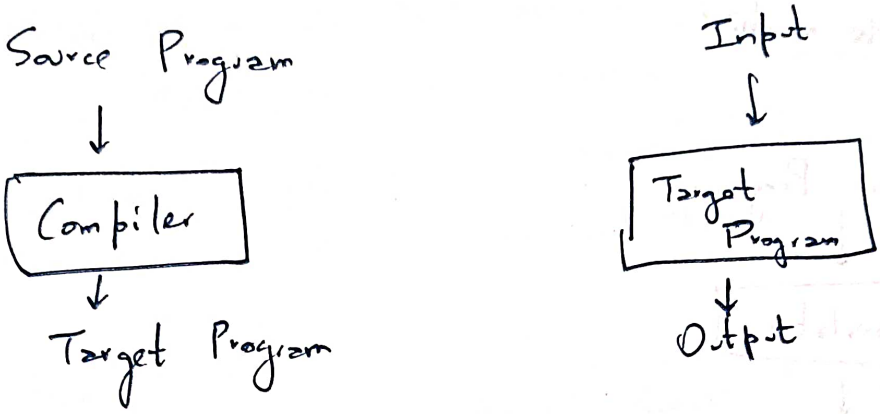
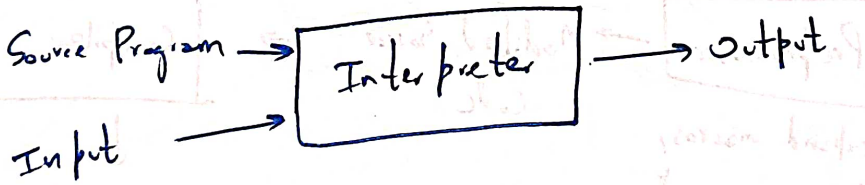
Compiler Design

→ Before a program can be run, it must first be translated into certain form (target program) in which it can be executed by a computer.



Interpreter: Instead of producing Target Program as a translation, an interpreter appears ~~as a translation~~. to directly executes the operations specified in the source program on inputs supplied by user.



Assembler: converts Assembly Level language to binary.

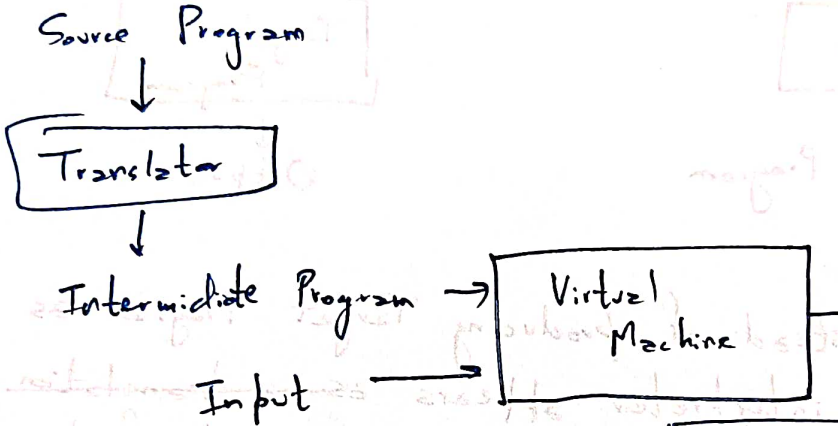
~~Bytecode & Virtual Machine~~

→ A ~~Java~~ source ~~program~~ is

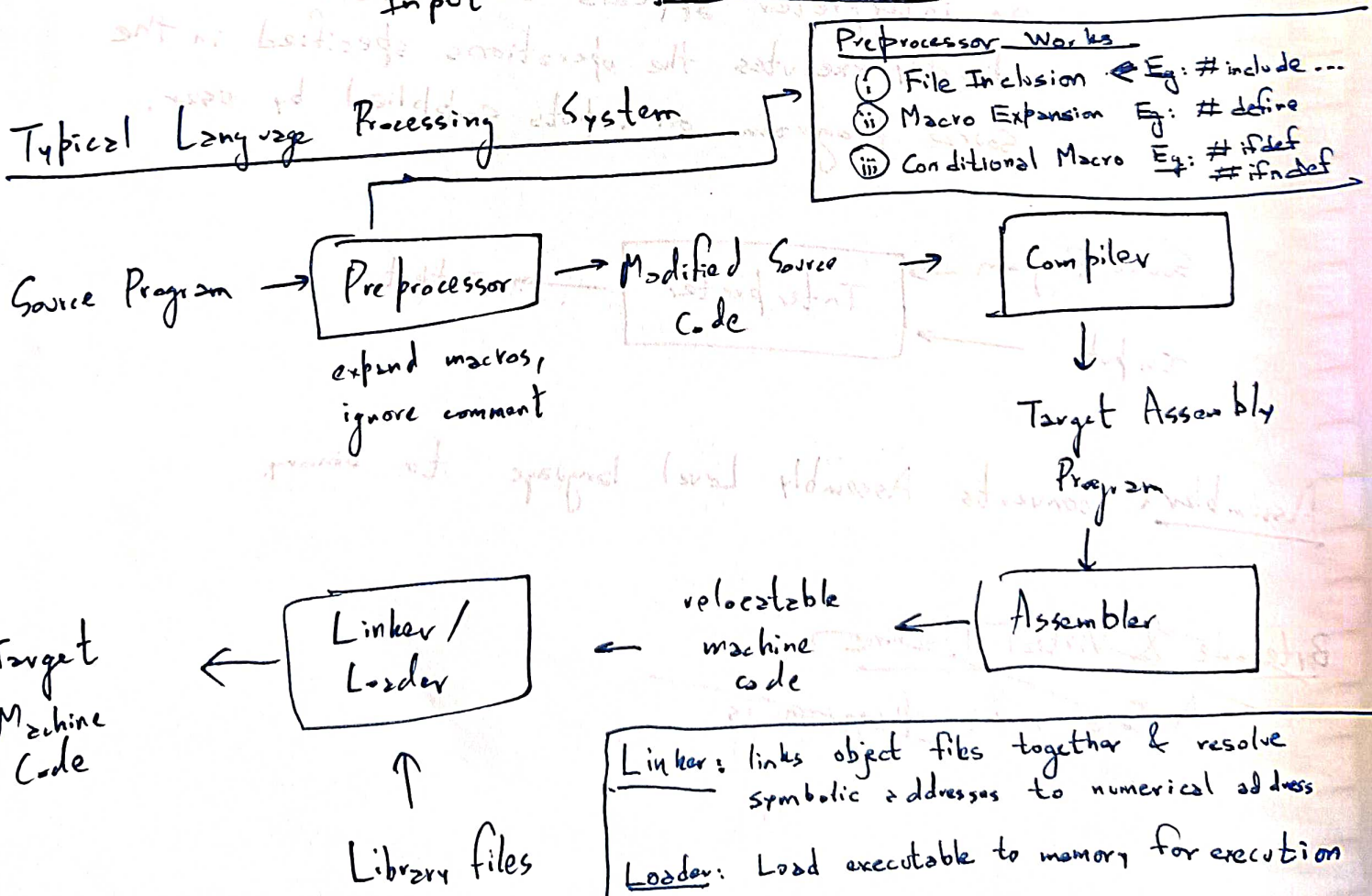
Bytecode & Virtual Machine

→ A Java Source code may first be compiled into an intermidiate form called bytecode., which are then interpreted by a virtual machine.

→ A bytecode compiled on one machine can be interpreted on another machine.



Typical Language Processing System



Linker: links object files together & resolve symbolic addresses to numerical address

Loader: Load executable to memory for execution

Analysis - Synthesis Model of Compilation

→ Two parts of compilation

(I) Analysis

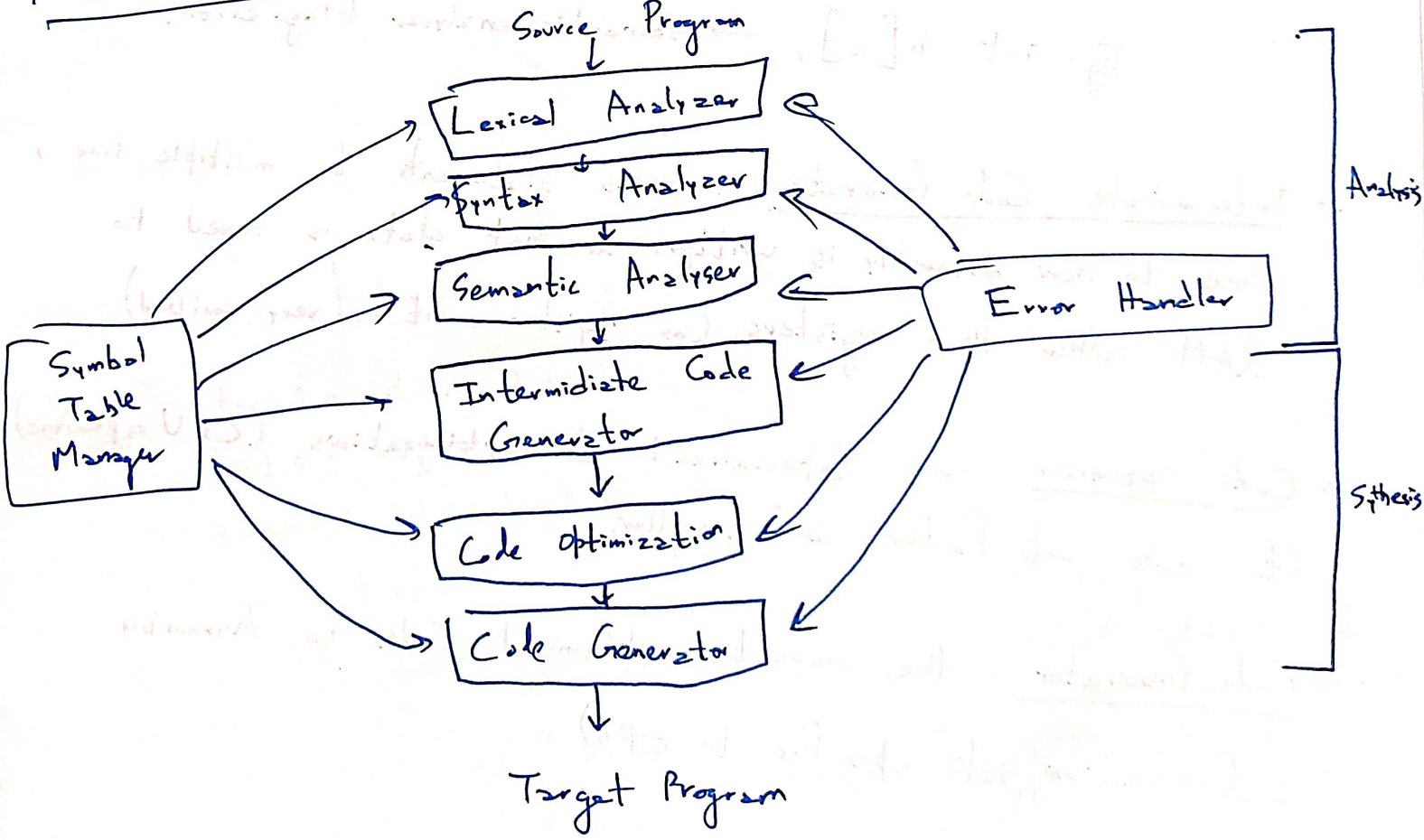
→ breaks up source program into constituent pieces & creates an intermidiate representation of program.

→ If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages so that user can take corective actions.

(II) Synthesis

→ It constructs the desired target program from the intermidiate representation & the info in symbol table.

Phases of Compilation



→ Lexical Analyzer takes the source program as input & produces a long string of tokens.

→ Syntax Analyser checks the syntax of statements and generates parse tree

→ It's also called parser

→ Eg: yacc

[Symbol Table generated in Lexical Analysis or Syntax Analysis]

→ Semantic Analyser checks the semantics and produces semantic tree

Eg: int a = b+c; → semantic analyser check if b & c possible or not (type checking, scope, etc)

Eg: int b[c]; → semantic analyser flag error.

→ Intermidiate Code Generator converts statements to multiple line, close to how assembly is written so that data is saved to RAM rather than registers. (as register count is very limited)

→ Code Optimizer uses Specialized optimizations (CPU specific) to make code faster and smaller.

→ Code Generator the converts optimized code to Assembly (instruction set specific to CPU)

Eg

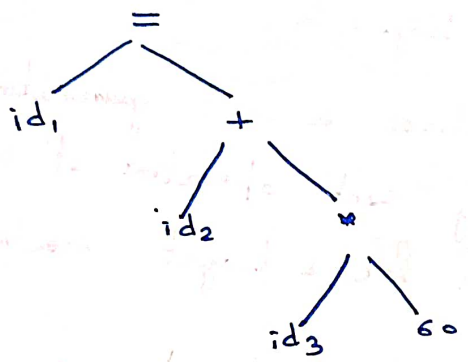
pos = init + rate * 60

↓ Lexical Analyzer

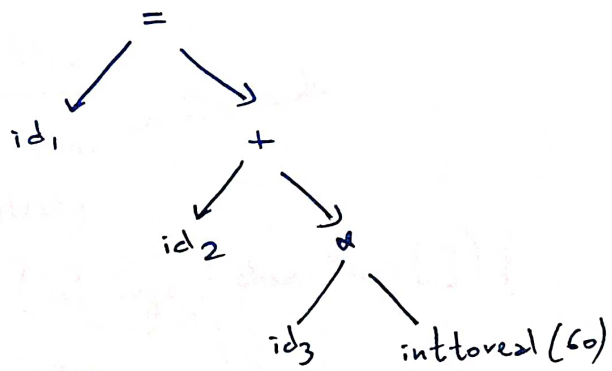
SYMBOL TABLE

$id_1 = id_2 + id_3 * 60$

↓ Syntax Analyzer



↓ Semantic Analyzer



↓ intermidiate code generator

temp1 = inttoreal(60)

temp2 = id_3 * temp1

temp3 = id_2 + temp2

id_1 = temp3

↓ Code Optimization

temp1 = id_3 * 60.0

id_1 = id_2 + temp1

↓ Code Generator

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

Compiler Construction Tools

(i) **Scannar Generators** that does lexical analysis from regexp

Eg: lex

(ii) **Parser Generators** that automatically produces syntax analyzer.

Eg: yacc

(iii) Syntax Directed translation Engine

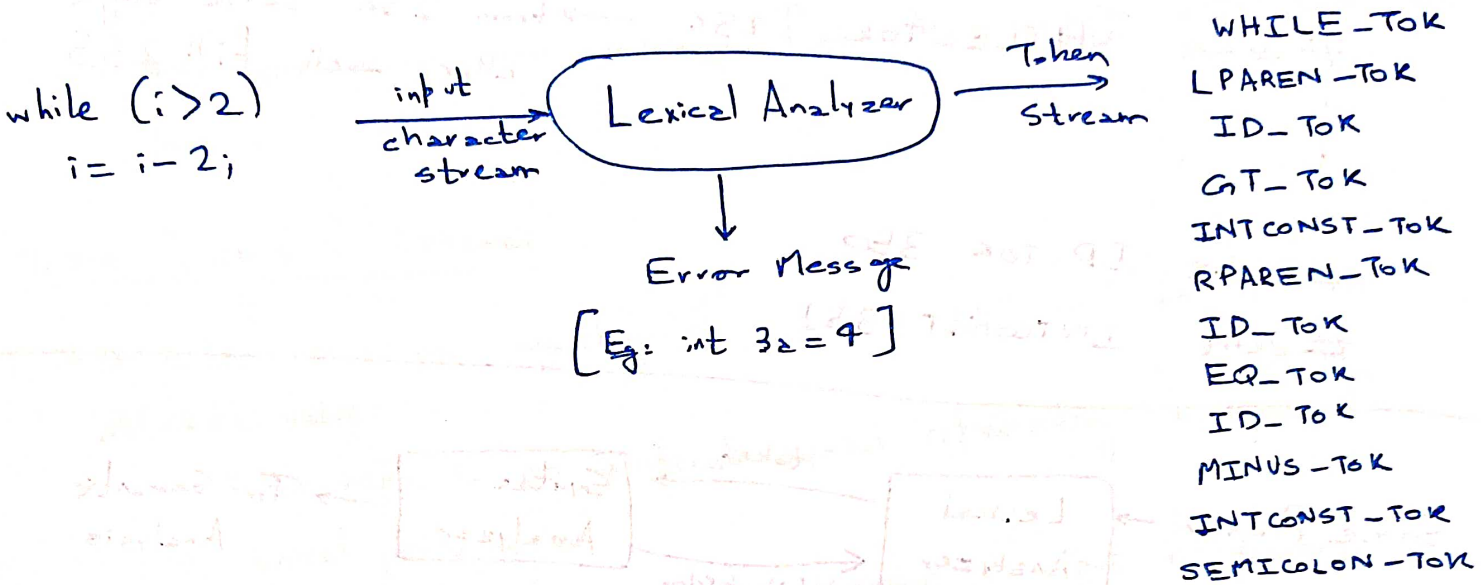
(iv) Code Generators that produces a code generator from a set of rules for translating each operation of intermidiate langugage into machine language for a target machine.

(v) Compiler Construction toolkit that provides an integrated set of routines for constructing various phases of a compiler.

LEXICAL ANALYZER

→ converts the input program into a sequence of tokens

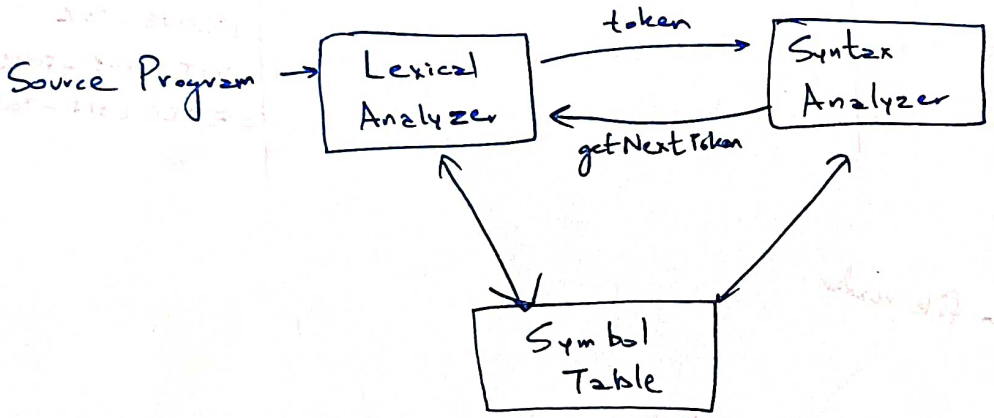
→ implemented using Finite Automata



Programmer's View

```
FILE * yyin;        → file reader
char * yytext;
void main (int argc, char *argv[]) {
    int token;
    if (argc != 2) return;
    yyin = fopen (argv[1], "r");
    while (!feof (yyin)){
        token = yylex();   ——→ returns the next token in yyin
        printf ("%d", token);
    }
    fclose (yyin);
}
```

```
#define  LPAREN_TOK  '('
#define  GT_TOK      '>'
#define  RPAREN_TOK  ')'
   ⋮
#define  WHILE_TOK   256   → from 256 cause single char ascii fill 255
   ⋮
#define  ID_TOK      350
#define  INTCONST    351
```



Eg: int a, b, c; → will give ID_TOK with resp. symbol table identifier.

Specification of Token.
- Regular Expression
- Deterministic Finite Automata
- Non-Deterministic Finite Automata.
- Non-Deterministic Finite Automata with empty Transitions.

Maximal Match Rule (or Longest Prefix Sum)

Eg: int x = y + + + z;

→ these two token (longest prefix) → so (increment, plus)

Lexical Analyzer

→ Main task is to read the input characters & produce as output a sequence of tokens.

→ Avoiding comments and white spaces in the form of blank, tab and newline character.

→ Also correlates error message from the compiler with same source program

Token, Patterns & Lexemes

→ A token is a pair consisting of a token name and an optional attribute value.

→ Token name is an abstract symbol representing a kind of lexical unit,

→ Eg: a particular keyword, or a sequence of input characters denoting an identifier.

→ This set of strings is described by a rule called pattern associated with that token. The pattern is said to match each string in the set.

→ A lexeme is a sequence of characters in the source program that matches the pattern for a token & is identified by the lexical analyzer. as an instance of that token.

→ These are smallest logical unit (words) of a program such as

A, B, 1.0, true, +, <=, ----

Eg: printf ("Total = %d", score);

- printf → lexeme matching pattern of id
- "Total = %d" → lexeme matching literal
- score → lexeme matching pattern of id

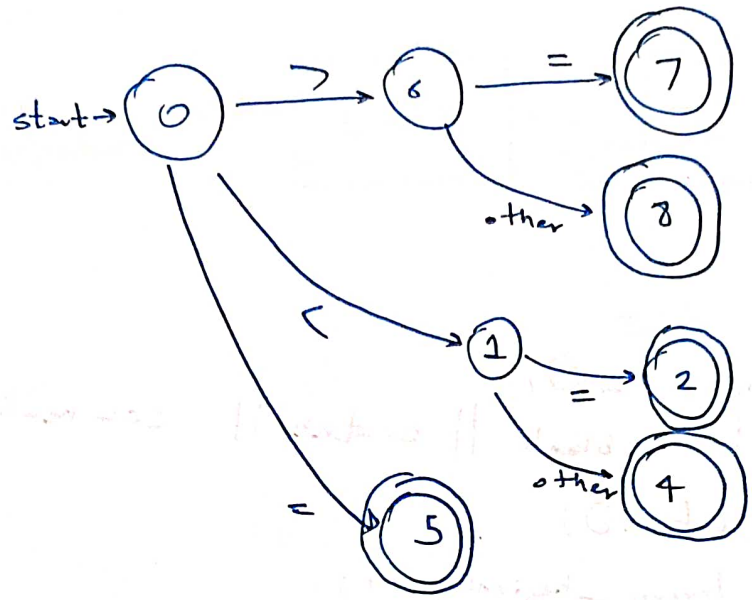
	Token	Sample Lexemes	Informal description of pattern
I	if	if	characters i, f
II	else	else	characters e, l, s, e
III	comparision	<, <=, ==, !=, >, >=, etc.	< or > or <= or >= ...
IV	id	pi, score, D2	letter followed by letters & digit
V	number	3.14159, 0, 6.02e23	any numeric constant
VI	literal	"Total = %d"	Total = %d

Eg: E = M * C + 2

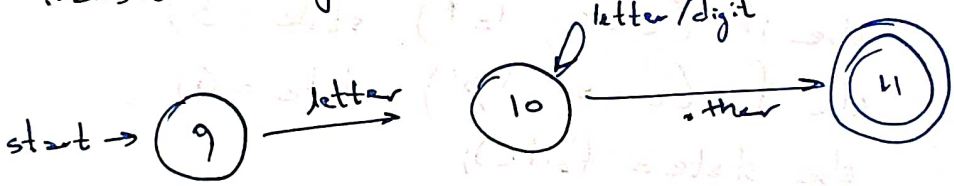
↓ tokens

<id, ptr to symbol table entry of E>
<assgn op>
<id, ptr to symbol table entry of M>
<mult op>
<id, ptr to symbol table entry of C>
<plus op>
<number, integer value 2>

Eg: Transition Diagram for >= , <= , & =

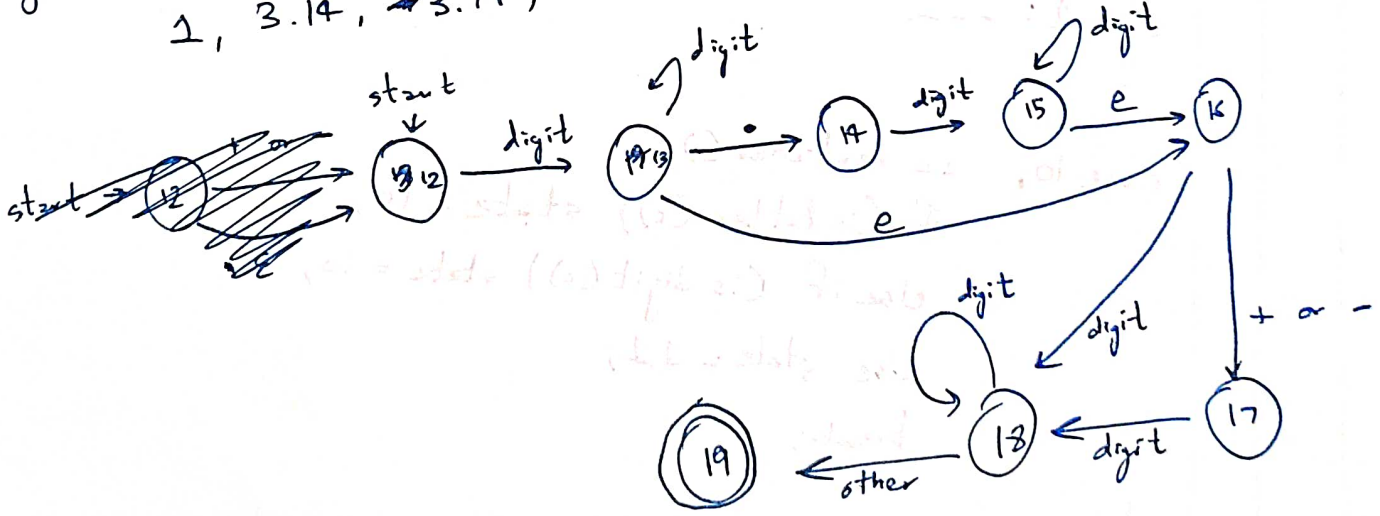


Eg: Transition diagram for identifier or keyword

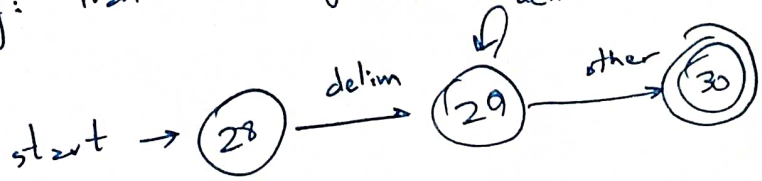


Eg: Transition diagram for numbers (ignore starting + or -)

1, 3.14, -3.14, 2.30e23, -2.30e-23, * 3.14e-10



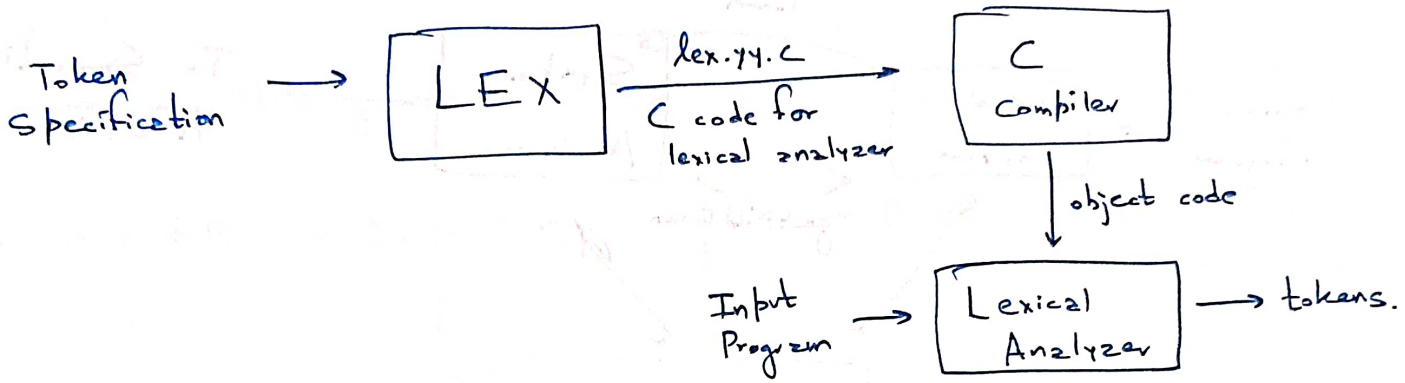
Eg: Transition Diagram for whitespace.



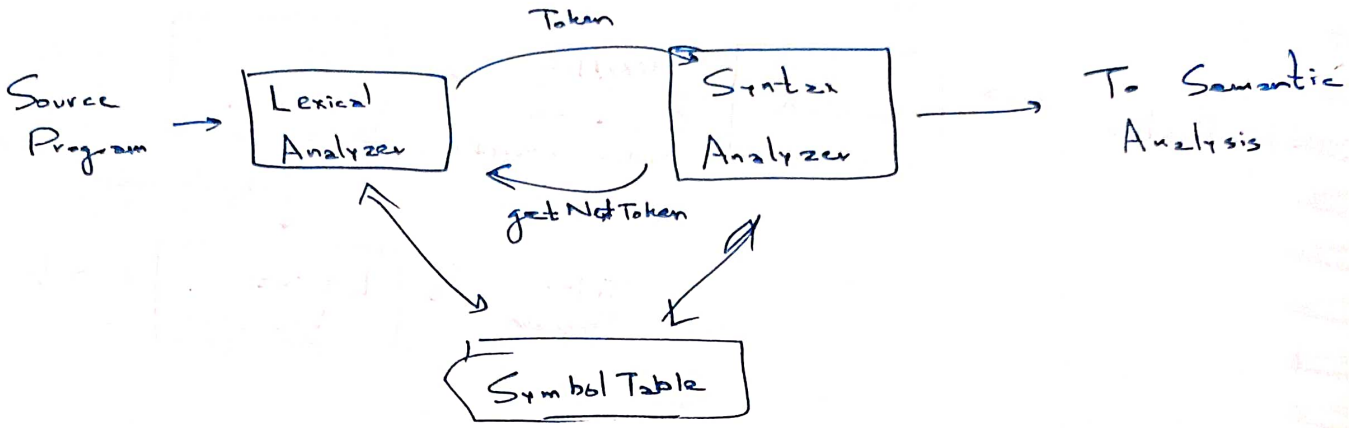
Implementing Transition Diagram

```
int state = 0;

token nextToken (){
  while (1){
    switch (state){
      case 0:  c = nextchar ();
               if (c == blank || c == tab || c == newline) {
                 state = 0;
                 lexeme_beginning ++;
               }
               else if (c == '<') state = 1;
               else if (c == '=') state = 5;
               else if (c == '>') state = 6;
               else state = fail();
               break;

      case 1: ...
              ;
      case 10: c = nextchar ();
               if (isletter (c)) state = 10;
               else if (isdigit (c)) state = 10;
               else state = 11;
               break;
      }
  }
}
```

Lex


Token Specification
LEX
lex.yy.c
C code for lexical analyzer
C Compiler
object code
Input Program
Lexical Analyzer
tokens.

Syntax Analyzer



→ checks syntax of input to determine if it is correct or not

→ also keeps track of symbol via symbol table.

→ Syntax can be defined using Context Free Grammar

$$G = (V, \Sigma, P, S)$$

- V: finite variables
- Σ: terminals
- P: finite Production Rules
- S: Starting Variable.

Eg: $E \rightarrow E+E \mid E*E \mid (E)$ [grammar for arithmetic symbol]

→ from

Eg: $E \rightarrow E+E \mid E*E \mid (E) \mid id$ [grammar for arithmetic symbol]

→ from this grammar "id + id * id" is derivable

$E \rightarrow E*E \rightarrow E+E*E$

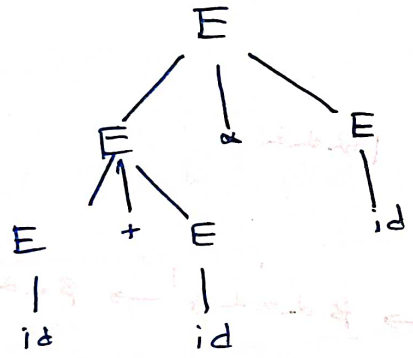
↓

$id+E*E$

↓

$id+id*id$

& corresponding parse tree



[leftmost derivation]

$E \rightarrow E+E \rightarrow E+E*E$

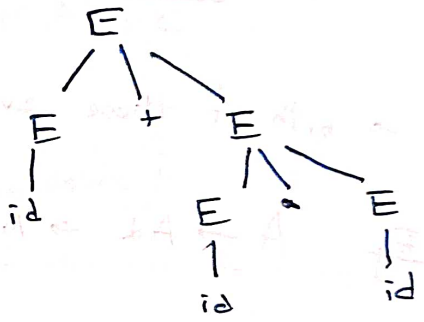
↓

$E+E*id$

↓

$id+id*id \leftarrow E+id*id$

& corresponding parse tree



[right most derivation]

→ Grammar can be ambiguous (should we use leftmost or rightmost derivation ?)

→ ~~We~~ In best case, grammar should not be ambiguous.

Elimination of Left Recursion

→ Rules which can recursively grow longer (by variable being in left) can cause problem in some situations

Eg:

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

⇓ we convert it

$A \rightarrow \beta A'$

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

here variable on right, easy for top down parser.

→ both of these are same

Eg $A \rightarrow A\alpha \rightarrow A\alpha\alpha \rightarrow A\alpha\alpha\alpha \rightarrow \beta\alpha\alpha\alpha$

$A \rightarrow \beta A' \rightarrow \beta\alpha A' \rightarrow \beta\alpha\alpha A' \rightarrow \beta\alpha\alpha\alpha A' \rightarrow \beta\alpha\alpha\alpha$

a) Remove Left Recursion if present

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

$A \to A\alpha \mid \beta$

$\Downarrow$

$A \to \beta A'$

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

(I) $E \to E+T \mid T$ $\Longrightarrow$ $E \to TA'$

$A' \to +TA' \mid \varepsilon$

(II) $T \to T*F \mid F$ $\Longrightarrow$ $T \to FA'$

$A' \to *FA' \mid \varepsilon$

(III) $F \to (E) \mid id$ [not left recursive]

→ In general for

$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \text{----} \mid \beta_1 \mid \beta_2 \mid \beta_3 \text{ ----}$

$\Downarrow$

$A \to \beta_1 A' \mid \beta_2 A' \mid \text{---}$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \text{----} \mid \varepsilon$

Left Factoring of Grammar

consider $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ (ambiguos)

$\Downarrow$

$A \rightarrow \alpha A'$

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

→ here β_1 & β_2 will be a token so choice will be based on next input

Eg:

statement : if expression then statement else statement

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

$E \rightarrow b$

$\Downarrow$

$S \rightarrow iEtSA' \mid a$

$A' \rightarrow \varepsilon \mid eS$

$E \rightarrow b$

Different Types of Parsers

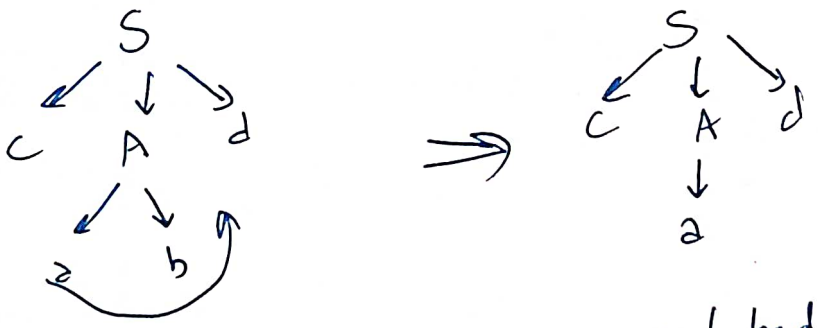
(I) Top Down Parser
- (i) Recursive Descent Parsing.
- (ii) Predictive Parser
- (iii) Non-Recursive Predictive Parser.

(II) Bottom-Up Parsing.
- (i) Shift Reduce Parser
- (ii) Operator Precedence Parser
- (iii) LR Parser → SLR, LALR (look ahead LR), Canonical LR
  - input from left
  - use rightmost parse

Eg: consider $S \to cAd$, input str. $w =$ "cad"

$A \to ab \mid a$

(i) Recursive Descent → Recursively check every case



did not match
back track
& try other option

matched.

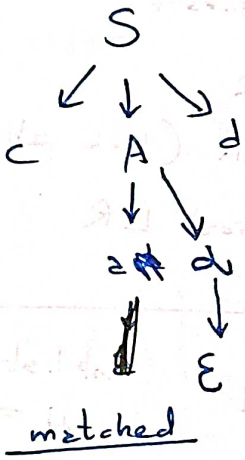
(ii) **Predictive Parser** : use left factorization & then just keep matching.

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

$\Rightarrow$

$S \to cAd$
$A \to a\alpha$
$\alpha \to b \mid \varepsilon$ } left factor

$w =$ "cad"



matched

Model of a Table Driven Nonrecursive Predictive Parser

(i) Input

(ii) Stack

(iii) Predictive Parsing Program

(iv) Parsing Table (M)

(v) Output

Eg: consider following Grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

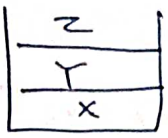
→ it's coressponding Parsing Table (how will come later) be

Non terminal ↓ \ Terminal	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

consider parsing " id + id * id "

Stack	Input	Output	Note
$E	id + id * id $		Initial Step always like this
$E'T	id + id * id $	E → TE'	E/id in M
$E'T'F	id + id * id $	T → FT'	T/id in M
$E'T'id	id + id * id $	F → id	F/id in M
$E'T'	+ id * id $		Terminal in stack, remove input tok
$E'	+ id * id $	T' → ε	T'/+ in M
$E'T+	+ id * id $	E' → +TE'	E'/+ in M
$E'T	id * id $		Terminal in stack remove +
$E'T'F	id * id $	T → FT'	T/id in M
$E'T'id	id * id $	F → id	F/id in M
$E'T'	* id $		Token/input remn
$E'T'F*	* id $	T' → * FT'	T'/* in M
$E'T'F	id $		Tok/In rem
$E'T'id	id $	F → id	F/id in M
$E'T'	$		Tok/In Rem
$E'	$	T' → ε	$/T' in M.
$	$	E' → ε	$/E' in M
Parsing Done	Parsing Done		Tok/In Rem

→ stack



(i) if $X = a = \$$ (terminal), the parser halts & announces successful completion of parsing

(ii) if $X = a \neq \$$ (terminal) the parser pops off the stack & advances the pointer to the next input token

(iii) If X is non terminal, the program consults $M[X, a]$ of parsing table M. The entry will either be an X-production of grammar or nothing (in this case, we throw error)

Eg: if $M[X, a] \Rightarrow \{X \rightarrow UVW\}$, the parser replaces X on top of stack by WVU (U on top)

LL(1) Grammar

→ A grammar whose parsing table has no multiply defined entries is said to be LL(1).

→ LL(1)
- 1 → use one input symbol of lookahead at each step for decision.
- L → scan input from L to R
- L → use Leftmost derivation.

Construction of Parsing Table

→ we use two functions FIRST() & FOLLOW()

$G = (V, \Sigma, P, S)$

(I) FIRST (X) → compute for all grammar symbol $X \in V$

(i) if X is a terminal, then FIRST (X) = {X}

(ii) if $X \to \varepsilon$ is a production, then add ε to FIRST(X)

(iii) if X is a non-terminal & $X \to Y_1 Y_2 \dots Y_n$ is a production, add FIRST(Y_1) to FIRST (X)

Eg: $E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$V = \{E, T, E', F, T'\}$

$\Sigma = \{+, *, id, (,)\}$

(a) FIRST (F) = FIRST ((E)) ∪ FIRST (id) [F → (E) | id]

= FIRST (( ) ∪ {id} [(iii) & (i)]

FIRST(F) = {(, id}

(b) FIRST (E) = FIRST (TE') [E → TE']

= FIRST (T) [(iii)]

= FIRST (FT') [T → FT']

= FIRST (F) [(iii)]

FIRST (E) = {(, id}

FIRST (T) = {(, id}

NOTE: CFG will have one non terminal in LHS

(c) FIRST(E') = FIRST(+TE') ∪ FIRST(ε) [E' → +TE' | ε]

= FIRST(+) ∪ {ε} [iii & ii]

= {+} ∪ {ε} [i]

FIRST(E') = {+, ε}

(d) FIRST(T') = FIRST(*FT') ∪ FIRST(ε) [T' → *FT' | ε]

= FIRST(*) ∪ {ε} [iii & ii]

= {*} ∪ {ε} [i]

FIRST(T') = {*, ε}

(II) compute FOLLOW(X) for all non-terminals A X ∈ V

(i) Place $ in FOLLOW(S) [S is the start symbol of grammar]
[$ is the input right end marker]

(ii) If there is a production A → αBβ, then everything in FIRST(β) except ε will be placed in FOLLOW(B)

(iii) If there is a production A → αB or A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) will be placed in FOLLOW(B)

Eg: (a) FOLLOW(E)

→ E is starting symbol → {$} [i]

→ rules where E in RHS

(I) F → (E)
A α B β

[FIRST( ) = {) }]

→ rule (ii) ⇒ {) }

∴ FOLLOW(E) = { $,) }

(b) FOLLOW(E')

Rules → (I) $\underset{A}{E} \rightarrow \underset{\alpha}{T}\ \underset{B}{E'}\ \underset{\beta}{}$ [FIRST(β) = {ε}] [(iii)]

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

→ (II) $\underset{\alpha}{E'} \rightarrow \underset{\alpha}{+}\ \underset{B}{T}\ \underset{\beta}{E'}$ [FIRST(β) = {ε}] [(iii)]

FOLLOW(E') = FOLLOW(E')

(c) FOLLOW(T)

rules with T in RHS → (I) $\underset{A}{E} \rightarrow \underset{\alpha}{T}\ \underset{B}{E'}\ \underset{\beta}{}$ [FIRST(E') = {+, ε}]

rule (iii) → FOLLOW(ET) = FOLLOW(E)
= {$,)}

~~rule (ii) → FOLLOW(T) =~~

(II) $\underset{A}{E'} \rightarrow \underset{\alpha}{+}\ \underset{B}{T}\ \underset{\beta}{E'}$ [FIRST($\overset{E'}{\beta}$) = {ε, +}]

rule (iii) → FOLLOW($\overset{T}{E'}$) = FOLLOW(E') = {$,)}

rule (ii) → FOLLOW(T) = FIRST(E') = ~~{+, ε}~~ {ε}
= {+}

∴ FOLLOW(T) = {$,), +}

(d) FOLLOW(T')

Rule (I) $\underset{A}{T} \rightarrow \underset{\alpha}{F}\ \underset{B}{T'}\ \underset{\beta}{}$ [FIRST(β) = {ε}]

rule (iii) → FOLLOW(T') = FOLLOW(T) = {$,), +}

(II) $\underset{A}{T'} \rightarrow \underset{\alpha}{*}\ \underset{B}{F}\ \underset{\beta}{T'}$ [FIRST(β) = {ε}]

rule (iii) → FOLLOW(T') = FOLLOW(T')

(e) FOLLOW(F)

→ Rules with F in RHS

(I) $T \to FT'$ (A → αBβ) [FIRST(β) = FIRST(T') = {*, ε}]

rule (iii) → FOLLOW(F) = FOLLOW(T) = {$,), +}

rule (ii) → FOLLOW(F) = FIRST(T') − {ε}

= {*}

∴ FOLLOW(F) = {$,), +, *}

(III) Use FIRST() & FOLLOW() to make parsing Table,

Eg: (a) $E \to TE'$

FIRST(E) = {(, id} → add rule to [E, (] & [E, id]

(b) $E' \to +TE'$

* consider only this rule

FIRST(E') = FIRST(+TE') = {+} → add rule to [E', +]

(c) $E' \to \varepsilon$

FIRST(E') = {ε} → no addition possible

↳ FOLLOW(E') = {$,)} → rule in [E', $] & [E',)]

(d) $T \to FT'$

FIRST(T) = {(, id} → [T, (], [T, id]

(e) $T' \to *FT'$

here FIRST(T') = {*} → [T', *]

(f) $T' \to \varepsilon$

FIRST(T')$_{here}$ = {ε} ⇒ FOLLOW(T') = {$,), +}

↳ [T', $], [T',)]

[T', +]

(g) $F \to (E)$

FIRST(F) = {(}

(h) $F \to id$

FIRST(*F) = {id}

hence resulting table

Non Terminal \ Terminals	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Implementation Detail of Non Recursive Predictive Parser

- → Table can be implemented in Dictionary / Array / etc.
- → Stack is already present
- → Input Buffer can be thought as an array

Q) $S \to iEtSS' \mid a$

$S' \to eS \mid \varepsilon$

$E \to b$

compute FIRST( ), FOLLOW( ) of every non terminals, make parsing Table, is this grammar LL(1)?

Bottom Up Parsing

→ we go from leaf to root [i.e we go reverse]

Eg: $S \to aABe$

$A \to Abc \mid b$ $\quad \omega = "abbcde"$

$B \to d$

String	Handle
a b b c d e	$A \to b$
a A b c d e	$A \to Abc$
a A d e	$B \to d$
a A B e	$S \to aABe$
S	[accept]

→ The Substring we choosing here are called handle

→ Choosing Proper Handle is the challenge

Handle Pruning

→ Technique in which we do Right Most Derivation in Reverse.

Eg: $E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to id$

→ E is start symbol

→ $\omega = "id_1 + id_2 * id_3"$

Right Sequential Form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

[look in reverse way, you can see RMD]

STack Implementation of Shift Reduce Parser

→ Programmatic Method of Implementing Handle Pruning

Three Units →
- Stack
- Input Buffer
- Program Executing Parsing

4 Actions →
- Shift: Shift current element of input buffer into stack
- Reduce: Take Top Element(s) of Stack and execute reverse production
- Accept: Accept the Input as a valid production (at end of parsing [only $ in Input])
- Error: Throw error if the Input is not accepted.

Eg: $E \to E+E$
$E \to E * E$
$E \to (E)$
$E \to id$

$\omega = id_1 + id_2 * id_3$

Stack	Input	Action
$	$id_1 + id_2 * id_3 \$$	Shift
$ id_1	$+ id_2 * id_3 \$$	Reduce by $E \to id$
$ E	$+ id_2 * id_3 \$$	Shift
$ $E+$	$id_2 * id_3 \$$	Shift
$ $E+id_2$	$* id_3 \$$	Reduce by $E \to id$
$ $E+E$	$* id_3 \$$	Shift [not Reduce cause RMD]
$ $E+E*$	$id_3 \$$	Shift
$ $E+E*id_3$	$\$$	Reduce by $E \to id$
$ $E+E*E$	$\$$	Reduce by $E \to E*E$
$ $E+E$	$\$$	Reduce by $E \to E+E$
$ E	$\$$	Accept

NOTE → Ambiguity about choosing Shift or Reduce is still present
→ Solution to this is by making a Parsing Table

Bottom Up Approach → Shift Reduce Parser] we are here.
→ Operator Precedence Parsing
→ LR Parsing → SLR
→ LLR
→ LALR
] uses table

→ Each method coming forward will have different method of generating Parsing Table.

→ Parsing Proceduce will be same as we did now.

Operator Precedence Parsing

Operator Grammar

→ Operator Precedance Parser can only handle operator grammar

(a) No Production Rule have ε in right side

(b) No Production rule which have two adjacent non-terminals on right side

consecutive non term

empty

Eg: $E \rightarrow EAE \mid (E) \mid -E \mid id$ } → Not Operator Grammar

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

⇓ convert

Eg: $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid -E \mid id$ } → operator grammar

(E)

Operator Precedence Parsing Table

Relation	Meaning
$a < b$	a yields precedence to b
$a = b$	a has same precedence as b
$a > b$	a takes precedence over b

→ These three relations are used to generate parsing table.

Eg: assume following parsing table. (how will come later)

first symbol \ second symbol	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

and we parsing w = " id + id * id "

(I) add delimeter to both side

$id + id * id $

↓ using table, add <, > & =

$ <id> + < id > * < id > $

↓ we use handle (rule) which are between < >

[E → id]

$ E + <id > * <id >$

↓ [E → id]

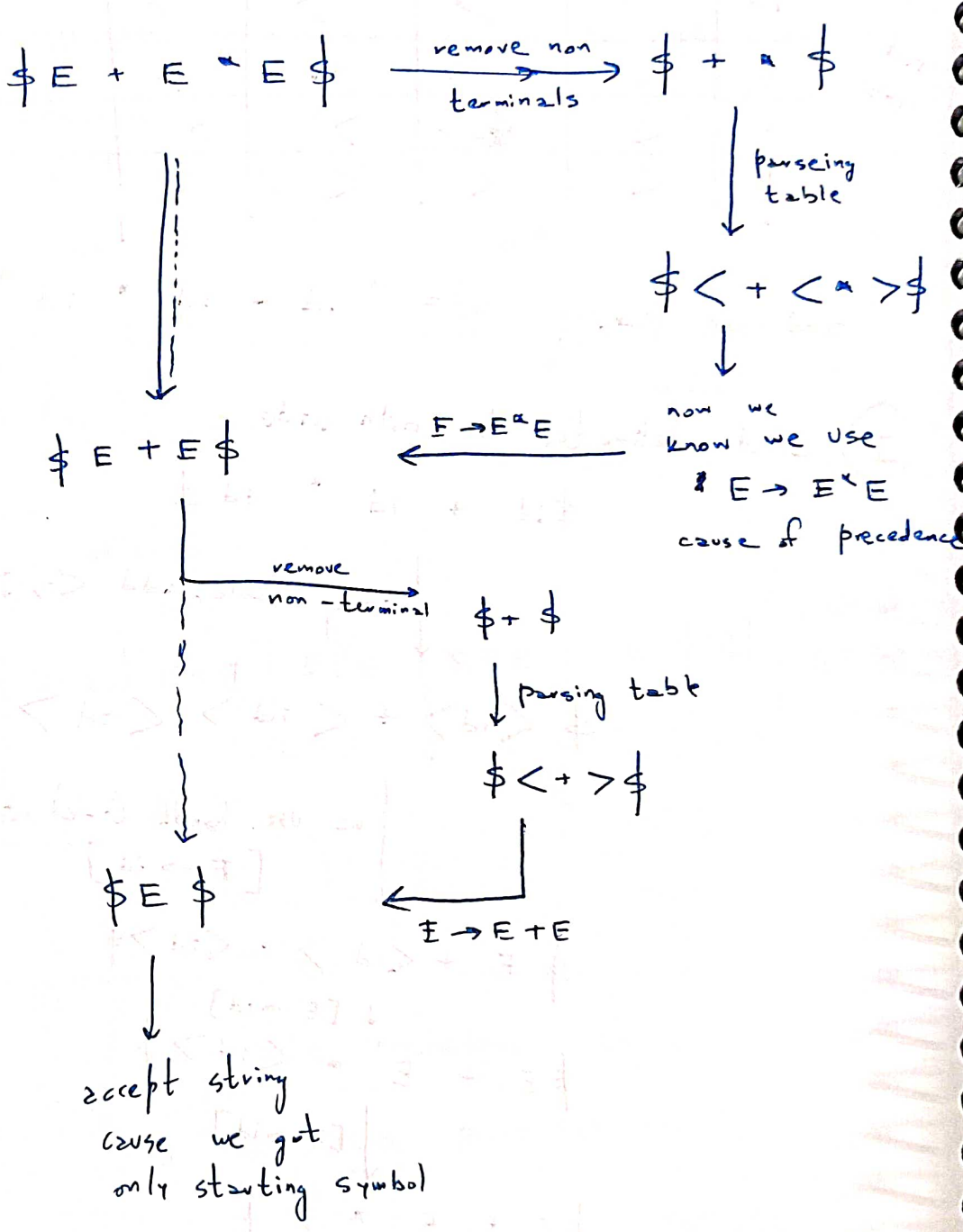
$ E + E * <id > $

↓ [E → id]

$ E + E * E $

$ E + E * E $

→ now there is a collision between what to parse, in this case, we follow



Rules to follow while making Operator Precedence Parsing Table

(1) If operator O_1 has higher precedence (according to BODMAS) than operator O_2, then $O_1 > O_2$ & $O_2 < O_1$

(2) If O_1 & O_2 have equal precedence, then
$O_1 > O_2$ & $O_2 > O_1$ (follow left associativity)

(3) Relations b/w $, id & operators & brackets

(i) $O < id$; $id > O$

(ii) $O < ($; $(> O$

(iii) $) > O$; $O <)$

(iv) $O > \$$;

(v) $\$ < ($;

(vi) $\$ < id$

(vii) $(< ($

(viii) $id > \$$

!

etc (use common sense & badmas)

Notes

→ This Parsing can only handle a small class of grammars (Called operator grammar)

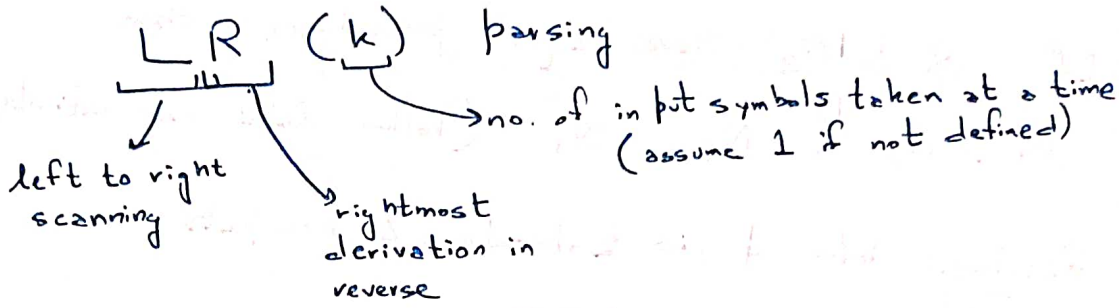
→ There is no way to define unary operators :

Solution → LR Parsing Table which can handle any grammar.

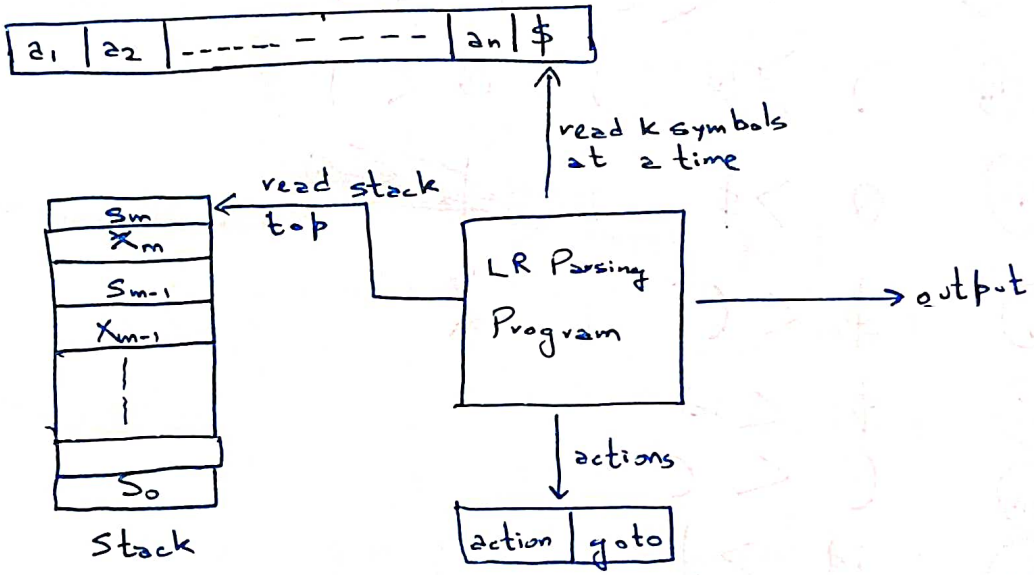
LR Parsing

→ used in yacc

→ any CFG (hence any programming language) can be parsed using this



LR Parsing Model



→ Three different methods of constructing parsing Table :

(i) Simple LR [SLR]

(ii) Canonical LR [CLR]

(iii) Look Ahead LR [LALR]

→ SLR simple but least powerful

→ CLR complex but most powerful

→ LALR is intermidiate

How is Parsing Done?

Eg: consider grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

→ consider following SLR parsing table for the grammar (how coming later)

Stack State	action (terminals)						goto (non-term)		
	id	+	*	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				accept			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		r_1 / s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

r_j → reduce stack element using rule no. j

s_j → shift curr input to stack and ~~store~~ push stack state j

acc → accept

blank → throw error

consider ω = " id * id + id "

Stack ~~State~~	input buffer	Action
0	id * id + id $	S_5
0, id, 5	* id + id $	r_6 [F → id]
0, F, 3 (used goto of)	* id + id $	r_4 [T → F]
0, T, 2 (used goto of 0,T)	* id + id $	S_7
0, T, 2, *, 7	id + id $	S_5
0, T, 2, *, 7, id, 5	+ id $	r_6 [F → id]
0, T, 2, *, 7, F, 10 (goto, 7, F)	+ id $	r_3 [T → T * F]
0, T, 2 (goto, 0, T)	+ id $	r_2 [E → T]
0, E, 1 (goto, 0, E)	+ id $	S_6
0, E, 1, +, 6	id $	S_5
0, E, 1, +, 6, id, 5	$	r_6 [F → id]
0, E, 1, +, 6, F, 3 (goto, 6, F)	$	r_4 [T → F]
0, E, 1, +, 6, T, 9 (goto, 6, T)	$	r_1 [E → E + T]
0, E, 1 (goto, 0, E)	$	accept

→ after reduce, we also add coressponding goto of non-term being added

→ if rule has α variables in RHS, we pop 2α symbols from stack

NOTE

→ we didn't have to remove left recursion or ambiguity

→ we add/read/er variables in symbol table while we parse

Constructing SLR Parsing Table

LR(0) item → " . " in the RHS of production rule

Eg: A → XYZ

⇓

A → . XYZ
A → X . YZ
A → XY . Z
A → XYZ .

→ To make SLR Parsing Table, we need canonical LR(0) items by augmenting the grammar

→ To make canonical LR(0) items, we use closure & goto

~~(I) we augment grammar by intro~~

→ assume we making parsing table for previous grammar:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

(I) we augment grammar by introducing new starting variable

E' → E

→ if new rule is used, we can safely announce that syntax of sentence is correct [we remember this is bottom up parsing]

→ let I = set of one item = {[E' → . E]}

→ we use two functions closure (I) & goto (I) to find canonical LR(0) items

I_0 = Clousure (I) = { [E' → .E], ——— → initial item

[E → .E +T], [E → .T], ——— → any other rule which have E as start symbol as closure has .E

[T → .T * F], [T → .F], ——— → any other rule which have T as start symbol as closure have ".T"

[F → .(E)], [F → .id] } ——— → any other rule which have F as starting symbol as ".F" in closure

(keep . in start of rule)

→ after . , we can get terminal or non-terminal, for all those symbols, we keep finding goto(I) till we find all ~~gto~~ unique goto's, (there will be "loops", we stop that path then)

(II) goto (I_0, E) = { [E' → E.], [E → E. + T] } ——— → all item which had .E in RHS, . shifted one pos right

= I_1 (unique)

(III) goto (I_0, T) = { [E → T.], [T → T. * F] } = I_2 (unique)

(IV) goto (I_0, F) = { [T → F.] } = I_3 (unique)

(V) goto(I_0, () = { [F → (. E)], → . after non terminal, add closures of E

[E → . E+T],

[E → . T], → . after non-terminal, add closures of T

[T → . T * F],

[T → . F], → . after non-terminal, add closure of F

[F → .(E)],

[F → . id] } = I_4 (unique)

non-terminal, also add closure(Io)

(VI) goto(I_0, id) = { [F → id.] } = I_5 (unique)

(VII) goto(I_1, +) = { [E → E + . T], → . after non-terminal, add closures of non-term (from production rules)

[T → . T * F],

[T → . F],

[F → .(E)],

[F → . id] } = I_6 (unique)

(VIII) goto(I_2, *) = { [T → T * . F], → adding closure of F

[F → .(E)],

[F → . id] } = I_7 (unique)

(IX) goto(I_4, E) = { [F → (E .)],

[E → E . + T], } = I_8 (unique)

(X) goto(I_4, T) = I_2

(XI) goto(I_4, F) = I_3

XII goto $(I_4, C) = I_4$

XIII goto $(I_4, id) = I_5$

XIV goto $(I_6, T) = \{ [E \to E+T.], [T \to T. * F] \} = I_9$ (unique)

XV goto $(I_6, F) = I_3$

XVI goto $(I_6, C) = I_4$

XVII goto $(I_6, id) = I_5$

XVIII goto $(I_7, F) = \{ [T \to T*F.] \} = I_{10}$ (unique)

XIX goto $(I_7, C) = I_4$

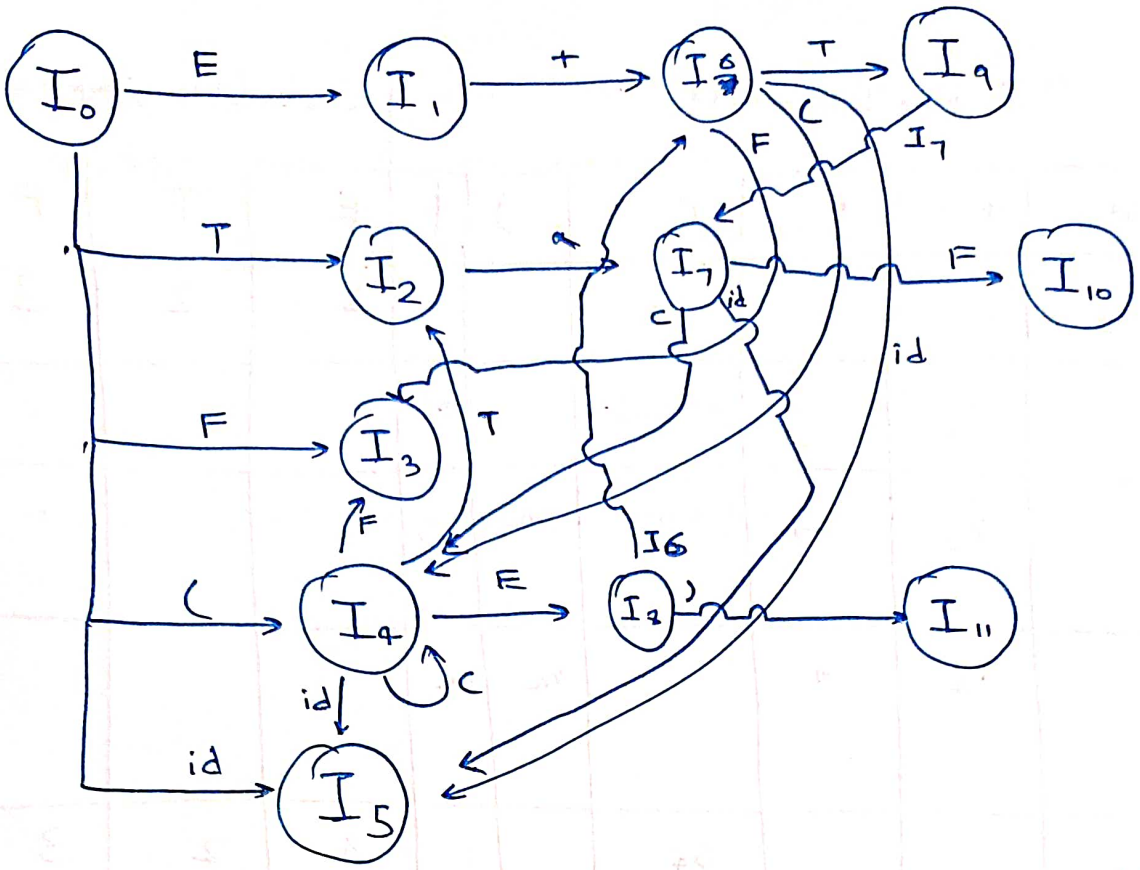
XX goto $(I_7, id) = I_5$

XXI goto $(I_8,)) = \{ [F \to (E).] \} = I_{11}$ (unique)

XXII goto $(I_8, +) = I_6$

XXIII goto $(I_9, *) = I_7$

→ we essentially tracked all possible ways the grammar can go & found 11 different & unique states



(just for demonstratio

→ now we make table using gotos

goto $(I_\alpha, \beta) = I_\gamma \Rightarrow$ action $[\alpha, \beta] = S_\sigma$ if β non term

goto $[\alpha, \beta] = \gamma$ if β term

→ in cases $A \to \alpha.$ for any state I_β, we use FOLLOW(A) to find where to put r_k, where k is the production rule no.

Eg $I_2 \Rightarrow [E \to T.] \Rightarrow$ FOLLOW(E) = { $, +,) }

↳ rule (2)

↓ starting symbol ↓ E→E+T ↳ F→(E)

(DIY)

so action $[2, \$] = r_2$

action $[2, +] = r_2$

action $[2,)] = r_2$

→ If ~~at end~~ goto possible → S

→ If . at end → FOLLOW()

↳ $, ACC if we have augmented rule $[E' \to E]$ in goto set

State	Action (Terminals)						Goto (Non-Terminal)		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				ACC			
2 (FOLLOW(E) = {+,), $}		r_2	S_7		r_2	r_2			
3 FOLLOW(T) = {+, *,), $}		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5 (FOLLOW(F) = {+, *,), $}		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9 FOLLOW(E) = {+,), $}		r_1	S_7		r_1	r_1			
10 FOLLOW(T) = {+, *,), $}		r_3	r_3		r_3	r_3			
11 FOLLOW(F) = {+, *,), $}		r_5	r_5		r_5	r_5			

[Same as before]

→ There can be cases when one location can have multiple steps which causes ambiguity

→ yacc follows the first step it found in case of ambiguity

→ DRAWBACK → we have to use FOLLOW() ~~for~~

→ SOLUTION → ~~LALR~~ Canonical LR table

Canonical LR

Construction of Canonical LR parsing table

→ we include extra info with items called look ahead

→ General form of an item becomes

$$[A \to \alpha . \beta \ , \ a]$$

→ This item is called LR(1) item

↑ amount of extra info used

↑ look ahead

where $A \to \alpha\beta$ is a production rule
& a is terminal / right end marker.

→ because of Look Ahead, we don't need to find ~~Pt~~ FOLLOW() anymore.

How look ahead gets calculated?

→ for an item $[A \to \alpha . B \beta \ , a]$, look ahead for it's closure = FIRST(βa).

Eg $[S' \to . S \ , \$]$

→ ~~closure~~ look ahead for it's ~~look ahead~~ closure = FIRST($\$$) = {$}

Eg: $[S \to . cC \ , \ \cancel{\$} \ a]$

~~→ closure for it's to~~

→ look ahead for it's closure = FIRST(cCa) = ~~closure(c)~~

= FIRST(Ca)

= FIRST(C)

= ..... (according to grammar rule)

→ This is followed ~~only~~ only for closures & not gotos

Eg: $S' \rightarrow S$ [augment rule]

1. $S \rightarrow CC$
2. $C \rightarrow cC$
3. $C \rightarrow d$

(I) augment gramenar

(II) $I_0 = \{[S' \rightarrow .S, \$]\}$ ← initial state

(III) I_0 = closure (I) = { ← same as before
$[S' \rightarrow .S, \$]$,
$[S \rightarrow .CC, \$]$,
$[C \rightarrow .cC, c/d]$, } → look ahead from
$[C \rightarrow .d, c/d]$ }
this

(IV) goto $(I_0, S) = \{[S' \rightarrow S., \$]$ ← look ahead of before $= I_1$ (unique)

(V) goto $(I_0, C) = \{[S \rightarrow C.C, \$]$,
$[C \rightarrow .cC, \$]$,
$[C \rightarrow .d, \$]\} = I_2$ (unique)

(VI) goto $(I_0, c) = \{[C \rightarrow c.C, c/d]$,
$[C \rightarrow .cC, c/d]$,
$[C \rightarrow .d, c/d]\} = I_3$ (unique)

(VII) goto $(I_0, d) = \{[C \rightarrow d., c/d]\} = I_4$ (unique)

(VIII) goto $(I_2, C) = \{ [S \to CC.\ ,\ \$] \} = I_5$

(IX) goto $(I_2, c) = \{ [C \to c.C\ ,\ \$],$
$[C \to .cC\ ,\ \$],$
$[C \to .d\ ,\ \$] \} = I_6$ (unique)

(X) goto $(I_2, d) = \{ [C \to d.\ ,\ \$] \} = I_7$ (unique)

(XI) goto $(I_3, C) = \{ [C \to cC.\ ,\ c/d] \} = I_8$ (unique)

(XII) goto $(I_3, c) = I_3$

(XIII) goto $(I_3, d] = I_4$

~~(XIV)~~ (XIV) goto $(I_6, C) = \{[C \to cC.$ ~~$= I_6$~~
$= \{ [C \to cC.\ ,\ \$], \} = I_9$

(XV) goto $(I_6, c) = I_6$

(XVI) goto $(I_6, d] = I_7$

→ now we found canonical LR(1) items, now we make Table

→ How to fill reduce rules?

Eg: $I_4 = \{ [C \to d.\ ,\ c/d] \} \Rightarrow$ ~~rule 3~~ r_3 @ c&d

rule 3

→ as before if goto has augment rule, we add ACC

State	action (term)			goto (non-term)	
	c	d	$	S	C
0	S_3	S_4		1	2
1			ACC		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

→ here we only need to calculate FIRST(C) in closure, hence simpler than SLR

LALR

→ Steps

1. Construct LR(1) items (same as CLR)
2. ~~st~~ union the states that have same type of production rules but different look ahead

Eg in previous grammar

$I_3 = \{[C \to c.C, c/d],$
$[C \to .cC, c/d],$
$[C \to .d, c/d]\}$

$I_6 = \{[C \to c.C, \$],$
$[C \to .cC, \$],$
$[C \to .d, \$]\}$

we union them as $I_{36} = I_3 \cup I_6$

$= \{[C \to c.C, c/d/\$],$
$[C \to .cC, c/d/\$],$
$[C \to .d], c/d/\$]\}$

similarly

$I_4 = \{[C \to d., c/d]\}$ $I_7 = \{[C \to d., \$]\}$

union → $I_{47} = I_4 \cup I_7 = \{[C \to d., c/d, \$]\}$

similarly

$I_{89} = \{[C \to cC., c/d/\$]\}$

→ resultant table

state	action			goto	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			ACC		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

→ This reduces the states of CLR

Error - Recovery Strategies

Types of Parser

(I) **Universal**

→ Can parse any grammar
  (i) Cocke - Younger - Kasami Parsing Method
  (ii) Earley's Algorithm
→ Inefficient to be used in production of ~~grammar~~ compilers

(II) **Top - Down**

→ builds parse tree
→ Starts parsing from top (root) to the bottom (leaves)

(III) **Bottom - Up**

→ starts from leaves and work their way upto the root.

→ Both in Top-Down & Bottom-Up, we
  → scan the input from left to right
  → scan one symbol at a time.

What should happen when parser finds error in user input?

(I) Stop immidiately & signal error

or (II) Record error but try to continue.

→ In (I), user must recompile from scratch after trivial fix
→ In (II), user might be overwhelmed by a whole series of error messages, all caused by essentially the same problem.

Types of Errors

(i) **Lexical Error**
→ misspelling an identifier, keyword or operator

(ii) **Syntactic Error**
→ Eg: arithmetic expression with unbalanced parenthesis

(iii) **Semantic Error**
Eg: operator applied to an incompatible operand

(iv) **Logical Error**
Eg: infinitely recursive call.

Error - Recovery Strategies

(I) Panic Mode
(II) Phrase Level Recovery
(III) Error Production
(IV) Global Correction.

Panic Mode Error Recovery

→ Idea: skip input symbol until a symbol comes which is in a set of synchronizing token

Eg:
```
a = b+c   //no semicolon
d=e+f;
```
→ The compiler will discard ("d = e + f") all subsequent tokens till a semicolon is encountered.

Phrase Level Recovery

→ Performs local correction on the input to repair the error.

→ Changes input streams by inserting missing token.

Eg int id 5; ⇒ int id = 5;
(= added)

Error Production

→ Augment grammar with productions for erroneous constructs

Global Corrections

→ Choose a minimal sequence of changes to obtain a global least-cost correction.

Error Recovery in Predictive Parsing

~~Error~~

→ An Error detected ~~using~~ during predictive parsing when (i) the terminal on the top of stack does not match with the next input symbol.

(ii) → when non-terminal A is on top of the stack, a is next input symbol, & the parsing table entry M[A,a] is empty.

→ Some ~~Hue~~ Heuristics for the choice of Synchronizing Set are as follows :

(i) As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for the non-terminal A. Skip the tokens until an element of FOLLOW(A) is seen & pop A from the stack, it is likely that parsing can continue [Panic Mode]

(ii) If we add symbols in FIRST(A) to the synchronizing set for non-terminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in input.

(iii) If a terminal on top of a stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted and continue the parsing. [Phrase Level]

Eg: $E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

(DIY using ~~F~~ rules mention before)

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

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

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

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

Non Terminal \ Term	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

~~sync~~ → synch added using FOLLOW() of resp. non-terminal

~~→ Synch added in all places~~

→ Synch : the driver pops current non-terminal A & skips input till synch token OR skips ~~unit~~ until one of FIRST(A) is found (i.e production rule present)

→ If the parser looks up entry M[A,a] and finds it blank, the input symbol a is skipped.

→ If the entry is synch, the non terminal on top of stack is popped.

→ If a token on top of stack does not match the input symbol, then we pop the token from stack. (i.e add it to input)

Eg: consider parsing w = ") id "+id $ " using prev parsing table

Non-Term \ Term	id	+	*	(	)	$
E	$E \to TE'$			$E' \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	synch		$T' \to FT'$	synch	synch
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

Stack	input	output
$E	) id * +id $	synch error, skip) (not pop cause nothing in stack of then)
$E	id * +id $	E → TE' ~~reduce~~
$ E'T	id *+id $	T → FT'
$E' T'F	id * +id $	F → id
$E'T'id	id * +id $	~~non~~ terminal match
$E'T'	* +id $	T' → * FT'
$ E'T'F *	* +id $	terminal match
$E'T' F	+ id $	synch error, pop F
$E'T	+ id $	synch error, pop T
$E'	+ id $	E' → + TE'
$ E'T+	+ id $	terminal match
$ E'T	id $	T → FT'
$E'T'F	id $	F → id
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	~~non term etc~~ Accept

Phrase Level Recovery in Predictive Parsing

I Change input stream by inserting missing tokens

Eg: "id id" ⇒ "id * id"

Non Term \ Term	id	+	*	(	)	$
E	$E \to TE'$			$E' \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	synch		$T \to FT'$	Synch	synch
T'	Insert *	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	synch	synch	$F \to (E)$	Synch	synch

Error Production Recovery in Predective Parsing

→ include productions for common error

→ these production will make parser pop non terminal, essentially "adding token" in input stream"

Eg: add $T' \to FT'$ in $M[T', id]$

Non Term \ Term	id	+	*	(	)	$
E	$E \to TE'$			$E' \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'	$T' \to FT'$	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

NOTE → Phrase Level Recovery & Recovery via adding Error Production are very powerful, but needs manual additions

---

Error Handling in LR Parsing [Phrase Level]

→ for each error entry in table (blanks), insert a pointer to a particular error procedure, which assumes the most likely cause and takes appropriate action

→ NOTE: also need manual intervention.

Construction of Parsing Table

→ we use two functions FIRST() & FOLLOW()

$G = (V, \Sigma, P, S)$

(I) FIRST (X) → compute for all grammar symbol $X \in V$

(i) if X is a terminal, then FIRST(X) = {X}

(ii) if $X \to \varepsilon$ is a production, then add ε to FIRST(X)

(iii) if X is a non-terminal & $X \to Y_1 Y_2 \dots Y_n$ is a production, add FIRST(Y_1) to FIRST(X)

Eg: $E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$V = \{E, T, E', F, T'\}$

$\Sigma = \{+, *, id, (,)\}$

(a) FIRST(F) = FIRST((E)) ∪ FIRST(id) [$F \to (E) \mid id$]

= FIRST(( ) ∪ {id} [(iii) & (i)]

FIRST(F) = {(, id}

(b) FIRST(E) = FIRST(TE') [$E \to TE'$]

= FIRST(T) [(iii)]

= FIRST(FT') [$T \to FT'$]

= FIRST(F) [(iii)]

FIRST(E) = {(, id}

FIRST(T) = {(, id}

NOTE: CFG will have one non terminal in LHS