzer ~~Intermediate code Generator~~

~~t1 = int to float (60)~~



Intermediate code Generator

t1 = inttofloat (60)
t2 = id3 * t1
t3 = id1 + t2

id1 = t3

[Code Optimizer]

↓

t1 = id3 * 60.0

id1 = id2 + t1

[Code Generator]

↓

LDF R2, id3

MULF R2, #60.0

LDF R1, id2

ADDF R1, R2

STF id1, R1

Compiler Writting tools

1. Parser Generator

Input: Grammatical description of programming language

Output: Syntax Analyzer.

2. Scanner Generator

Input: Regular expression descriptions of tokens of a language

Output: Lexical Analyzer

Module 2

Context Free Grammar

-> set of tokens terminals
-> non-terminal
-> set of productions ($A \rightarrow B$)

$A \rightarrow$ L.H.S.
$B \rightarrow$ R.H.S

-> Start Symbol.

-> $S \rightarrow AB$
$A \rightarrow a/\epsilon$

$AB \rightarrow$ Non-Terminals
$a/\epsilon \rightarrow$ ~~Non~~ terminals.

Terminals

a, b, ... (Lowercase letters)
-> operator symbol
-> punctuation symbols
-> id or if

Non-terminals

-> uppercase letters.
-> Lower italic names (expression, statement, operator (op)

->

-> if $A \rightarrow \alpha_1 | \alpha_2 , \ldots A \rightarrow \alpha_k$

we can write

$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$.

Derivation

→ sequence of production rules / sequence of replacements of non-terminals.

eg: derivation of (id*id) → generate derivation for (id*id).

$E \to E+E$
$\to E*E$
$\to (E)$
$\to -E$
$\to id$

} productions.

$E \to (E)$
$\to (E*E)$
$\to (id*E)$
$\to (id*id)$

$S \overset{*}{\Rightarrow} \alpha$ if α contains non-terminals, it is called sentential form of G.
if α does not contains non-terminals it is called sentence of G.

→ if we choose the leftmost non-terminal in each derivation step. This derivation called leftmost derivation. (used in top down).

→ if we always choose the rightmost ~~derival~~ non-terminal that derivation rightmost derivation. (used in bottom up parse tree)

⇒ Leftmost derivation.

generate derivation (id*id)+id

$E \to E+E$
$\to E*E$
$\to (E)$ $\to -E$
$\to id$.

E → E * E
→

E → E + E
→ (E) + E
→ (E * E) + E
~~→ (id * id) + id~~ ..
→ (id * E) + E
→ (id * id) + E
→ (id * id) + id

Rightmost derivation

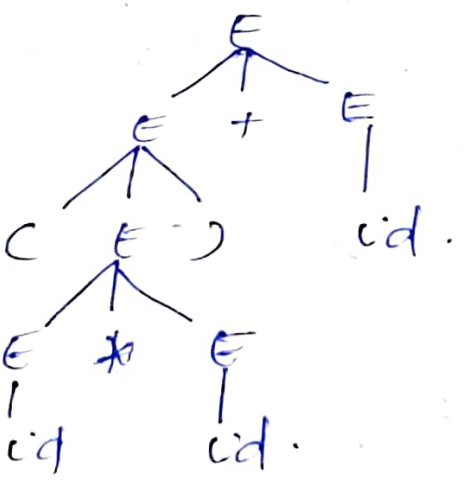
(id * id) + id.

E → E
→ E + E
→ E + id
→ (E) + id
→ (E * E) + id
→ (E * id) + id
→ (id * id) + id.

Parse tree using ~~right~~ left most derivation.

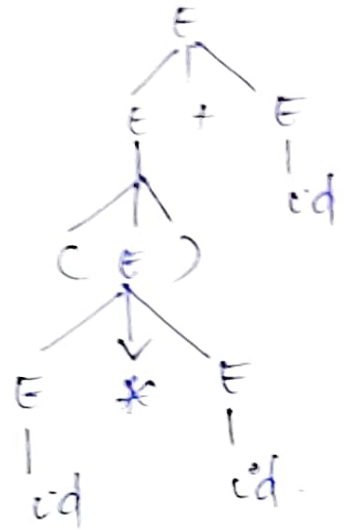
(id * id) + id

E → E
→ E + E
→ E + id
→



using rightmost derivation.

(id+id)*id



Ambiguous Grammar.

→ A grammar is ambiguous if there are multiple parse trees for the same sentence. grammar for programming language is ambiguous.

→ ~~(id+id)*id~~

→ id+id*id

leftmost derivation

E → E
→ E+E
→ id + ~~E*E~~
→ id + E*E
→ id + id*E
→ id + id*id.

→ leftmost derivation #2

E → E
→ E*E
→ E+E*E
→ id+E*E
→ id+id*E
→ id+id*id.

Left Recursion

If we have production of form:

$$A \rightarrow A\alpha \mid \beta$$

it could be replaced by non-recursive productions

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Q. $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Q. $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

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

$A \rightarrow A\alpha \mid \beta$

$E \rightarrow E+T \mid T$

$A = E$

$\alpha = +T$

$\beta = T$

$E \rightarrow TE'$

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

$T \rightarrow T*F \mid F$

$A = T$

$\alpha = *F$

$\beta = F$

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Indirect Left Recursion

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sc \mid \epsilon$

Remove Left Recursion.

Left Factoring

Production of form: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Replaced By

$A \rightarrow \alpha A'$

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

Production Form: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

Replaced By:

$A \rightarrow \alpha A' \mid \gamma$

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

Q. $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

General: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$:

$A \rightarrow \alpha A' \mid \alpha$.

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

A = S

$\beta \rightarrow iEtS$

$\gamma \rightarrow \epsilon$

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

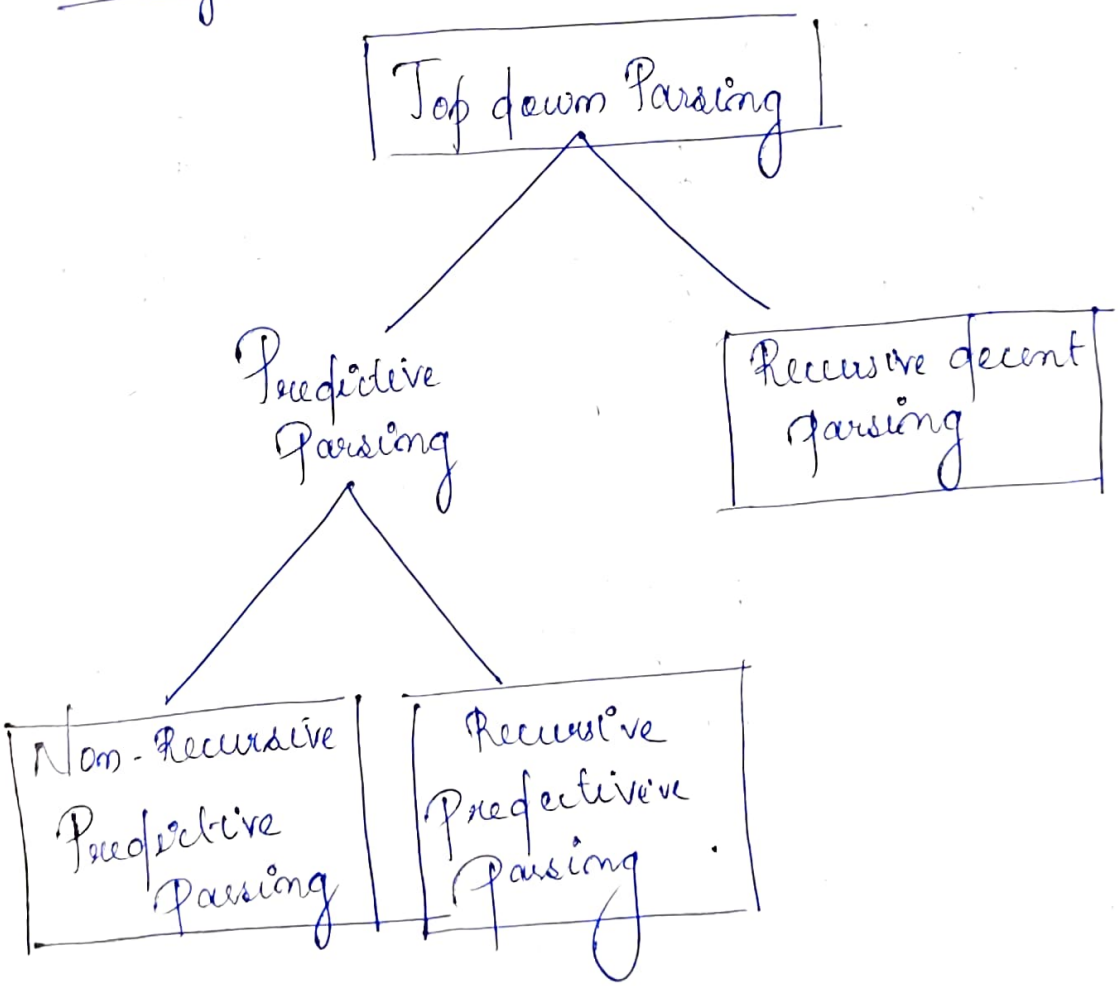
$E \rightarrow b$

→ left factoring useful for producing a grammer a su
for predictive or top-down parsing.

→ when choice between two alternative A productions
is not clear, we may be able to rewrite produ
to defer the decision until enough of input has be
seen to make right choice.

↓

Parsing



Recursive Decent Parser (RDP)

→ Top down parsing technique

→ Input scanning from left to right

→ Non terminal: Implement Recursive Procedure.

→ Terminal: Compare lookahead to check with input string.

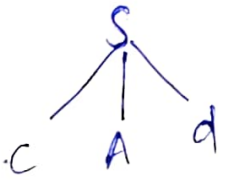
→ It need backtracking to identify the correct A-production.

Productions

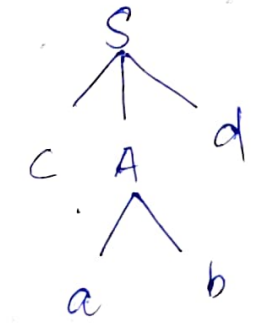
S → cAd

A → ab | a

Input string: w: cad.

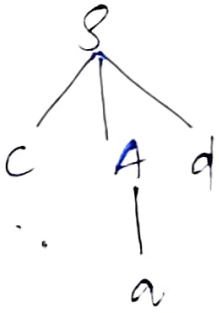


w = cad



w = cad

b and d mismatch so, backtrack



w ∈ cad

Q. Production: $E \rightarrow iE'$

$E' \rightarrow +iE' \mid \epsilon$

lookahead, l = getchar();

```
match (char t)
{
  if (l == t)
    l = getchar();
  else
    printf ("error");
}

E()
{
  if ( l == 'i')
    {
      match (i);
      E'();
    }
}

E'()
{
  if ( l == '+')
    {
      match (+);
      match (i);
      E'();
    }
  else
    {
      return;
    }
```

```
main()
{
  E();
  if ( l == '$')
  printf ("Parsed successfull");
}
```

Non-Recursive Predictive Parsing (Predictive Parsing)

→ Predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly.

→ 2 types
1. Recursive predictive Parsing.
2. Non-Recursive predictive Parsing.

Predictive Parsing Table

→ uses 2 functions:
1. FIRST()
2. FOLLOW()

→ ① FIRST()

① If x is a terminal then FIRST(x) is {x}

② ~~Ex~~ eg: $X \to a$

First(X) = First(a) = {a}

② If $X \to \epsilon$ then add ϵ to FIRST(X).

③ If x is a non-terminal and $x \to y_1 y_2 y_3 \dots y_n$. then put a in FIRST(X) if for some i, a is First(of y_i) and ϵ is all of FIRST(y_1) ... FIRST(y_n).

Eg: 1) $E \to TE'$
2) $E' \to +TE' \mid \epsilon$
3) $T \to FT'$
4) $T' \to *FT' \mid \epsilon$
5) $F \to (E) \mid id$

$F \to (E) \mid id \mid \epsilon$

$FIRST(F) = \{ (, id, \epsilon \}$

(2) $E' \to +TE' \mid \epsilon$

$FIRST(E') = \{ +, \epsilon \}$

Term
Rule No: 1 of 2.

(4) $T' \to *FT' \mid \epsilon$

$FIRST(T') \to \{ *, \epsilon \}$

(5) $F \to (E) \mid id$

$FIRST(F) \to \{ (, id \}$

(1) $E \to TE'$

$FIRST(E) = FIRST(TE')$

$= FIRST(T)$

$FIRST(T) = FIRST(FT')$

$= FIRST(F)$

$= FIRST((E) \mid id)$

$= \{ (, id \}$

$FIRST(E) = \{ (, id \}$

③ $T \rightarrow FT'$

FIRST(T) = FIRST(F).

= { (, id }

2. FOLLOW()

Rules

① If S is a start symbol then add $ to the FOLLOW(S)

② If there is a production rule of form $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except for 'ϵ' is placed in FOLLOW(B).

③ If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ then everything in FOLLOW(A) in FOLLOW(B).

Eg:1) $E \rightarrow TE'$

2) $E' \rightarrow +TE' | \epsilon$

3) $T \rightarrow FT'$

4) $T' \rightarrow *FT' | \epsilon$

5) $F \rightarrow (E) | id$

1) $E \rightarrow TE'$

$F \rightarrow (E) | id$

FOLLOW(E) = { $,) }

$E \longrightarrow TE'$

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

$FOLLOW(E') = \{\$,)\}$

③ $T \longrightarrow FT'$

$E \longrightarrow TE'$

$FOLLOW($

R.H.S → variable ke
fir

③ $T \longrightarrow FT'$

$E \longrightarrow TE'$

$FOLLOW(T) = \{+, \$,)\}$

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

$FOLLOW(T) = \{+, \$,), \}$

④ $T' \longrightarrow FOLLOW(T') = \{+, \$,), \}$

$T \longrightarrow FT'$

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

⑤ $FOLLOW(F) = \{*, +, \$,)\}$

$T \longrightarrow FT'$

$T \longrightarrow F$

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

Q. $E \rightarrow TE'$

$E' \rightarrow +TE' | \epsilon$

$T \rightarrow FT'$

$T \rightarrow *FT' | \epsilon$

$F \rightarrow (E) | id | \epsilon$

~~FIRST (T) ⟹~~

FIRST(E) ⟶ FIRST(T)

⟶ FIRST(F)

⟶ { (, id, *, +, ϵ }

FIRST(E') = { +, ϵ }

FIRST(T) = FIRST(F)

= { (, id, *, ϵ }

FIRST(T') = { *, ϵ }

FIRST(F) = { (, id, ϵ }

* FOLLOW(E) = { $,) }

* FOLLOW(E') = { $,), - }

* FOLLOW(T) = { +, *, $,) }

* FOLLOW(T') = { +, $,) }

* FOLLOW(F) = ~~{ +, $,) }~~ { *, +, $,) }

Parsing Table

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

Q. $S \to iEtSS' \mid a$ ✓

$S' \to eS \mid \epsilon$ ✓

$E \to b$

FIRST(S) $\longrightarrow \{i, a\}$

FIRST(S') $\longrightarrow \{e, \epsilon\}$

FIRST(E) $\longrightarrow \{b\}$

FOLLOW(S) $\longrightarrow \{e, \$\}$

FOLLOW(S') $\longrightarrow \{e, \$\}$

FOLLOW(E) $\longrightarrow \{t\}$

Parsing table

	i	a	e	b	t	$
S	S → iEtSS'	S → a				
S'			S' → eS S' → ∈			S' → ∈
E				E → b		

Since two production come in one row so this grammar is not LL(1).

LL1

→ the first L stands for scanning the input from left to right

→ the second L stands for producing leftmost derivation

→ and 1 stands for using one input symbol of lookahead at each step to make parsing action decision.

Algorithm (for creating parsing table)

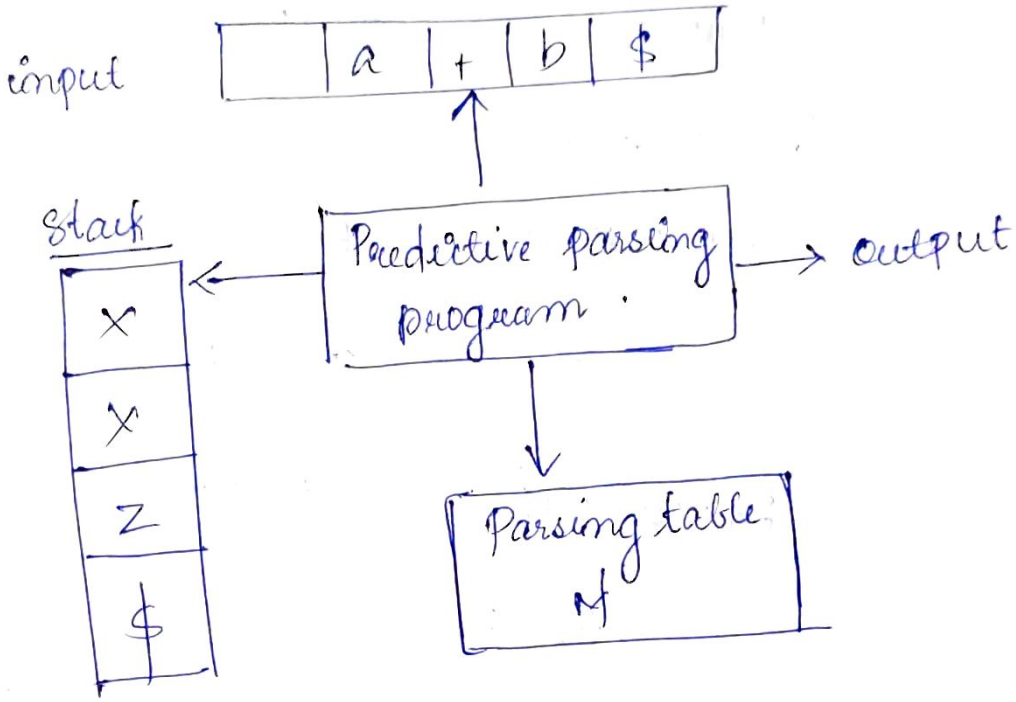
(1) For each production A → α do steps 2 of 3

(2) For each terminal a in FIRST(α), add A → α to M[A, a]

(3) If ϵ is in FIRST(α), add $A \to \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α); and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$

(4) Make each of undefined entry of M be error.

Non-Recursive predictive parsing calgorithm.



Algorithm

Set ip to point to the first symbol of w\$;

repeat

let X be the top stack symbol and a the symbol pointed to

if X is a terminal or \$ then

if X = a then

pop X from the stack and advance ip

else error()

else /* X is a nonterminal */

if M[x, a] = $X \rightarrow Y_1, Y_2, \ldots$ then begin

pop x from the stack;

push $Y_k, Y_{k-1}, \ldots Y_1$ onto the stack, with Y_1

output production $X \rightarrow Y_1, Y_2 \ldots Y_k$

end.

else error()

until X = $ /* stack is empty */

Q. Input: id+id*id

Stack	Input	Output
$E	id+id*id$	
$E'T	id+id*id$	$E \rightarrow TE'$
$E'T'F	id+id*id$	$T \rightarrow FT'$
$E'T'id	id+id*id$	$F \rightarrow id$
$E'T'	+id*id$	
$E'	+id*id$	$T' \rightarrow \epsilon$
$E'T+	+id*id$	$E' \rightarrow +TE'$
$E'T	id*id$	
$E'T'F	id*id$	$T \rightarrow FT'$
$E'T'id	id*id$	$F \rightarrow id$

$E'T'	*id$	
$E'T'F*	*id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	$T' \rightarrow \epsilon$
$E'	$	
$	$	$E' \rightarrow \epsilon$