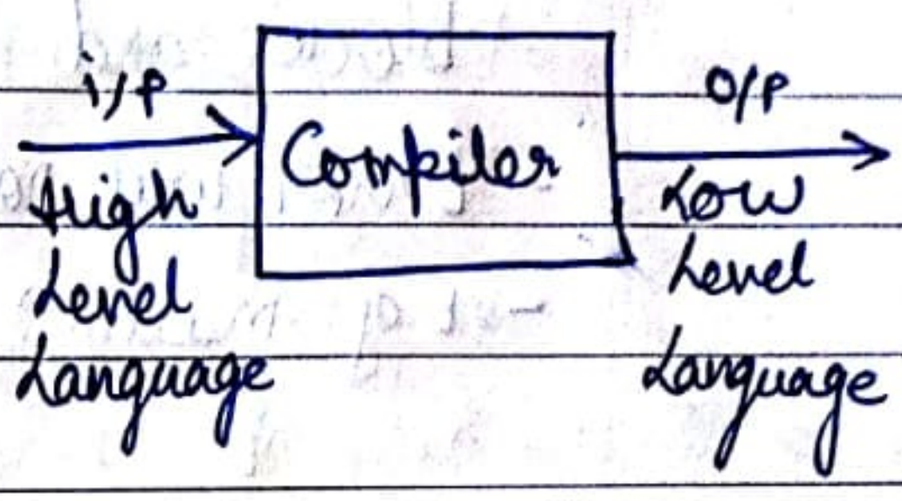
INTRODUCTION

1-12-'18

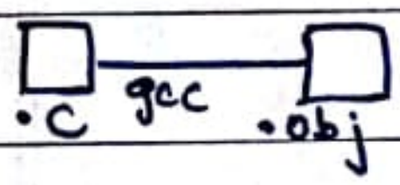
Introduction

- Translator (compiler).
- Converting high level language to low level.
- Lexical (Regular expression).

High Level Language	Machine Understandable Format
	Translation Process



- Different types of Phases:



Unit I
- Lexical analysis (use regular expressions)

Unit II
- Syntax analysis (generate parse tree)

Unit III
- Semantic analysis
- Intermediate code genaluon

Unit IV
- Code optimization } about memory
- Code generation
- ~~Optimization~~ Runtime Environments } Synthesis Phase

Unit V
- Optimization

P.T.O.

13-12-'18

UNIT-I : INTRODUCTION, LEXICAL ANALYSIS

Lexical Analysis

- In .c file, the invisible units are called lexical units. In c, identifiers, keywords, function declarations, arithematic operators, delimiters, blocks and comments are lexical units.
- Error will not be having a regular expression correctly
- Eg: main()
{
int a, b, c;
a = 5;
b = 6;
c = a+b;
printf(c);
}

main	+
int	=
a	;
b	(
c	)
printf	}
	{

They are invisible units

- In this, we have identifiers, i, n, t, keywords, arithmetic operators, relational operators, delimiters, ~~blocks~~, (;,)·····), blocks {, }
- 5, 6 are numbers.
- For Lexical Analysis:
  - Deletion of comment lines (free from comment lines)
  - First we need to identify the keyword (store all keywords in database), Delete delimiters
  - Differentiate between keywords and identifiers
  - delimiter → space, ; (,), {, }, enter
  - In linux or unix, it will contain a tool like LEXTOOL to identify the lexemeres (keyword)
    - to identify, we need to write the R.E
    - instead of consisting higher level language we write

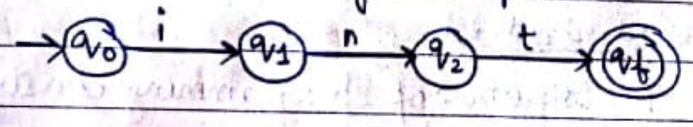
- LEXTOOL

For eg:

Instance of lexemens		tokens
main	keyword	
{	delimiter	
int	keyword	
a	identifier	
b	identifier	
c	identifier	

Instances of tokens are called lexemens

- For regular expressions, identifier should start with alphabet followed by alphabet or digit alphabet.
- It should be in Regular Expression



R.E for int

- Identifier → should start with alphabet (alphabet + digit)* identifier
- Alphabet [0···9] Digits [0···9]

- The output of lexical analysis will be tokens. [all the lexes are stored in a table called symbol tables]. Tokens will be input for syntax analysis phase.

Syntax Analysis

- Arrange the tokens in a systematic manner.
- Syntactical errors are identified in syntax analysis phase.

13-12-

Uses of Content Free Grammar

→ Write a content free grammar for declaration statement : int a,b,c;

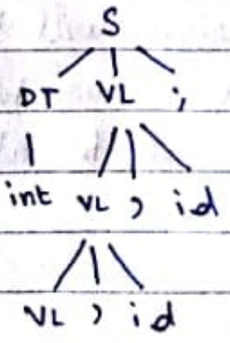
S → DT VL;
DT → int | float | char;
VL → VL, id | id;

int a,b,c;
↓ ↓
DT VL

- VL has single identifier or multiple identifiers (id id should be first

VL id, id
VL id, id, id
VL id, id, id, id

- The structure of programming constraints we need to draw the CFG.
- The given syntax is valid for existing CFG.
- To identify whether a sentence is valid or not, we will draw a parse tree



It will construct a parse tree

int id,id,id;

* The output of syntax analysis phase is a Parse Tree.

Semantic Analysis

- The meaning of a statement is analyzed.
- Type conversions are done in semantic analysis phase
- Type mismatch.

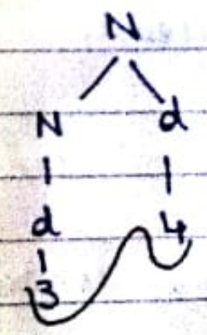
Eg: int a,a;
a = a * 5.5;
↳ Here, there is a type mismatch

Eg: int c,a;
float e;
b = a * 5.5;
↳ Here, 'a' is converted to float from integer. Hence, it is correct.

Content Free Grammar for Generating Numbers

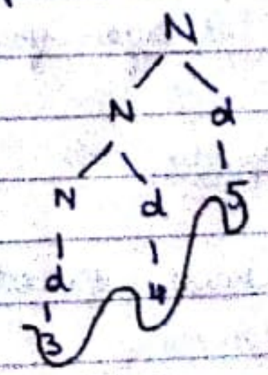
digit → 0|1|.....|9
Number → digit | number digit

Eg: 34



→ 34

Eg: 345



→ 345

* - Output of Semantic Analysis Phase is Annotated Parse Tree
- Types of errors such as mismatch errors, out of bounds errors are identified in semantic analysis.

Intermediate Code Generation

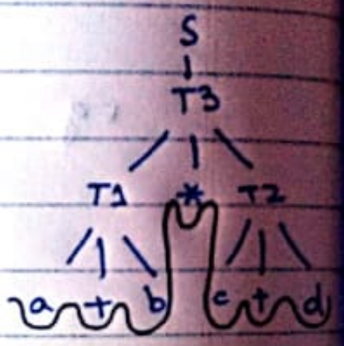
- In between analysis and synthesis phase or frontend and backend.
- 3 address code or p code
  - ↳ The maximum number of addresses in instruction is 3
- Eg. S = a + b * c + d

  T1 = a + b

  T2 = c + d

  T3 = T1 * T2 → 3 address code

  S = T3

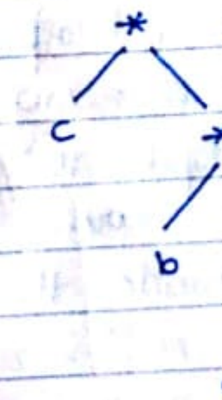


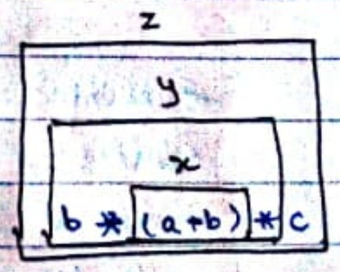
- Three address code is converted to code generator and it strictly goes to target machine.
  Eg. 8085 microprocessor
- Advantage of intermediate code generation is platform independence. i.e., the intermediate code generation has or can be gone to any machine like converting 8085 to 8086, etc; and it again goes to optimization.

Optimization

- Optimization or code optimization is reducing the instructions.
- Optimization, according to machine architecture depends on the target machine.

Eg. b * (a+b) * c





P.T.O →

12-12-'18

Programs Related To Compilers

- Interpreter
- Assembler
- Linker
- Loader
- Preprocessors
- Editors
- Debuggers
- Profilers
- Project Manager

Many other programs work along with the compiler to identify the errors and execute the program

Interpreter

- Same as compiler; but checks for syntax errors line by line, unlike the entire program like compiler.

- Interpreter is advantages for frequent modification and compilation.
- Java and Python use both compiler and interpreter.
- BASIC and LISP are interpreter level languages.

Assembler

- Compiler converts a high level language to assembly language, and the assembler converts assembly level language to low level language.

Linker

- Includes standard library functions in our program.
- If our program is divided into 3 parts, it combines or links them.
- Macro Substitutions.

Loader

- Data is stored in secondary memory, but to execute our program, it must be stored in main memory.
- Loader gives relocatable address from secondary memory to main memory, to execute the program.
- Mapping of main memory to cache memory.

Preprocessors

- Deletion of comments.
- Macro Substitutions.
- Giving line numbers for program in advanced compilers.

Editors

- Most important to execute program.
- Structured Editors.
- IDE.

Debuggers

- No compilation errors, but expected output not produced.
- Debuggers identify such run time errors, by tracing values according to line numbers or break points.

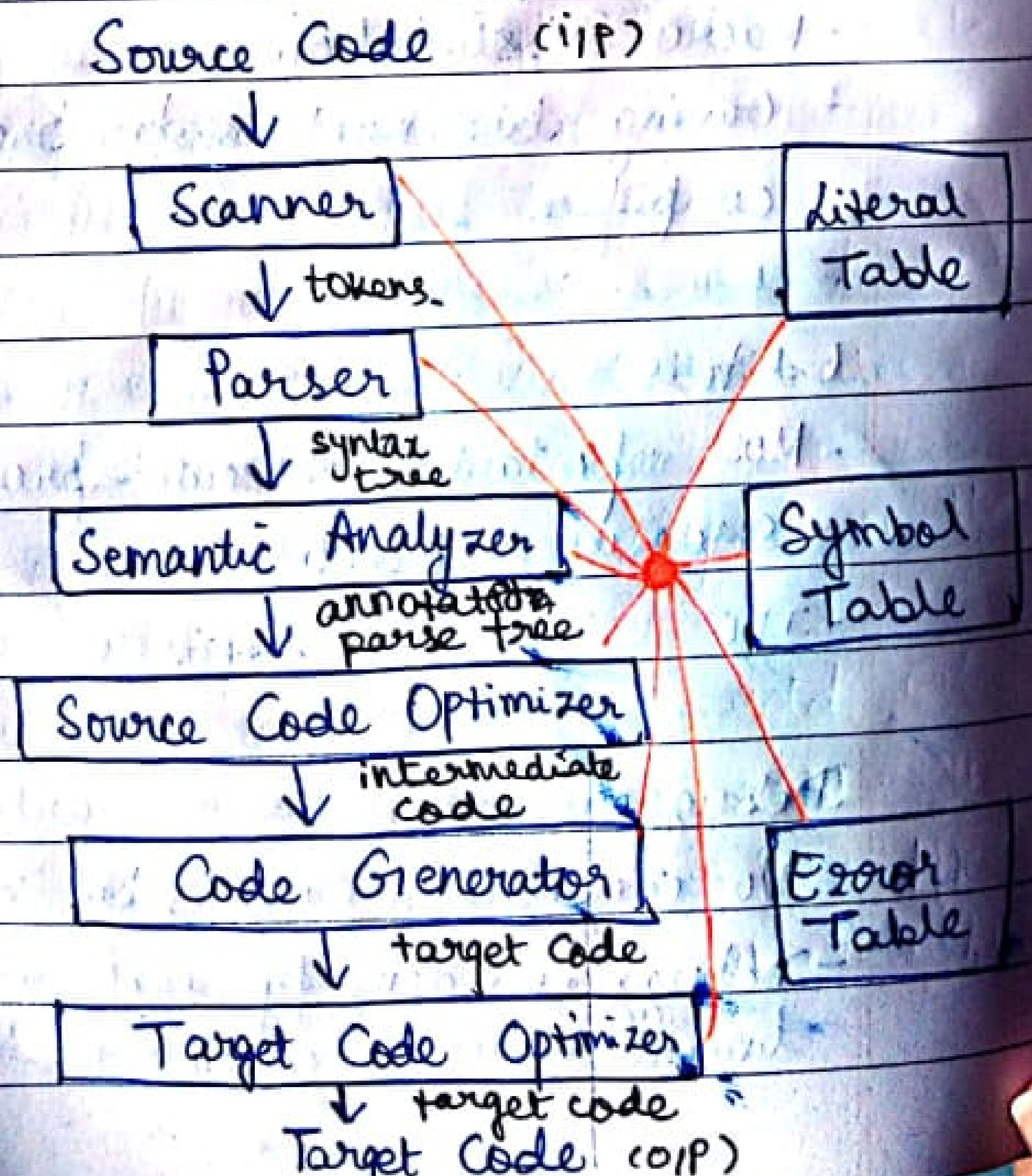
Profilers

- Used to know the statistics of the program.
- Eg. To know how much time is spent on a particular loop or how many times that loop is executed.
- Used to improve time complexity.

Project Managers

- There are in built project managers in LINUX to monitor the work of the compilers. They are:
  i) SCCS - Source Code Control System
  ii) RCS - Revision Control System

➡ Translation Process



- Auxillary codes used by some/all the phases of compiler are: literal table, symbol table, error table.

Eg. a[index] = 4 + 2

- Scanner Phase
  - Scans every expression and divides every token using lexical analyzer, scans identifiers/symbols.

a	identifier
[	left bracket
index	identifier
]	right bracket
=	~~arithematic~~ assignment operator
4	number
+	plus sign / arithematic operator
2	number

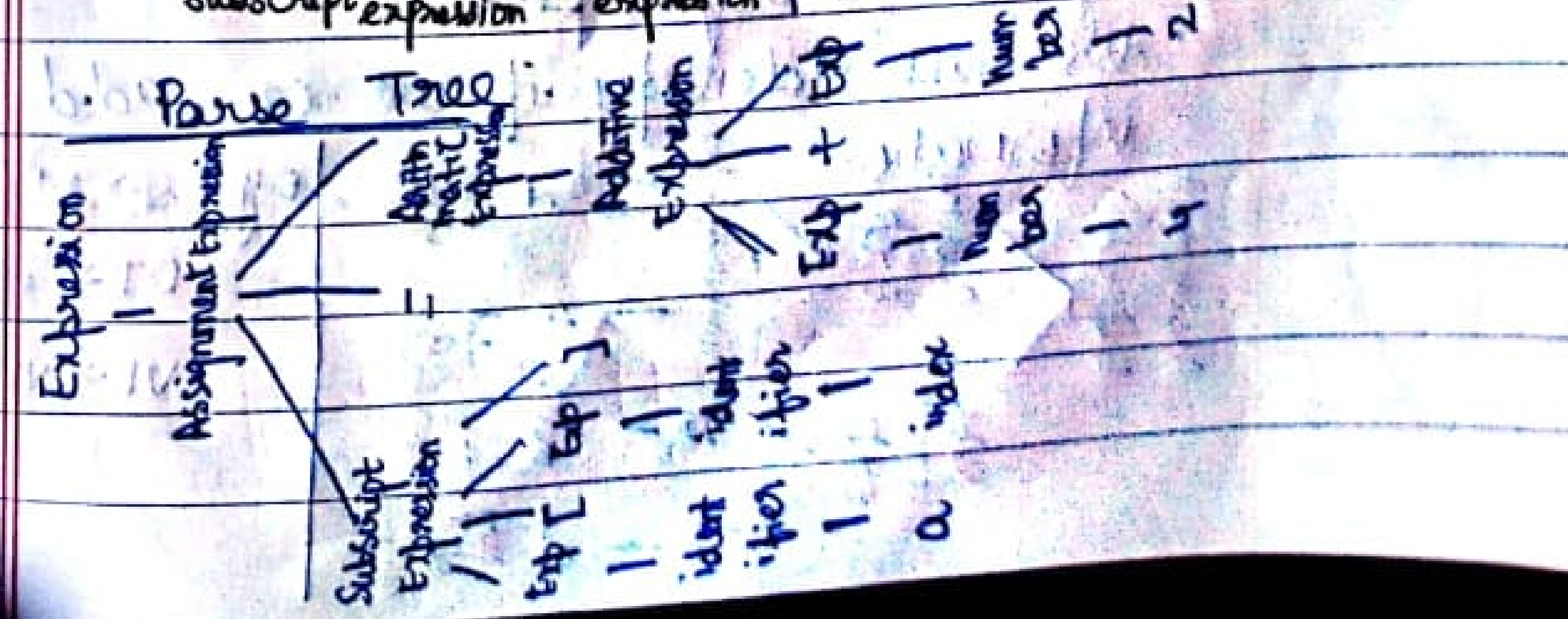
  - Also gives line numbers if not given by lexical analyzer.

- ~~Parser Phase~~
  - Arranges the tokens (O/P of scanner phase) to form a syntax tree

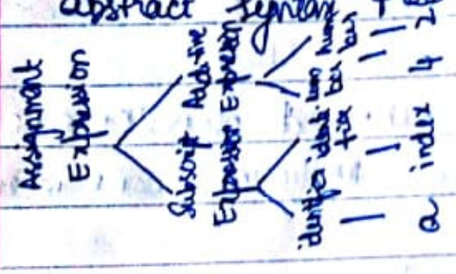
a[index] = 4 + 2 — a[index]: subscript expression; 4 + 2: arithematic expression

arr[i] → Subscript expression
a+b → assignment expression

- Parse Tree:

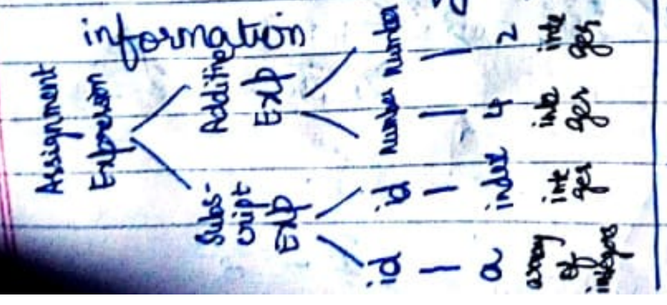


- We know that a subscript expression implies left paranthesis and right paranthesis and an additive expression implies +.
- So, we can reduce our syntax tree to abstract syntax tree.



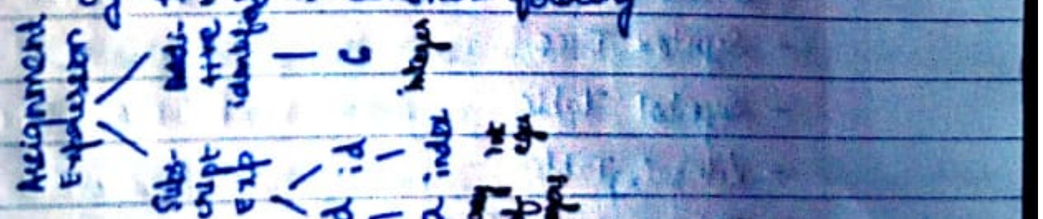
- Parser Phase
  - Verifies syntax of our program.

- Semantic Analyzer
  - Static Semantics
    - → Possible at compile time.
    - → Eg. Type declarations, Type Matching, etc.
  - Dynamic Semantics
    - → Not possible at compile time.
    - → Eg. Runtime Memory Allocation, Out of Array Index
  - It adds extra information (attributes) for context free grammar.
  - Meaning of a sentence is analyzed in the semantic analyzer phase.
  - We must identify if we can add extra information



Eg. S → DT VL;
DT → int/float
VL → VL id/id;

- Source Code Optimizer
  - Checks if there is any redundant code.
  - Eg. 4+2 = 6 → Constant folding



- Code Generator
  - Generates code by using assembly level language (8085/8086).

- Target Code Optimizer
  - Is there any optimization possible after generating code?
  - Use faster instructions instead of slow instructions
    - Eg. Use direct addressing mode instead of indirect addressing mode.
  - Use registers instead of main memory.

Code Generator Phase Code:

```
MOV  B0, index
MUL  R0, 2
MOV  R1, B0
ADD  R1, R0
MOV  R1, 6
```

Optimized code →

```
MOV  R0, index
SUL  R0
MOV  0a[R0], 6
```

→

13-12-'18

Major Data Structures in a Compiler

- Tokens
- Syntax Tree
- Symbol Table
- Literal Table
- Intermediate Code
- Temporary Files

Tokens

- Identifier / Keyword / relational operator / arithmetic operator
- Can be represented as enumerated data type in C.
- Global variables can be used. A global variable is a string. In C, array of characters.

Syntax Tree

- Tokens are an input for the syntax analysis phase, whose output is a parse tree, stored in the form of a tree.
- Syntax tree is a pointer based structure, and the entire tree is pointed by a single variable called as root. The intermediate nodes in the syntax tree are some variables.
- Syntax tree is a tree structure which is a pointer based structure, where each field has some data.

Symbol Table

- Whatever token is identified, it is stored in the symbol table.
- It may be a normal identifier/structure identifier/function identifier.
- Generally, all phases of the compiler use symbol table.
- In some phases, information is entered into symbol table, and in some phases, information is retrieved from symbol table.
- Give information about all symbols.
- Best data structure for frequent lookups/insertions/deletion is hash data structure.
- Symbol Table uses hash table, various tree structures. Several tables are used and maintained in a list or stack.
- Hash table is best used to implement symbol table.

Literal Table

- Used to store constants.
- Modification not possible.
- Only insertion and searching is possible

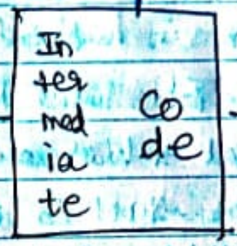
Intermediate Code

- Array of strings or temporary files ~~may be~~ or linked list of structures are the different data structures used to store intermediate code

Temporary Files
- Required to run any program.
- Since entire program cannot be stored in memory, we store the program in temporary files and bring them to the main memory during execution.

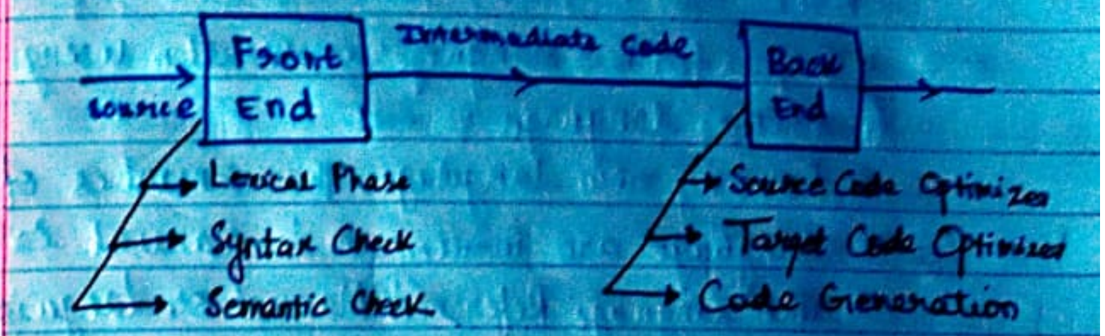
Other Issues In Compiler Design

- Compiler design is divided into analysis phase and synthesis phase.
- Analysis Phase
  → Lexical
  → Syntax
  → Semantic



- Synthesis Phase
  → Optimization
  → Code Generation
- Intermediate code is a part of both the analysis phase and the synthesis phase.
- Analysis Phase
  • Mathematic Techniques are Used.
  • Code is Analyzed
  • Intermediate Code is generated.
- Synthesis Phase
  • Specialized Techniques are Used.
  • Takes input as intermediate code.
- According to the methods adapted for translation, analysis and synthesis phases are generated.

- Front End and Back End



- Portability of language is possible using intermediate code.

- Phase and Pass
  • Phase is used to complete a specific task. For each and every phase, the pass is specific task.
  • If there are more optimizations, there are more passes in the compiler.
  • One Pass Compiler: All phases in a single pass, lexical to code generation phase in a single pass.
  • Two Pass Compiler: Lexical, Syntax and Semantic Phases in 1st pass and the remaining phases in the 2nd pass.
  • 3 pass, 4 pass, ..., multiple pass compilers.
  • If we want immediate result, we use one pass compiler.
  • One pass compilers are generally not used.
  • Intermediate languages like Pascal and C use one pass compiler and module 2 is 2 pass compiler and lisp and python use more passes.

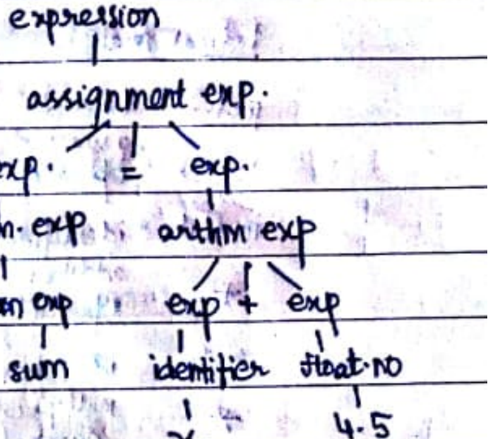
Language Definition and Compilers

- Compiler for new language is written using already existing language.
- For any new language, the semantics of the language are written in a manual called as the Language Reference Manual or Language Definition
- It consists of all the semantic rules and reference rules.
- Existing language used to write compiler must be created using an existing compiler that must be standardized.
- ANSI and ISO are standardizations:
  ANSI: American National Standardization Institution
  ISO: International Organization For Standardization
- Most of the compilers are written in C language.
- Semantics of the language can be given using mathematical models in the Reference Manual
- Depending on whether our program supports memory dynamically or not, we must design our compiler considering run time environment.
  - Fortan 77: no pointers, no dynamic allocation, no recursive function calls.
  - Thus, no dynamic memory used and entire memory allocated during function call.
- Pascal, C, Algol: Limited form of dynamic allocation and recursive function calls, requires semidynamic or stackbase runtime environment and heaps.
- LISP and smalltalk: Functional and Most Object Oriented Language, requires fully dynamic environment, compiler must be designed in such a way that garbage collection is automatically done when the program executes.

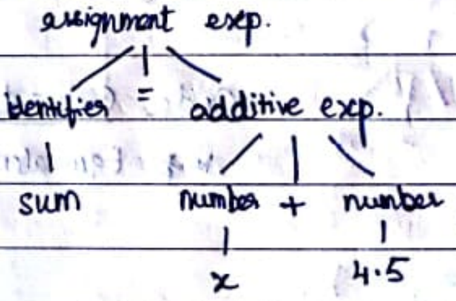
→ Exercise: Write the different phases for the given expression assuming that 'x' is an integer. sum = x + 4.5

→ Scanner Phase

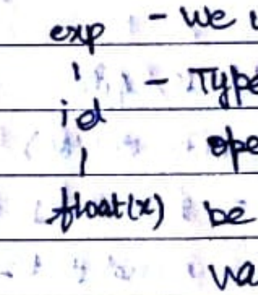
sum	identifier
=	assignment operator
x	identifier
+	plus sign/ arithmetic operator
4.5	floating point number



→ Parser Phase



→ Annotated Parse Tree



- We use type conversion
- Type conversion is an operator. It must be stored in a temporary variable. y = float(x)
  sum = float(x) + 4.5

→

14-12-'18

→ Write the translation process for the given statement. Assume that the rate value is a floating point.

→ position = initial + rate * 60

E → E+E

E → E*E

E → E=E

E → id

E → NO

t1 = int to real(60)

Lexical Analysis

position	identifier
=	assignment operator
initial	identifier
+	addition operator
rate	identifier
*	multiplication symbol
60	number

Expression

Assignment Expression

Expression = Expression

Identifier | Arithematic Expression

position (id1) | + | identifier * identifier number

initial (id1) | rate (id3) | int to real | 60

3-address code

t1 = int to real(60)

t2 = rate * t1

t3 = initial + t2

position = t3

Code Generation

* Equivalent 8085/8086 code

* Source Code Optimization

→ Write 60 as 60.0

→ No need to write int to real

→ Write the translation process for the given expression, assuming that 'i' and 'j' are float. i = j * 70 + j + 2

Lexical Analysis

i	identifier
=	assignment operator
*	multiplication operator
70	number
+	addition operator
j	identifier
2	Number

Syntax Tree

Expression

Assignment Expression

Expression = Expression

Identifier | + | id1 | * | +

identifier | int to real | identifier | int to real

id1 | 70 | j | 2

→ What is language generated by the given grammar?

$E \to E+E \mid E*E \mid (E) \mid a \mid b$

→ Expressions containing 'a' and 'b' are generated.

$((a*b)+(a+b))$ → Draw Leftmost and Rightmost Derivation

Leftmost:

$E \overset{lm}{\Rightarrow} (E) \overset{lm}{\Rightarrow} (E+E) \overset{lm}{\Rightarrow} ((E)+E) \overset{lm}{\Rightarrow} ((E*E)+E)$

$\overset{lm}{\Rightarrow} ((a*E)+E) \overset{lm}{\Rightarrow} ((a*b)+E) \overset{lm}{\Rightarrow} ((a*b)+(E))$

$\overset{lm}{\Rightarrow} ((a*b)+(E+E)) \overset{lm}{\Rightarrow} ((a*b)+(a+E)) \overset{lm}{\Rightarrow} ((a*b)+(a+b))$

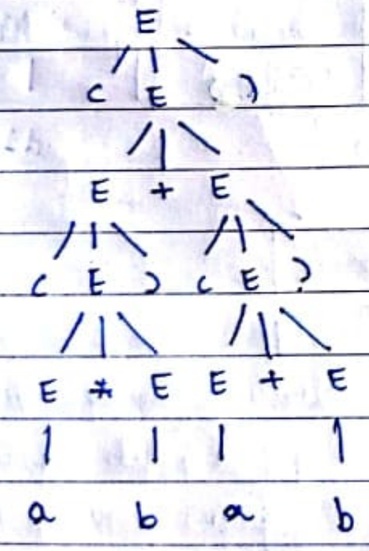
Rightmost:

$E \overset{rm}{\Rightarrow} (E) \overset{rm}{\Rightarrow} (E+E) \overset{rm}{\Rightarrow} (E+(E)) \overset{rm}{\Rightarrow} (E+(E+E))$

$\overset{rm}{\Rightarrow} (E+(E+b)) \overset{rm}{\Rightarrow} (E+(a+b)) \overset{rm}{\Rightarrow} ((E)+(a+b))$

$\overset{rm}{\Rightarrow} ((E*E)+(a+b)) \overset{rm}{\Rightarrow} ((E*b)+(a+b)) \overset{rm}{\Rightarrow} ((a*b)+(a+b))$

Tree:



$\Rightarrow ((a*b)+(a+b))$

P.T.O. →

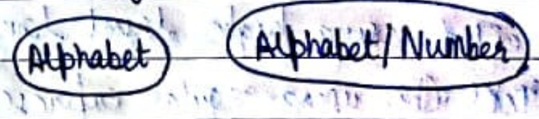
→ Write the regular expression and transition diagram for number, floating point number and identifier

→ i) Number / Digit

ii) Floating Point Number

$\Rightarrow (0....9)^+ . (0....9)^+$

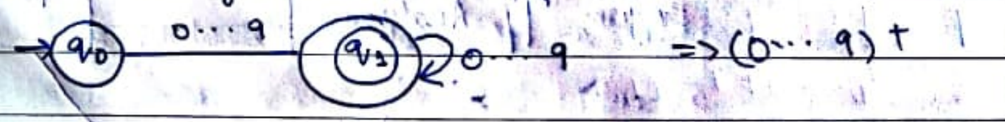
iii) Identifier



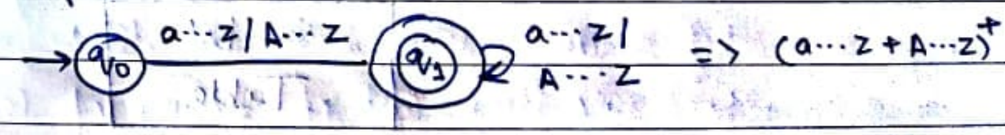
(alphabet) (alphabet + Number)*

$\Rightarrow ((a \cdots z)+(A \cdots z))((a \cdots z)+(A \cdots z)+(0 \cdots 9))^*$

i) Number / Digit



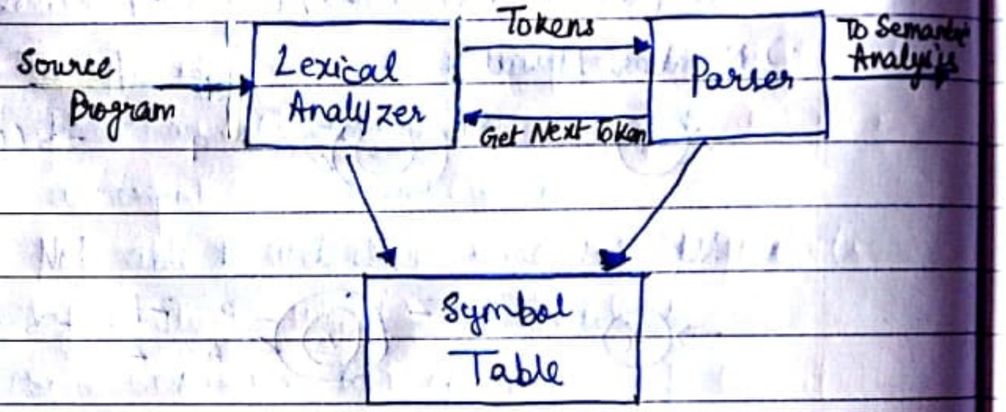
ii) Alphabet



19-12-'18

Lexical Analysis

- First phase in compiler.
- It is divided into two parts:
  i) Scanning - Read the entire program and delete the comment lines, give valid number of spaces (i.e., delete the extra spaces (int a and int a)), include line numbers, etc.
  ii) Lexical Analysis
- It is implemented using higher level languages or by using Lex Tool (by using regular expressions internally).
- The role of Lexical Analyzer



* - Why to separate Lexical Analysis and parsing?
  i) Simplicity of Design
  ii) Improving Compiler Efficiency
  iii) Enhancing Compiler Profitability
  → If Lexical Analysis is separated, we can use specialized buffering techniques (to reduce comm time from memory to processor and vice versa)
  → In Lexical Analysis phase, it is interactive with the input, the input is stored.

* - Tokens, Patterns and Lexemes
  → A token is a pair, a token name and an optional token value.
  → A pattern is a description of the form that the lexemes of a token may take.
  → A lexeme is a sequence of characters in the source program that matches the pattern for a token.
  • Token name may be identifier, delimiter, arithmetic operator, relational operator, keyword, etc.
  • A token must be described using a regular expression.
  • Lexemes Examples: abc, i2, rr2, r123, sum, $, =, +, etc.
  • Lexical Analysis Phase will give the descriptions of all kinds of tokens
  • If the given error input is not identified exactly by the compiler, i.e., not identified by the existing regular expression, it is identified in the lexical analysis phase. Eg: 1i

  • Example:

Token	Informal Description	Sample Lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by integer and digits	pi, score, D2
number	Any numeric Constant	
literal	Anything but surrounded by " "	"core dumped"

  • Main role of lexical analyzer is to identify the tokens

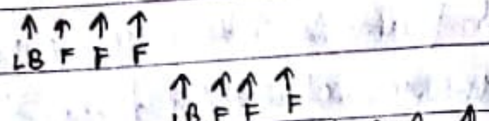
- Attributes for tokens
  - In a program, various lenemes are matched or categorized under one ~~is~~ token.
  - Eg: sum, D2, pi = identifiers; 12, 3.14, 0.2932 = numbers
  - Eg: E = M * C ** 2
    - → <id, pointer to a symbol table for entry E>
    - → <assign-op>
    - → <id, pointer to symbol table for entry for M>
    - → <mult-op>
    - → <id, pointer to symbol table entry for E>
    - → <exp-op>
    - → <Number, integer value 2>

- Lexical Errors
  - The types of errors or identified in lexical analyzer phase are: not able to identify patterns given as input.
  - Not able to understand errors in keywords.
    Eg: writing while() as hwile()
  - Some errors are out of power of lexical analyzer to recognize: Eg. fi (a == f(x))
    - ↳ It treads fi as function identifier
  - However, it may be able to recognize errors like d = 2r
  - Such errors are recognized when no patterns for tokens matches a character sequence.

- Error Recovery
  - In Lexical Analysis Phase, all the errors recovered are related to single character errors.
  - Panic Mode: Successive characters are ignored until we reach to a well formed token, i.e., panic mode recovery means discard all characters until a valid sequence of characters are found; i.e., a well formed token is found.
  - Delete one character from the remaining input.
  - Insert a missing character into the remaining input.
  - Replace a character by another character.
  - Transpose two adjacent characters.
  - Eg. int 222; → treated as no in lexical analysis
    int ##??!(abc);
    discarded => panic mode recovery

- Input Buffering
  - Sometimes Lexical Analyzer needs to look ahead some symbols to decide about the token to return
    - → In C language, we need to look after -, = or < to decide what token to return
    - → In Fortan: DO5I = 1.25
  - We need to introduce a two buffer scheme to handle large look-aheads safely.
  - The main disadvantage with one buffer technique is that we reach end of ~~file~~ buffer in first buffer, even before so, character reading takes time.
  - If we use two buffer technique, we simultaneously load second buffer while reading from first buffer.
  - In this technique, we must check if it is end of buffer or end of file. If it is end of buffer, reload the buffer to read further.
  - Eg: | | E | = | M | * | C | * | * | 2 | eof | | | |

• To avoid this, we use sentinels.
• We use two pointers, lexeme begin and forward.
• Between lexeme begin and forward lies our token.
• Eg: sum = x y z + a b c



• Sentinels

E = M eof * C * * 2 eof : eof

→ Program:

```
switch (*forward++)
{
  case eof: if (forward is at end of first buffer)
            {
              reload second buffer;
              forward = beginning of second buffer;
            }
            else if (forward is at end of second buffer)
            {
              reload first buffer;
              forward = beginning of first buffer;
            }
            else
            /* eof within a buffer marks the end
               of input */
            terminate lexical analysis;
            break;
}
```

* — Specification of Tokens

• The main role of Lexical Analysis is to identify tokens
• For this, we have to give the description of tokens called specification of tokens, done by using regular expressions
• In theory of compilation, regular expressions are used to formalize the specification of tokens.
• Regular Expressions are means for specifying regular expressions.

* — Regular Expressions

• ε is a Regular Expression, L(ε) = {ε}
• If 'a' is a symbol in Σ, then 'a' is a regular expression L(a) = a
• (r)|(s) is a regular expression denoting the language L(r) ∪ L(s)
• (r)(s) is a regular expression denoting the language L(r)L(s)
• (r)* is a regular expression denoting (L(r))*
• (r) is a regular expression denoting L(r)

20-12-18

→ Phases in a Compiler:

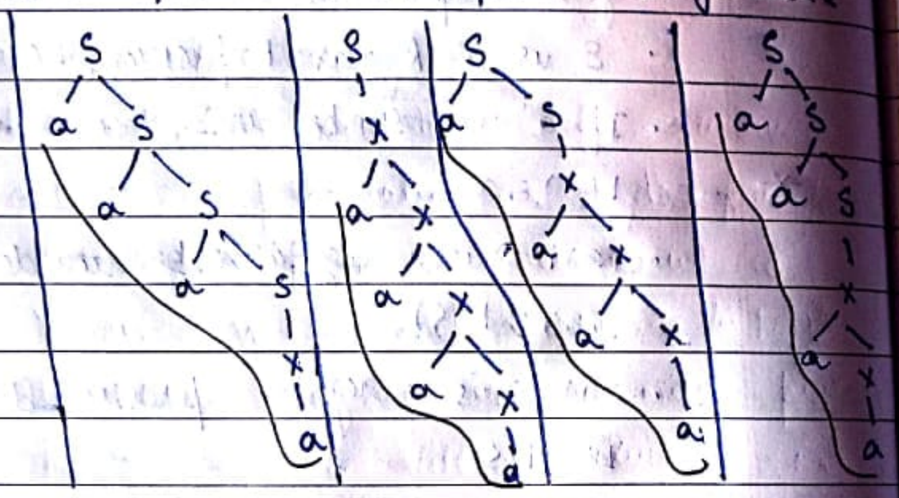
i) Lexical Analysis
ii) Syntax Analysis
iii) Semantic Analysis
iv) Source Code Optimization
v) Code Generation
vi) Target Code Optimization

Context Free Grammar - CFG

- Specification for defining the structure of a language.
- Unambiguous Grammar: Unique tree for every derivation
- Ambiguous Grammar: For atleast one string (in the grammar) we can construct more than one parse tree
- Problems:

(i) Verify if the given grammar is ambiguous or not for the string aaaa. S→aS/x, X→aX/a

→ S → aS/x X → aX/a string = aaaa



=> Thus, the grammar is ambiguous.

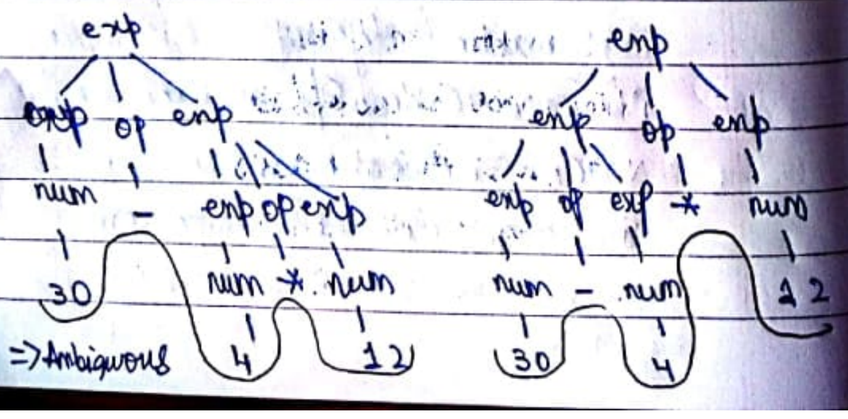
(ii) Derive the expression w = number - number * number.

i.e., w = 30-4*12 from the grammar:

exp → exp op exp | number op → +/-/*

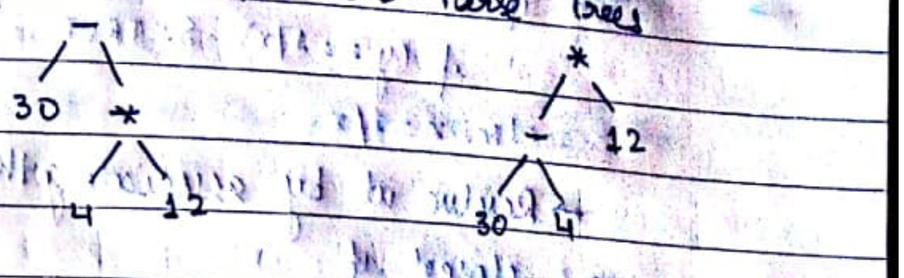
→ exp → exp op exp | number op → +/-/*

w = num - num * num, 30 - 4 * 12



=> Ambiguous

=> Syntax Trees for above Parse Trees



(iii) Derive the string : w = if (0) if (1) other else other with the given grammar. Check if ambiguous.

→ stmt → ifstmt / other

ifStmt → if (exp) stmt / if (exp) stmt else stmt

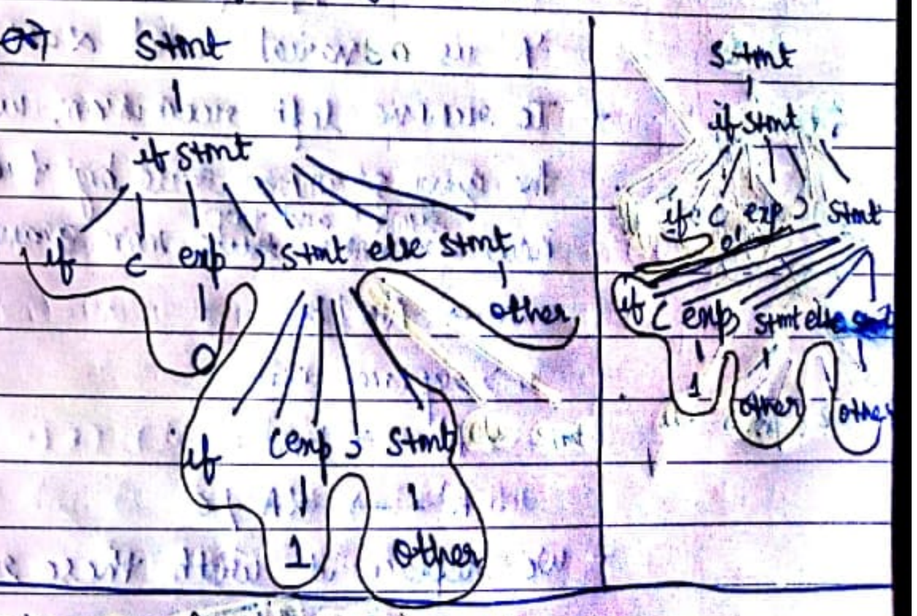
exp → 0/1

Variables: stmt, ifStmt, exp

Terminals: if, (,), else, 0, 1

~~w = if (exp) stmt / if (exp) stmt else stmt~~

w = if (0) if (1) other else other



- Ambiguous Grammar
- Each else is associated with a different if in both the parse trees, so dangling else problem arises
- So, we have to solve it explicitly by associating the else with the nearest if.

Eg: if (x != 0)
    if (y == 1/x) S = TRUE
  else z = 1/x

→ Resolve it by associating the else with the closest if.

→ Or, give curly braces to specify blocks

→ Left Recursion

- Rewriting grammar rules to give us unambiguous grammar.
- A grammar is said to be left recursive if atleast one production is in the form of: $A \longrightarrow A\alpha / \beta$ for Grammar G where 'A' is a variable and α and β are V or T or VUT
- To remove left recursion, we must replace the given grammar rule by taking one more symbol and two new ~~grammar rule~~ more grammar rules

  $A \rightarrow A\alpha / \beta$

  $\Rightarrow$ replace with

  $A' \longrightarrow \beta A'$

  $A' \longrightarrow \alpha A' / \epsilon$

- We replace it with these rules, we encounter right recursion.
- But we have right recursion. So the string ends at some or the other point

21-12-'18

→ Eg: Content Free Grammar for Data Type Declaration

S → DT VL;

DT → int / float / char;

VL → VL, id / id;

→ Eg: Content Free Grammar for 'expression'

~~S → while (expr) { S }~~

exp → exp * exp

exp → exp + exp

exp → exp = exp

exp → (exp)

exp → id

exp → number

Tokens		Regular Expressions
+	assignment op.	+
*	multiplication op.	*
=	assignment op.	=
(	left parenthesis	(
)	right parenthesis	)
id	identifier	(letter (letter + digit)*
number	floating point number	(0—9)+ . (0—9)+

letter → A|B|....|Z|a|b|....|z

digit → 0|1|....|9

id → letter_ (letter|digit)*

P.T.O. →

Lexical Analysis (continued)

→ Regular Definitions

- Extension
- Recognition of Tokens
  - Starting point is the language grammar to understand the tokens.

Eg. stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| number

For the give CFG, tokens are:
if, then, else, ~~ten~~ relop, id, number

- The next step is to formalize the pattern

digit → [0-9]

Digits → digit+

number → digit (. digit)? (E [+-]? Digit

letter → [a-z A-Z _]

id → letter (letter | digit)*

If → if

Then → then

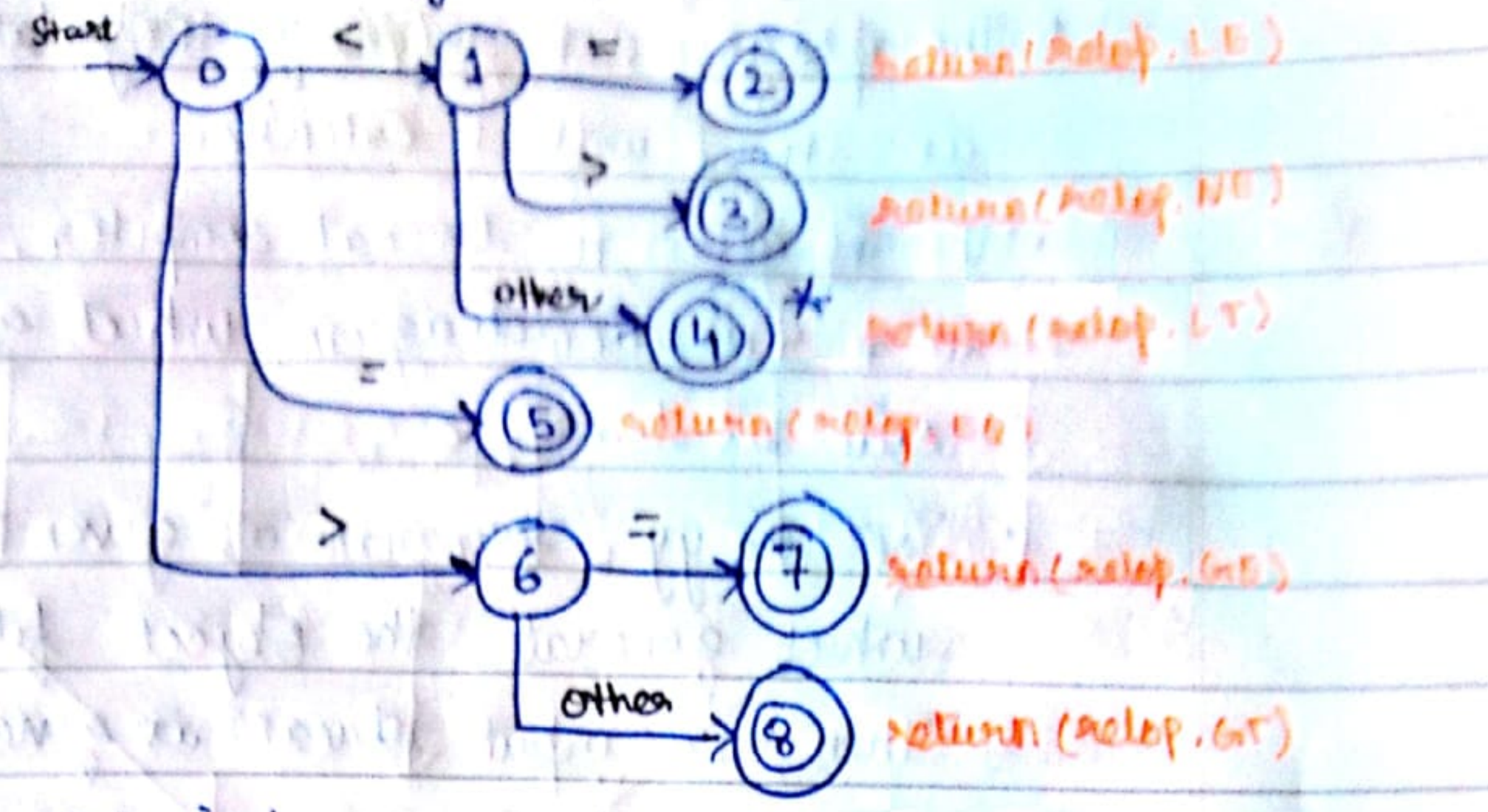
Else → else

Relop → < | > | <= | >= | = | <>

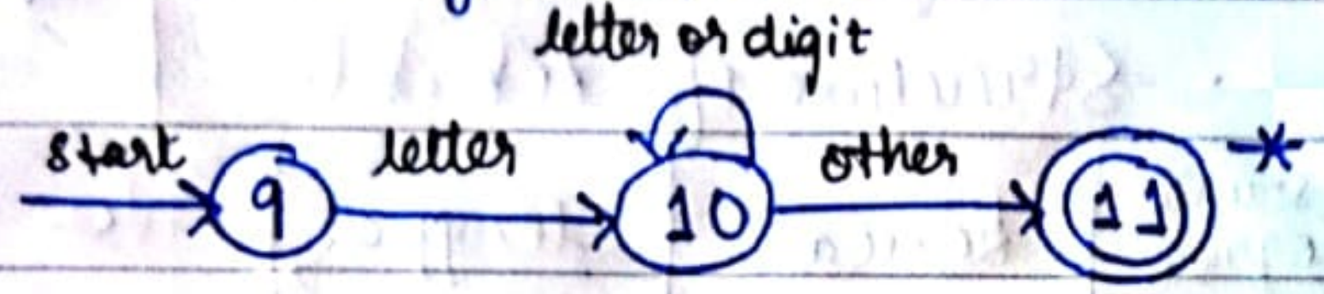
We also handle whitespaces; ws → (blank

- Transition Diagrams

• Transition diagram for the relop

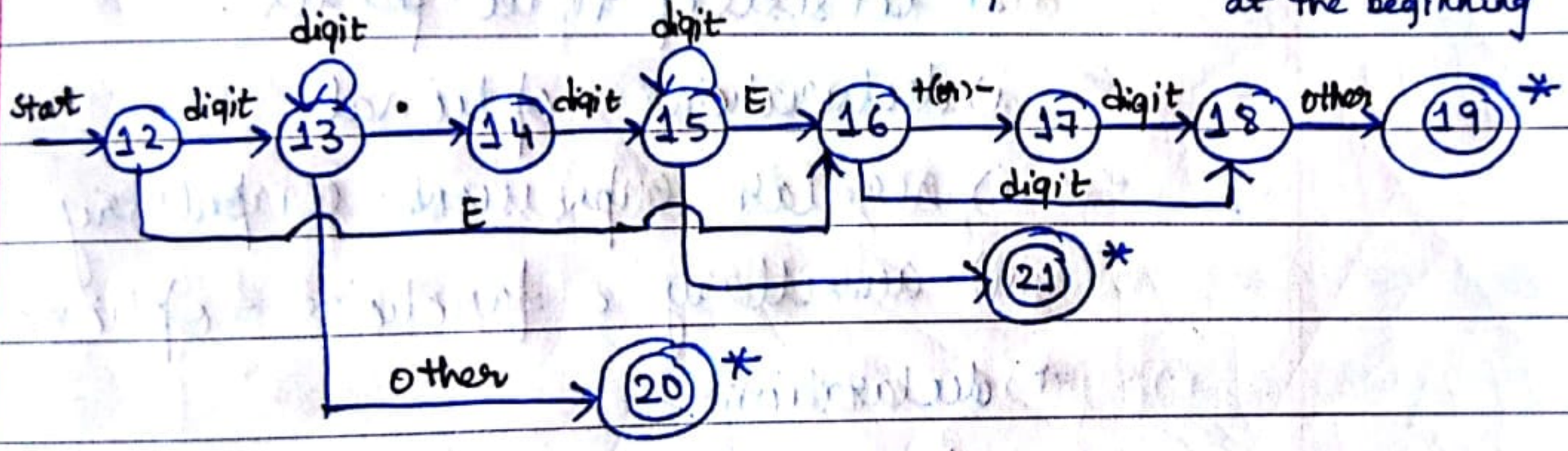


• Transition diagram for reserved words and identifiers

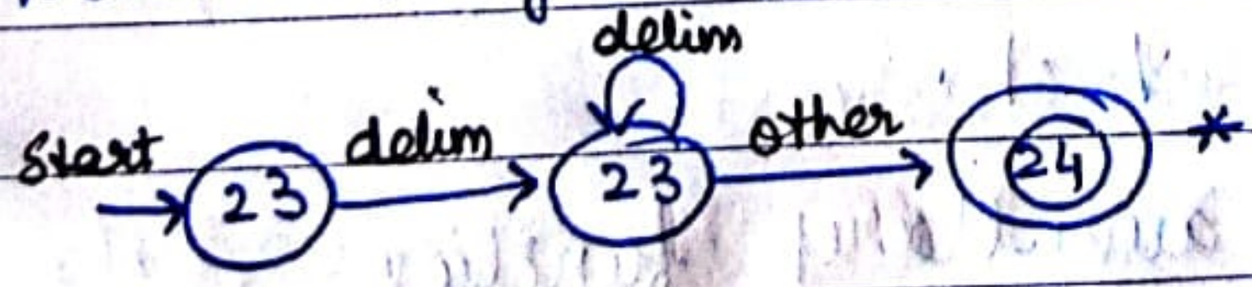


• Transition diagram for unsigned numbers

→ For signed nos, add a + or - char at the beginning



• Transition diagram for whitespace

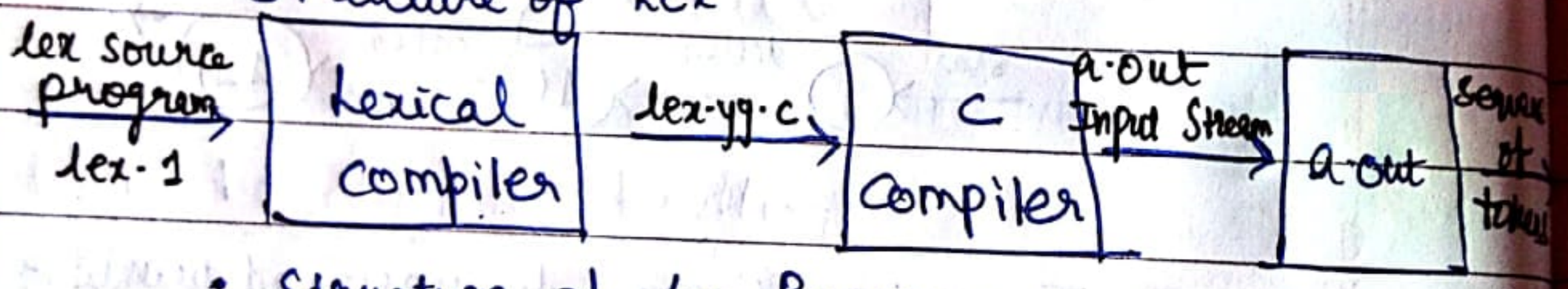


- Architecture of a Transition Diagram based Lexical Analyzer

• We convert transition diagrams to a program.

• Program for Relational Operator, Identifier, Unsigned Number, Whitespace.

- Lexical Analyzer Generator - Lex
  - The lex program has .l extension.
  - The programs to identify regular expressions is stored with .l extension.
  - It is given to lexical compiler, which converts it to transition diagram, which gives output code as lex.yy.c
  - The lex.yy.c program is given to the C compiler which generates the object file .obj.
  - Then, the input stream is given to ./a.out to generate a sequence of tokens.
  - Structure of Lex



  - Structure of Lex Programs
    It consists of three parts:
    i) declarations - optional
    ii) regular expressions - compulsory
    iii) auxillary C functions - optional
    → declarations
    %.%.
    translation rules ⟶ pattern {Action}
    %.%.
    auxillary functions

Syntax Analysis

- Programming language constructs can be represented in CFG form or BNF form.
- CFG

$G = (V, T, P, S)$; $V = S, DT, VL$; $T = int, float, ,, ;, id$; $S = S$

$S \rightarrow DT\ VL;$

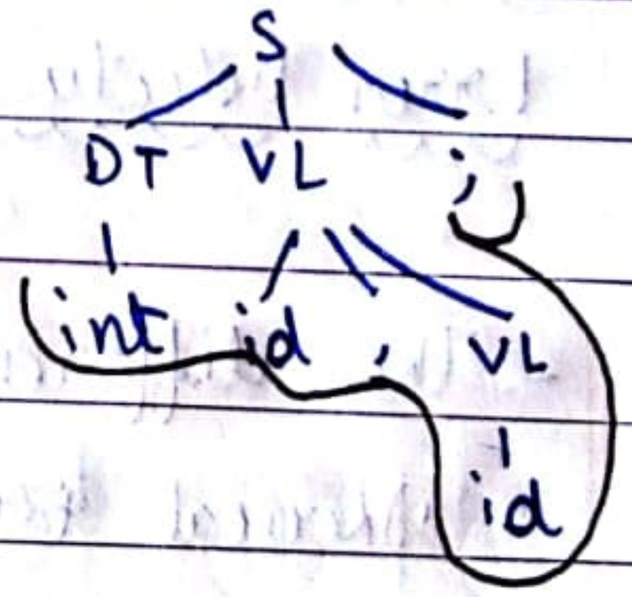
$DT \rightarrow int / float;$ } P

$VL \rightarrow id / id, VL;$ }

- Syntax Analysis has two types of algorithms:
  i) Top-Down Parsing
  ii) Bottom-Up Parsing
- Eg. int id, id; is valid or not?

→ Top-Down Parsing:

Start with Start symbol and check if we can reach/derive string/sentence



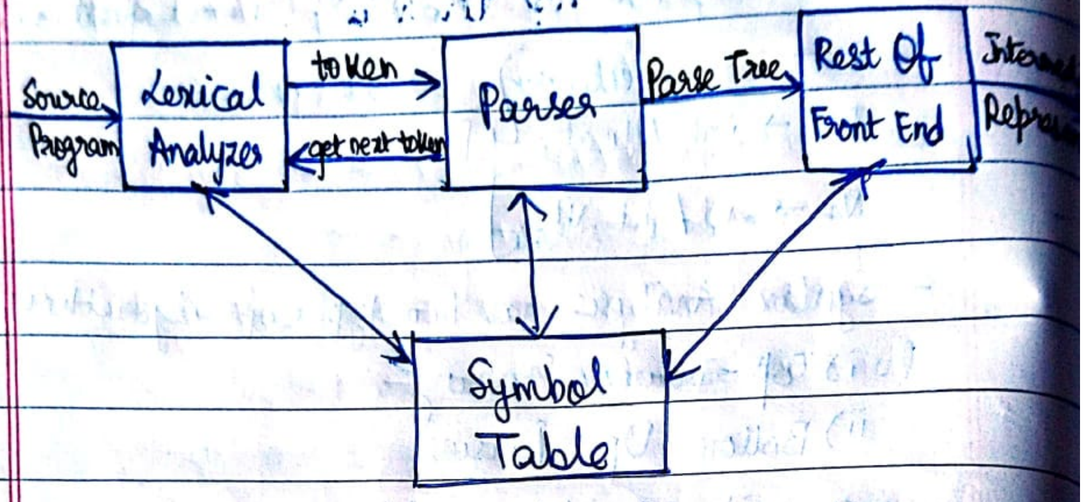
→ Bottom-Up Parsing:

Start with string and check if we can reach start symbol.

- Two algorithms can be used in top-down parsing:
  i) Predictive Parsing
  ii) Recursive Present Parser
- Four algorithms can be used in bottom-up parsing:
  i) Shift-Reduce Parser
  ii) SLR Parser (Simple LR Parser)
  iii) CLR Parser (Canonical LR Parser)
  iv) LALR Parser
  ↳ mostly used in compilers

- Parser Generator
  - It is used to generate parsers automatically.
  - Generated by using YACC tool.

The Role Of Parser



Error Detection And Handling

- The different types of errors are:

i) Lexical Errors
- Identified in lexical analysis phase.
(Eg. Not able to identify exact token, eg. if)

ii) Syntax Errors
- An error in the syntax.
(Eg. Missing ; and if ( ), etc.)

iii) Semantic Errors
- Eg. Type mismatch, type declaration

iv) Logical Errors
- Very difficult to find, must analyze syntax errors but expected output is
(Eg. i=j for i==j)

Context Free Grammar

- Left Recursive Grammar
  - First symbol in LHS and RHS is same, then this grammar is called left recursive grammar.
- Precondition for top down approach is that the grammar must not be left recursive.

Error Recovery Strategies

- Panic Mode Recovery
  - Discard input symbol one at a time until one of designated set of tokens is found.
  (Eg id ++ id ++ id (OR) id + / id + / id (or) id // id)
- Phrase Level Recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue.
- Error Productions
  - Augment the grammar with productions that generate the erroneous constructs.
- Global Correction
  - Choosing minimal sequence of changes to obtain a globally least cost corrections.
  - Only theoretical concept, it is not used practically.

Derivations

- Productions are treated as rewriting rules to generate a string.
- Rightmost and leftmost generations:

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

- The string is $-(id+id)$
- Leftmost: $E \to -E \to -(E)+E \to -(E+E)$

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

- Rightmost: $E \to -E \to -(E) \to -(E+\overset{E}{id})$

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

Elimination of Left Recursion - Immediate Left Recursion

- A grammar is left recursive if it has a non terminal A such that there is a derivation: $A^{+} \Rightarrow A\alpha$
- Top down parsing methods can't handle left recursive grammars
- A simple rule for direct left recursion elimination:
- For a rule like $A \to A\alpha \mid \beta$ we may replace it with

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

- Note: $A^{+} \Rightarrow A\alpha \Rightarrow A \to A\alpha \Rightarrow A \to AA\alpha \Rightarrow A \to AAA\alpha \Rightarrow \dots \Rightarrow A^{+}$
- Eg. $E \longrightarrow E+T \mid T$

$A \longrightarrow A\alpha \mid \beta$

Here: $A = E$ $\quad A \to \beta A' \Rightarrow E \to TE'$

$\alpha = +T$ $\quad A' \to \alpha A' / \epsilon \Rightarrow E' \to +TE' / \epsilon$

$\beta = T$ $\quad \hookrightarrow$ Eliminates Left Recursion

- Eg. $T \rightarrow T*F \mid F$

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

Here: $A = T$ $\quad A \rightarrow \beta A' \Rightarrow T \rightarrow FT'$

$\alpha = *F$ $\quad A' \rightarrow \alpha A' \mid \epsilon \Rightarrow T' \rightarrow *FT' \mid \epsilon$

$\beta = \#F$ $\quad$ ↳ Elimination of left Recursion

Left Factoring

- Left Factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down pars
- Consider the following grammar:
  - stmt → if expr then stmt else stmt | if expr then stmt
- On seeing input if it is not clear for the parser which one to use.
- We can easily perform left factoring:
  - If we have $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ then we replace it with:
    - → $A \rightarrow \alpha A'$
    - → $A' \rightarrow \beta_1 \mid \beta_2$
- For the above example:

  $S \rightarrow$ if expr then stmt S'

  $S' \rightarrow$ else stmt $\mid \epsilon$

Elimination of Left Recursion - Continuation

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

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

$A = L, \alpha = ,S, \beta = S$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

Grammar:

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

Eg. $E \to E+E/T$, $T \to TF/F$ $F \to F*/a$

→ $E \to E+E/T$ → $T \to TF/F$ → $F \to F*/a$

$E \to TE'$ $T \to FT'$ $F \to aF'$

$E' \to +TE'/\epsilon$ $T' \to FT'/\epsilon$ $F' \to *F'/\epsilon$

Grammar:

$E \to TE'$ $T \to FT'$ $F \to aF'$

$E' \to +TE'/\epsilon$ $T' \to FT'/\epsilon$ $F' \to *F'/\epsilon$

12-18

→ For any number of A-productions - Immediate Left Recursion

$$A \to A\alpha_1 / A\alpha_2 / \ldots\ldots / A\alpha_m / \beta_1 / \beta_2 / \ldots\ldots / \beta_n$$

where $\alpha \in (V \cup T)^*$

Then:

$$A \to \beta_1 A' / \beta_2 A' / \ldots\ldots / \beta_n A'$$

$$A' \to \alpha_1 A' / \alpha_2 A' / \ldots\ldots / \alpha_m A' / \epsilon$$

→ Indirect Left Recursion

$S \to Ab/d$ → Indirect Left Recursion

$A \to Aa/Sac/\epsilon$

$\Rightarrow$ $S \Rightarrow Ab \Rightarrow Sacb$

NOTE:

The above formulae for any number of A productions and 2 number of A productions are useful only for immediate left recursion.

Indirect Left Recursion cannot be eliminated using the above formulae.

→

Eg. $A \longrightarrow ABd / Aa / a / b$ $B \longrightarrow Be / b$

$\rightarrow A \longrightarrow ABd / Aa / a / b$ $B \longrightarrow Be / b$

$A \longrightarrow A\alpha_1 / A\alpha_2 / \beta_1 / \beta_2$ $A \longrightarrow A\alpha / \beta$

$\Rightarrow A \longrightarrow aA' / bA'$ $\Rightarrow B \longrightarrow \in / eB'$

$A' \longrightarrow BdA' / aA' / \epsilon$ $B' \longrightarrow eB' / \epsilon$

$\therefore$ Grammar: $A \longrightarrow aA' / bA'$ $B \longrightarrow eB'$

$A' \longrightarrow BdA' / aA' / \epsilon$ $B' \longrightarrow eB' / \epsilon$

Eg. $F \longrightarrow F* / a / b$

$A \longrightarrow A\alpha / \beta_1 / \beta_2$

$F \longrightarrow aF' / bF'$ $F' \longrightarrow *F' / \epsilon$

$\therefore$ Grammar: $F \longrightarrow aF' / bF'$ $F' \longrightarrow *F' / \epsilon$

Generalized Algorithm for Eliminating Left Recursion (both immediate and indirect)

Assume that the grammar has no cycles! $A \overset{+}{\Longrightarrow} A$

Algorithm:

i/p: Grammar G with no cycles.

o/p: Equivalent grammar G with no left recursion.

Method: (i) Arrange the non-terminals in some order $A_1, A_2, A_3, \ldots, A_n$

(ii) for (each i from 1 to n) {

for (each j from 1 to i-1) {

replace each production of the form $A_i \longrightarrow A_j\gamma$ by the productions $A_i \longrightarrow \delta_1\gamma / \delta_2\gamma / \ldots / \delta_k\gamma$ where $A_j \longrightarrow \delta_1 / \delta_2 / \ldots / \delta_k$ are all current A_j productions }

Eliminate the immediate left productions among the A_p productions }

→ Problem:

Assume CFG: $S \to Aa/b$, $A \to Ac/Sd/\epsilon$ and eliminate Left Recursion.

* Rename the productions and rewrite the grammar

$S \to A_1$

$A \to A_2$

$A_1 \to A_2a/b$

$A_2 \to A_2c/A_1d/\epsilon$

* Loop:

$i = 1$ j $A_1 \to A_2a/b$ (No immediate left recursion)

$i = 2$ $j = 1$

$A_1 \to A_2a/b \to$ NO ILR

$A_2 \to A_2c/A_1d/\epsilon$

$A_2 \to A_2c/A_2ad/bd/\epsilon$

⇓

$A_2 \to bdA_2'/A_2'$

$A_2' \to cA_2'/adA_2'/\epsilon$

Grammar:

$A_1 \to A_2a/b$

$A_2 \to bdA_2'/A_2'$

$A_2' \to cA_2'/adA_2'/\epsilon$

By substituting the values, we get:

$S \to Aa/b$

$A \to bdA'/A'$

$A \to cA'/adA'/\epsilon$

Left Factoring

- If the prefix of the productions are same, then there arises an ambiguity as to which production to choose.

- Left factoring for two options:

$$A \longrightarrow \alpha\beta_1 / \alpha\beta_2$$

$$\Rightarrow \quad A \rightarrow \alpha A'$$

$$A' \longrightarrow \beta_1 / \beta_2$$

- General Algorithm for Left Factoring

Algorithm: Left Factoring a Grammar

i/p : Grammar G

o/p : Grammar with left factoring eliminated

Method : (i) For each non-terminal A, find the largest prefix α common to two or more of its alternatives.

(ii) If $\alpha \neq \epsilon$, i.e., there is a non-trivial common prefix, replace all the A-productions.

$$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \ldots\ldots\ldots / \alpha\beta_n / \gamma$$

where γ represents all productions that don't begin with 'α'

is replaced as:

$$A \rightarrow \alpha A' / \gamma$$

$$A' \rightarrow \beta_1 / \beta_2 / \beta_3 / \ldots\ldots\ldots / \beta_n$$

where A' is a new non-terminal

(iii) Repeatedly apply this transformation until no two alternatives for a non terminal have a common prefix.

Eg. $S \to iEtS / iEtSeS / a$

$E \to b$

$\Rightarrow S \to iEtSS' / a$

$S' \to eS / \epsilon$

$E \to b$

i = if

E = Expr

t = then

S = stmt

e = else

Eg. $A \to aAB / aA / a$

$B \to bB / b$

$\Rightarrow A \to aA'$

$A' \to AB / A / \epsilon$

$\Downarrow$

$A \to aA'$

$A' \to AA'' / \epsilon$

$A'' \to B / \epsilon$

$B \to bB'$

$B' \to B / \epsilon$

Grammar:

$A \to aA', A' \to AA'' / \epsilon, A'' \to B / \epsilon, B \to bB', B' \to B / \epsilon$

Eg. $E \to E + T / T$ $\quad T \to T * F / F$ $\quad F \to (E) / id$

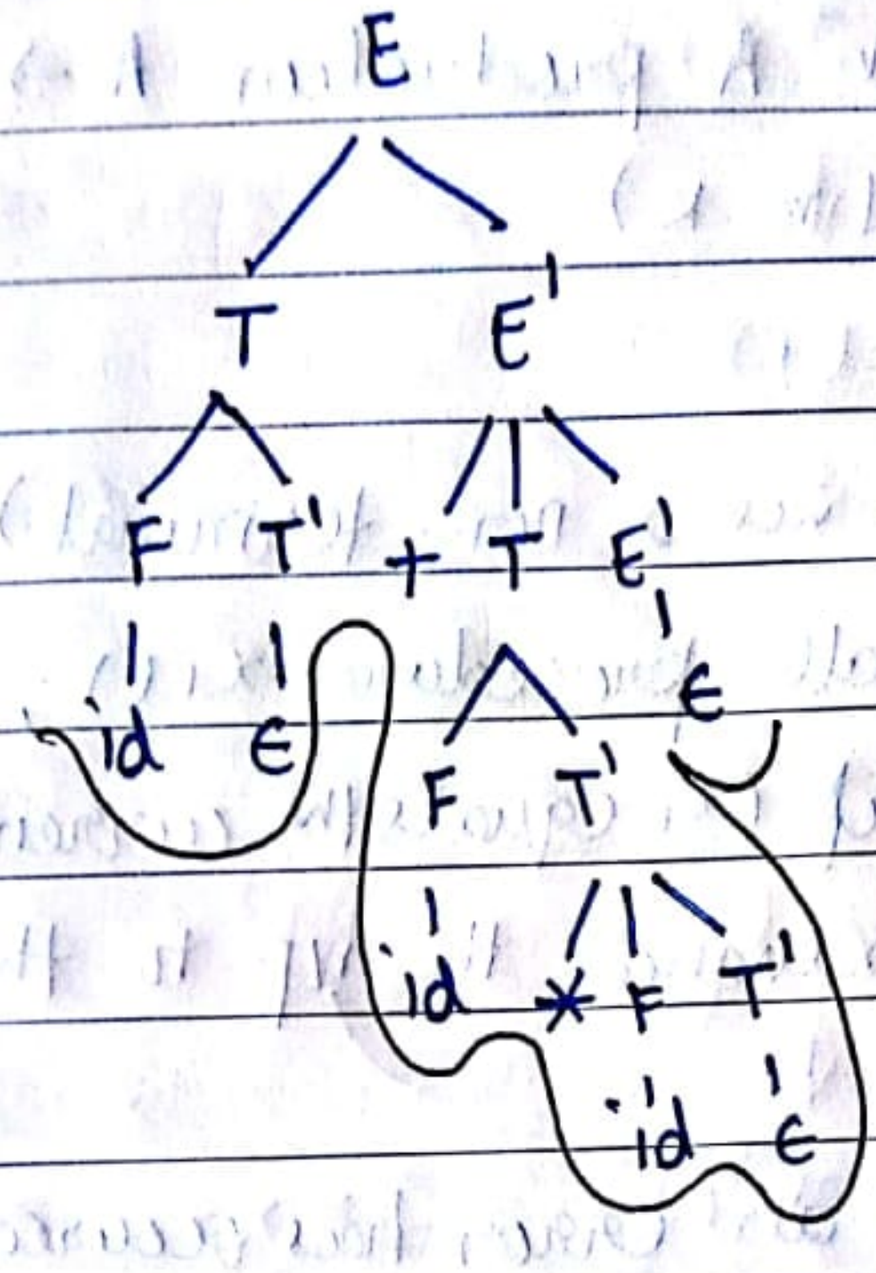
$\Rightarrow E \to TE'$ $\quad T \to FT'$

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

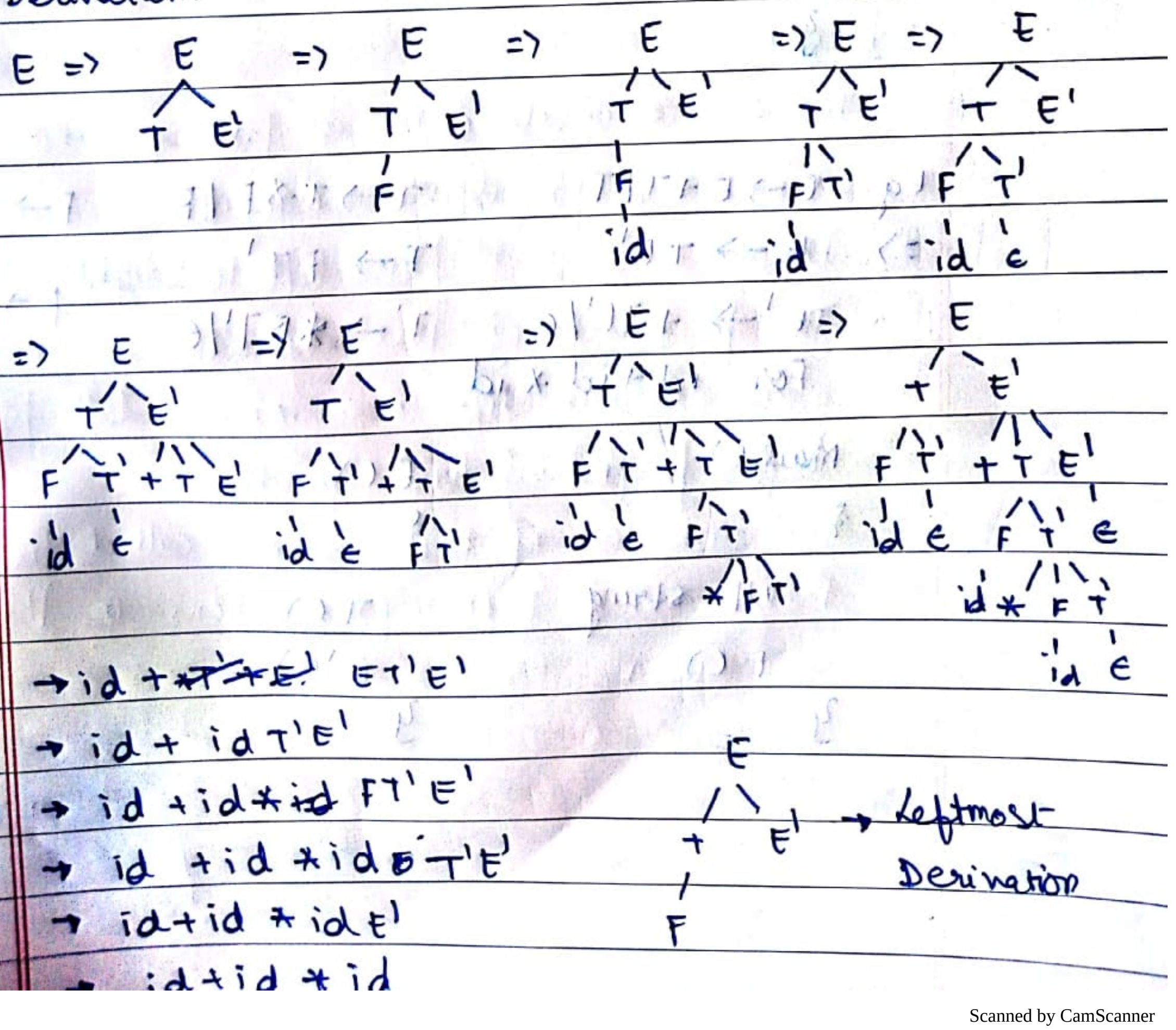
→ id + id * id

Top-Down Parsing

id + id * id $



Derivation:



→ Leftmost Derivation

- Recursive Descent Parsing (RDP)

```
void A()
{
    choose an A production A → x1 x2 ... xk,
    for (i = 1 to k)
    {
        if (xi is a non-terminal)
            call procedure xi();
        else if (xi equals the current i/p symbol a)
            advance the i/p to the next symbol;
        else
            /* an error has occured */
    }
}
```

Eq. $E \to E+T \mid T$ $\quad T \to T*F \mid F$ $\quad F \to (E) \mid id$

$\Rightarrow$ $E \to TE'$ $\quad T \to FT'$

$E' \to +TE' \mid \epsilon$ $\quad T' \to *FT' \mid \epsilon$

For: id + id * id

```
main()
{
    i/p string
    E()
}
```

```
E()
{
    T()
    E'()
}
```

Scanned by CamScanner

```
E'()
{
  if cp='+'
    advance();
    T();
    E'();
}

F()
{
  if cp='('
    advance();
    E();
  if cp=')'
    advance;
  else
    printf (")' enpected");
  else if cp=id
    advance();
  else
    printf("identifier enpected");
```

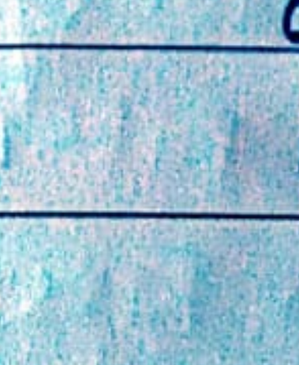
```
T()
{
  F();
  T'();
}

T'()
{
  if cp='*'
    advance();
    F();
    T'();
  else
    return;
}
```

id * id + id

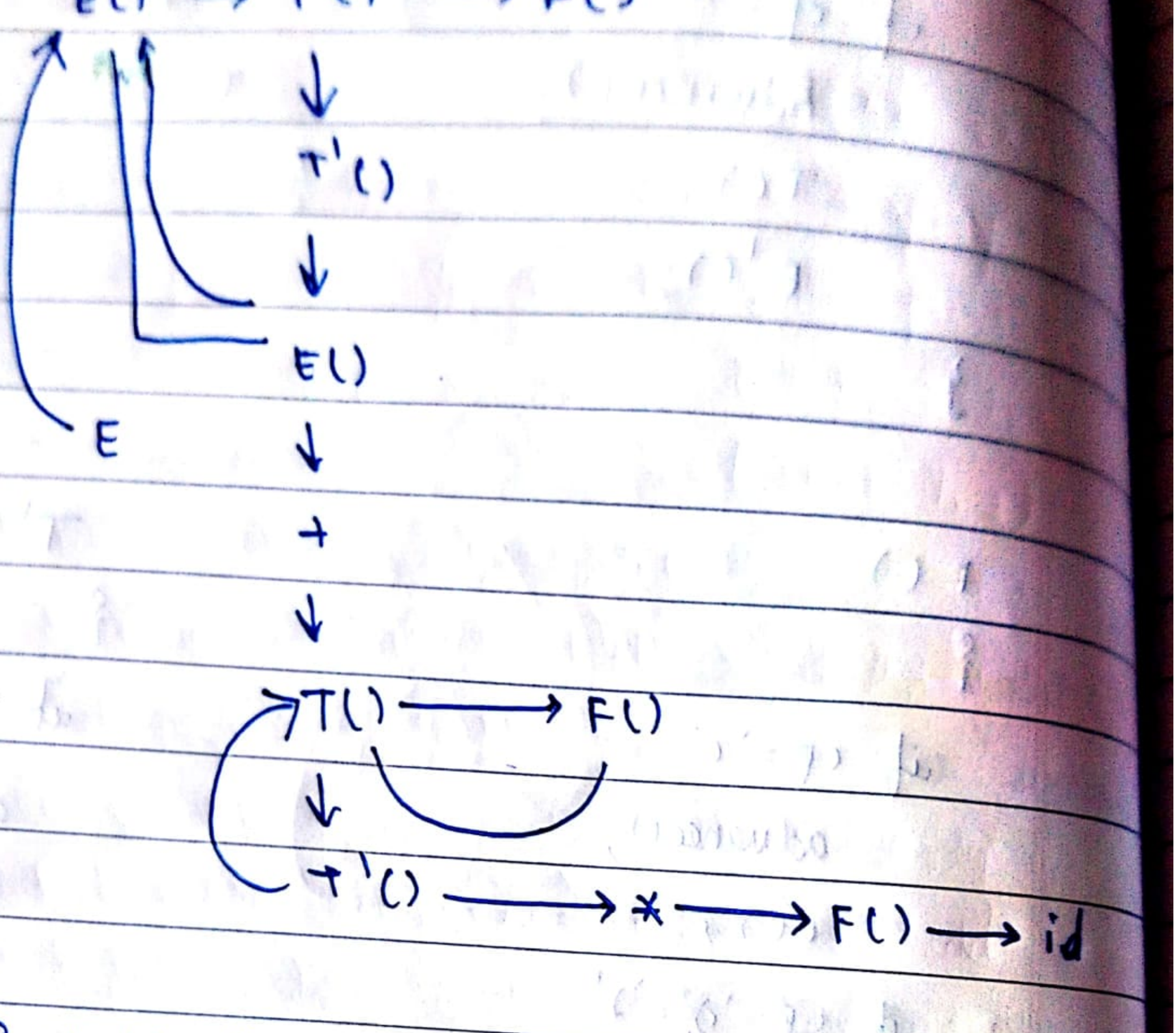
E → E + T | T

T → T * F | F

F → (E) | id



Eg. id + id * id



- Predictive Parsing

- This method is also called as table filling algorithm / table driven algorithm.
- Input it scanned from left to right.
- The prequisites of this algorithm are that we must eliminate left recursion and left factoring before performing this algorithm.
- We make use of two functions called the FIRST() and FOLLOW() functions in this algorithm.

- To compute FIRST(X) for all grammar symbols apply the following rules until no more terminals or ϵ can be added to any FIRST set:

1) If X is a terminal, then FIRST(X) = {X}

2) If X is a non terminal and $X \to Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1) ..... FIRST(Y_{i-1})

that is $Y_1 \dots Y_{j-1} \Rightarrow \epsilon$

If ϵ is in FIRST(Y_j) for all $j = 1, 2, \dots, k$ then add ϵ to FIRST(X)

For eg. everything in FIRST(Y_i) is surely in FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

3) If $X \to \epsilon$ is a production, then add ϵ to FIRST(X)

★ The terminal symbol we get by selecting a particular non-terminal is called FIRST set.

★ The terminals that follow for a particular non-terminal are placed in the FOLLOW set.

Eg. FIRST(a) = {a}

$S \to aBCA \Rightarrow$ FIRST(S) = {a}

$S \to ABCD \Rightarrow$ Let FIRST(A) = {x, ϵ}, FIRST(B) = {y}

FIRST(S) = FIRST(ABCD)

= FIRST(A) = {x, ϵ} ∪ FIRST(BCD)

= {x, y}

$S \to ABCD$ => Let FIRST(A) = {x, ∈}, FIRST(B) = {y, ∈}, FIRST(D) = {a, ∈}

FIRST(C) = {z, € y}

FIRST(S) = FIRST(ABCD) = FIRST(A) U FIRST(BCD)

= {x, ∉) U FIRST(B) U FIRST(CD)

= {x y U {y, ∉} U FIRST(C) U FIRST(D)

= {x, y} U {z, ∉} U FIRST(D)

= {x, y, z} U {a, ∈} = {x, y, z, a, ∈}

Eg. Find out the FIRST set for the grammar:

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

→ After eliminating left recursion and left ~~factoring~~ factoring, we get the productions as:

$E \to TE'$ $\quad T \to FT'$ $\quad F \to (E) \mid id$

$E' \to +TE' \mid \epsilon$ $\quad T' \to *FT' \mid \epsilon$

FIRST(E) = FIRST(TE') = FIRST(T) = FIRST(FT')

= FIRST(F) = { c, id }

∴ FIRST(E) = { c, id } [left parenthesis, id $F \to \underbrace{(E)}_{T} \mid \underbrace{id}_{T}$]

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

FIRST(E') = ~~FIRST~~ { +, ∈ }

FIRST(T') = { *, ∈ }

FIRST(F) = { c, id }

[c, id, +, * = Terminals }

Eg. Find out the FIRST set for the grammar:

$S \to AaAb / BbBa \quad A \to \epsilon \quad B \to \epsilon$

→ FIRST(S) = FIRST(A) = {ε} = a

= FIRST(B) = {ε} = b

∴ FIRST(S) = {a, b} ∴ FIRST(A) = {ε} FIRST(B) = {ε}

[∵ FIRST(S) = FIRST(AaAb) = FIRST(A) = {ε} ∪ FIRST(a) = a

FIRST(S) = FIRST(BbBa) = FIRST(B) = {ε} ∪ FIRST(bBa) = b]

- To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set:

1) Place $ in FOLLOW(S) where S is the start symbol and $ is the input right end marker.

2) If there is a production $A \to \alpha B\beta$ then everything in FIRST(β) except ε is in FOLLOW(B)

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

Eg. $A \to \alpha B\beta$

$\beta \in (V \cup T)^*$

→ The non-terminal B is following the terminals generated by β, i.e., the terminals of β if β is not generating β. FIRST(β) = FOLLOW(B)

→ If β generates ε, the delete ε and FOLLOW(B) = FIRST(β) ∪ FIRST(A)

Eg. Find out the FOLLOW set for the grammar:

$E \to E+|T \qquad T \to T*|F \qquad F \to (E)|id$

→ After eliminating left recursion and left factoring, we get the productions as:

$E \to TE' \qquad T \to FT' \qquad F \to (E)/id$

$E' \to +TE'/\epsilon \qquad T' \to *FT'/\epsilon$

FOLLOW(E):

$F \to (|E|)$

$A \to \alpha|B|\beta$

∴ FOLLOW(E) = {), $}

FOLLOW(E'):

$E' \to +TE'$

$A \to \alpha B$

∴ FOLLOW(E') = FOLLOW(E) = {), $}

$E \to TE'$

$A \to \alpha B$

∴ FOLLOW(E') = FOLLOW(E) = {), $}

$E' \to +TE'$

$\alpha B \beta$

FOLLOW(T):

$E \to TE'$

$A \to \alpha B \beta'$

∴ FOLLOW(T)

= FOLLOW(E) = {), $}

$E' \to +TE'$

$A \to \alpha B \beta'$

FOLLOW(T):

$E' \to +TE'$

$\alpha B \beta$

FOLLOW(T) = FIRST(E') = {+, ϵ} ∪ FOLLOW(E') = {+,), $}

$E' \to +TE'$

$\alpha B \beta$

$A \to B$

$\alpha \; B \; \beta$

Eg. Find out the FIRST set for the grammar:

$S \to AaAb / BbBa \quad A \to \epsilon \quad B \to \epsilon$

→ FIRST(S) = FIRST(A) = {ε} = a

= FIRST(B) = {ε} = b

∴ FIRST(S) = {a, b} ∴ FIRST(A) = {ε} FIRST(B) = {ε}

[∵ FIRST(S) = FIRST(AaAb) = FIRST(A) = {ε} ∪ FIRST(aAb) = {a}

FIRST(S) = FIRST(BbBa) = FIRST(B) = {ε} ∪ FIRST(bBa) = {b}

- To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set:

1) Place $ in FOLLOW(S) where S is the start symbol and $ is the input right end marker.

2) If there is a production $A \to \alpha B \beta$ then everything in FIRST(β) except ε is in FOLLOW(B)

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

Eg. $A \to \alpha B \beta$

$\beta \in (V \cup T)^*$

→ The non-terminal B is following the terminals generated by β, i.e., the terminals of β if β is not generating β. FIRST(β) = FOLLOW(B)

→ If β generates ε, the delete ε and FOLLOW(B) = FIRST(β) ∪ FIRST(A)

Eg. Find out the FOLLOW set for the grammar:

$E \to E+|T \quad T \to T*|F \quad F \to (E)|id$

→ After eliminating left recursion and left factoring, we get the productions as:

$E \to TE' \quad T \to FT' \quad F \to (E)/id$

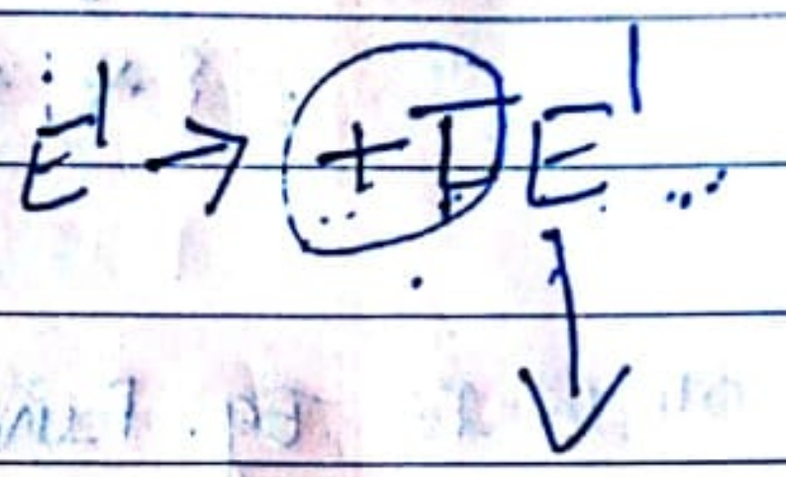
$E' \to +TE'/\epsilon \quad T' \to *FT'/\epsilon$

FOLLOW(E):

$F \to (|E|)$

$A \to \alpha|B|\beta$

∴ FOLLOW(E) = {), $}



FOLLOW(E'):

$E' \to +TE' \quad E \to TE'$

$A \to \alpha B \quad A \to \alpha B$

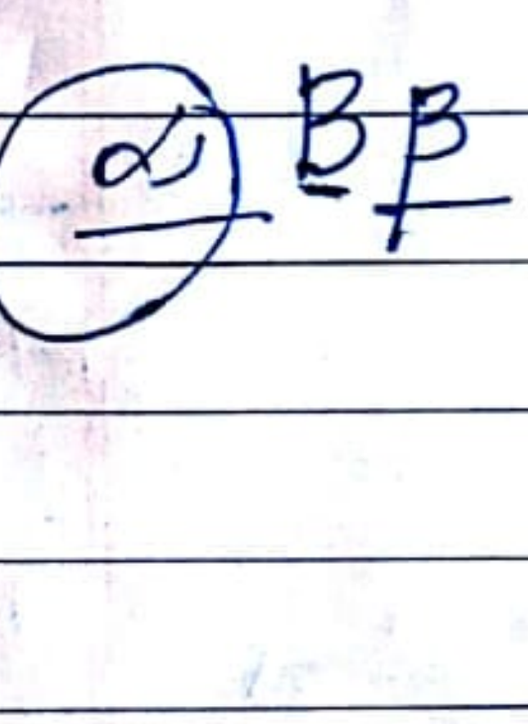
∴ FOLLOW(E') ∴ FOLLOW(E')

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

FOLLOW(T):

$E' \to +TE' \quad E' \to +TE'$

$A \to \alpha B\beta' \quad A \to \alpha B\beta'$



∴ FOLLOW(T)

= FOLLOW(E) = {), $}

FOLLOW(T):

$E' \to +TE' \quad E' \to TE'$

$\alpha B\beta \quad B\beta$

FOLLOW(T) = FIRST(E') = {+, ε} ∪ FOLLOW(E') = {+,), $}

FOLLOW(T'):

T → FT' T' → *FT'

| αB

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

FOLLOW(F):

T → F T' T' → * F T'

| αBβ

FOLLOW(F) = FIRST(T')

= {*, ϵ} = {*} ∪ FOLLOW(T) [∵ FIRST(T') contains ϵ]

= {*, +,), $}

3-01-'18 Eg. Find out FOLLOW(S), FOLLOW(A), FOLLOW(B)

S → AaAb / BbBa ∴ A → ϵ B → ϵ

→ FOLLOW(S) = {$}

FOLLOW(A) => S → AaAb

FOLLOW(A) = FIRST(aAb) ∪ FIRST(b)

= {a} ∪ {b} = {a, b}

FOLLOW(B) => S → BbBa

FOLLOW(B) = FIRST(bBa) ∪ FIRST

= {b} ∪ {a} = {a, b}

Eg. Find out the FIRST and FOLLOW sets for the grammar:

S → aAB / bA / ϵ

A → aAb / ϵ

B → bB / ϵ

→ S → aAB / bA / ∈

FIRST(S) = FIRST(aAB) ∪ FIRST(bA) = {a} ∪ {b} = {a, b}

FIRST(A) = FIRST(aAb) = {a, ∈}

FIRST(B) = FIRST(bB) = {b, ∈}

FOLLOW(S) = {$}

FOLLOW(A)

⇒ S → aAB / bA A → aAb

αBβ αB αBβ

• S → aAB ⇒ FIRST(B) = {b, ∈} ∪ FOLLOW(S) = {b} ∪ {$} = {b, $}

• S → bA ⇒ FOLLOW(S) = {$}

• S → aAb ⇒ FIRST(b) = {b}

∴ FOLLOW(A) = {b, $}

FOLLOW(B)

⇒ S → aAB B → bB S → aAB

αB αB

• S → aAB ⇒ FOLLOW(S) = {$}

• B → bB ⇒ FOLLOW(B) = {$}

∴ FOLLOW(B) = {$}

Eg. Find the FIRST and FOLLOW functions for the grammar:

S → ictSA / a A → eS / ∈ C → b

→ No left recursion and left factoring in the grammar

FIRST(S) = FIRST(ictSA) ∪ FIRST(a) = {i, a}

FIRST(A) = FIRST(eS) = {e, ∈}

FIRST(C) = FIRST(b) = {b}

FOLLOW(S) = {$} [For start symbol]

⇒ S → ictSA A → eS

αBβ αB

• S → ictSA ⇒ FIRST(A) = {e, ∈} ∪ FOLLOW(S) = {e, $}

• A → eS ⇒ FOLLOW(A)

$\therefore$ FOLLOW(S) = {e, $} $\cup$ FOLLOW(A) $\leftarrow$ = {e, $}

FOLLOW(A)

$\Rightarrow S \longrightarrow i\ c\ t\ S\ A$

$\alpha \quad B$

$\therefore$ FOLLOW(A) = FOLLOW(S) = {e, $} ~~$\cup$ FOLLOW(A)~~ = {e, $}

FOLLOW(C)

$\Rightarrow S \longrightarrow i\ \underbrace{t\ S\ A}$

$\alpha\ B\ \beta$

FOLLOW(C) = FIRST(tSA) = {t}

$\therefore$ FIRST(S) = {i, a} FOLLOW(S) = {e, $}

FIRST(A) = {e, $\in$} FOLLOW(A) = {e, $}

FIRST(C) = {b} FOLLOW(C) = {t}

Eg. Find FIRST and FOLLOW functions for the grammar:

$S \rightarrow (L) / a$ $L \rightarrow L, S / S$

$\rightarrow L \rightarrow L,S / S$

$L \rightarrow SL'$ } eliminating left recursion $S \rightarrow (L) / a$

$L' \longrightarrow ,SL' / \in$ }

FIRST(S) = {(, a}

FIRST(L) = FIRST(S) = {(, a}

FIRST(L') = {,, $\in$}

FOLLOW(S) = {)}

$\Rightarrow L \longrightarrow SL'$ $L' \longrightarrow ,SL'$

$\alpha B \beta$ $\alpha B \beta$

- $L \rightarrow SL' \Rightarrow$ FOLLOW(S) = FIRST(L') = {,, $\in$} $\cup$ FOLLOW(L)

= {,} $\cup$ FOLLOW(L)

- $L' \rightarrow ,SL' \Rightarrow$ FOLLOW(S) = FIRST(L') = {,, $\in$} $\cup$ FOLLOW(L')

= {,} $\cup$ FOLLOW(L')

FOLLOW(S) = {,, $} $\cup$ FOLLOW(L) $\cup$ FOLLOW(L')

= {,, $} $\cup$ {)} $\cup$ {)} = {,, $,)}

FOLLOW(L)

=> S → (L)

αBβ

FOLLOW(L) = FIRST()) = {)}

FOLLOW(L')

=> L → SL'

αB

L → ,SL'

αB

• L → SL' => FOLLOW(L') = FOLLOW(L) = {)}

• L' → ,SL' => FOLLOW(L') = FOLLOW(L') = {)}

∴ FIRST(S) = {(, a}

FIRST(L) = {(, a}

FIRST(L') = {,, ∈}

FOLLOW(S) = {$,)}

FOLLOW(L) = {)}

FOLLOW(L') = {)}

Eg. Find the FIRST and FOLLOW functions for the grammar:

E → E+T | T

T → TF | F

F → F* | a | b

→ E → E+T | T

E → TE'

E' → +TE' / ∈

T → TF | F

T → FT'

T' → FT' / ∈

F → F* | a | b

F → *F'

F' → aF' / bF' / ∈

↳ We eliminate left recursion from the grammar

★ FIRST(E) = FIRST(T) = FIRST(F) = {*}

★ FIRST(T) = FIRST(F) = {*}

★ FIRST(F) = {*}

★ FIRST(E') = {+, ∈}

★ FIRST(T') = FIRST(F) ∪ {∈} = {*, ∈}

★ FIRST(F') = {a, b, ∈}

FOLLOW(E)

=> E → E + T

αB β

FOLLOW(E) = FIRST(β)

= FIRST(+T) = {+}

08-01-'19

LL(1) Parser

- It is a table driven parser.
- The grammar must not be left recursive grammar or ambiguous grammar.
- Disadvantage of Recursive Descent Parser.
  * May enter infinite loop if recursive depth is more.
  * In syntax analysis phase, if we are not able parse the string, it must give error. But recursive descent parser is not good for error messaging
  * Lookahead signals are long.
- To overcome the disadvantages of recursive descent parser, we use LL(1) parser.
- Recursive descent parser uses Brute Force method
- LL(1) parser is a table driven parser which uses dynamic programming.
- LL1 parser:
  1st L: i/p is scanned from left to right.
  2nd L: derives left most derivation.
  '1' : no. of look ahead symbols is 1
- Predictive Parsing Table:
  • To construct a table and predict whether the string is
- Before performing predictive parsing, for the given CFG, we must eliminate left recursion and left factoring, make the grammar unambiguous and find out the FIRST and FOLLOW sets of the grammar.

- Algorithm to Construct Predictive Parsing Table:
  → I/P : The Content Free Grammar 'G'
  → O/P : Predictive Parsing Table 'M'
  → Algorithm: For each production $A \to \alpha$ of the grammar, do the following:
    i) For each terminal 'a' in FIRST(α), add $A \to \alpha$ to M(A, α)
    ii) If ϵ is in FIRST(α), then for each terminal 'b' in FOLLOW(A), add $A \to \alpha$ to M[A,b]
    If ϵ is in FIRST(A) and $ is in FOLLOW(A), add M(A,$) as well.
  → If after performing the above, there is no production at all in M[A,a], then set M[A,a] to error.

Eg: $E \to E+T/T$ $T \to T*F/F$ $F \to (E)/id$

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

	FIRST()	FOLLOW()
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{), +, $}
T'	{+, ε}	{), +, $}
F	{(, id}	{), +, *, $}

→ Now, take variables as rows and terminals as columns in the next table

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

- E→TE' FIRST(TE') = FIRST(T) = FIRST(F) = {(, id}
  M[E, (] = E→TE' M[E, id] = E→TE'
- E'→+TE' FIRST(+TE') = {+} M[E', +] = E'→+TE'
- E'→ϵ FOLLOW(E') = {), $}
  M[E',)] = E'→ϵ M[E', $] = E'→ϵ
- T→FT' FIRST(FT') = FIRST(F) = {(, id}
  M[T, (] = T→FT' M[T, id] = T→FT'
- T'→*FT' FIRST(*FT') = {*} M[T, *] = T'→*FT'
- T'→ϵ A→α α=>ϵ ⇒ FOLLOW(T') = {), +, $}
  M[T',)] = T'→ϵ FIRST(A) = FIRST(T') = {*, ϵ}
  M[T', +] = T'→ϵ FOLLOW(A) = FOLLOW(T') = {), +, $}
  M[T', $] = T'→ϵ ↳ M[T', $] = T'→ϵ
- F→(E) FIRST(F) = {(} M[F, (] = F→(E)
- F→id FIRST(F) = {id} M[F, id] = F→id

* All empty blocks/cells are error entries
* If for any cell, we get two productions, the grammar is not LL1 grammar and is ambiguous.

Eg. E → E+T | T T → TF | F F → F* | a | b

→ E → TE' T → FT' F → *aF' | bF'

E' → +TE' | ϵ T' → FT' | ϵ F' → *F' | ϵ

Check: a+b*a

a*b+b

	FIRST	FOLLOW
E	{a, b}	{$}
E'	{+, ϵ}	{$}
T	{a, b}	{+, $}
T'	{a, b, ϵ}	{+, $}
F	{a, b}	{+, a, b, $}
F'	{*, ϵ}	{+, a, b, $}

	a	b	+	*	$
E	E→TE'	E→TE'			
E'			E'→+TE'		E'→ϵ
T	T→FT'	T→FT'			
T'	T'→FT'	T'→FT'	T'→ϵ		T'→ϵ
F	F→aF'	F→bF'			
F'	F'→ϵ	F'→ϵ	F'→ϵ	F'→*F'	F'→ϵ

↳ The grammar is unambiguous and parsed by LL1

- $E \to TE'$ FIRST(TE') = FIRST(T) = FIRST(F) = {a,b}

  M[E,a] = $E \to TE'$ M[E,b] = $E \to TE'$

- $E' \to +TE'$ FIRST(+TE') = {+} M[E', +] = $E' \to +TE'$

- $E' \to \epsilon$ $\alpha \Rightarrow \epsilon$ ⇒ FOLLOW(E') = {$,)} M[E', $] = $E' \to \epsilon$

- $T \to FT'$ FIRST(FT') = FIRST(F) = {a, b}

  M[T,a] = $T \to FT'$ M[T, b] = $T \to FT'$

- $T' \to FT'$ FIRST(FT') = FIRST(F) = {a,b}

  M[T',a] = $T' \to FT'$ M[T', b] = $T' \to FT'$

- $T' \to \epsilon$ $\alpha \Rightarrow \epsilon$ ⇒ FOLLOW(T') = {+, $,)}

  M[T', +] = $T' \to \epsilon$ M[T', $] = $T' \to \epsilon$

- $F \to aF'$ FIRST(aF') = a M[F, a] = $F \to aF'$

- $F \to bF'$ FIRST(bF') = b M[F, b] = $F \to bF'$

- $F' \to *F'$ FIRST(*F') = * M[F', *] = $F \to *F'$

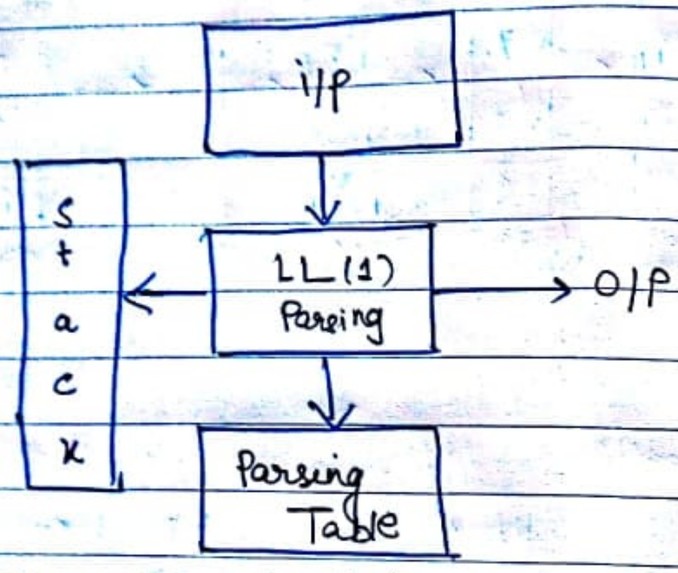
- $F' \to \epsilon$ $\alpha \to \epsilon$ ⇒ FOLLOW(F') = {+, a, b, $,)}

  M[F', a] = $F' \to \epsilon$ M[F', b] = $F' \to \epsilon$ M[F', +] = $F' \to \epsilon$ M[F', $] =

Eg. $S \to ictsA/a$ $A \to eS/\epsilon$ $C \to b$

09-01-'19

- Text Parsing.



- LL(1) parser gives better error message compared to recursive descent parser.
- Table Driven Parsing : Algorithm
  - → i/p : A string 'w' and a parsing table 'M' for grammar 'G'.
  - → o/p : If 'w' is in 'L(G)', a leftmost derivation of 'w'. Otherwise, an error indication.
  - → Method : Initially, the parser is in a configuration with 'w$' - in the input buffer. Start symbol S of G is on top of the stack above $.

```
Let 'a' be the first symbol of 'w'.
Let 'x' be the top stack symbol.
While (x ≠ $) stack is not empty
{
  if (x == a)
    pop the stack and 'a' be the next symbol of 'w';
  else if (x is a terminal)
    error;
  else if (M[x,a] is an error entry)
    error;
  else if (M[x,a] = x → Y1, Y2, Y3, ....., Yk)
  {  pop the stack;
     push Yk, Yk-1, Yk-2, ....., Y1 onto the stack
       with Y1 on top;
  }
  Let 'x' be the top stack symbol;
}
```

Eg. E → E+T | T T → T*F | F F → (E) | id

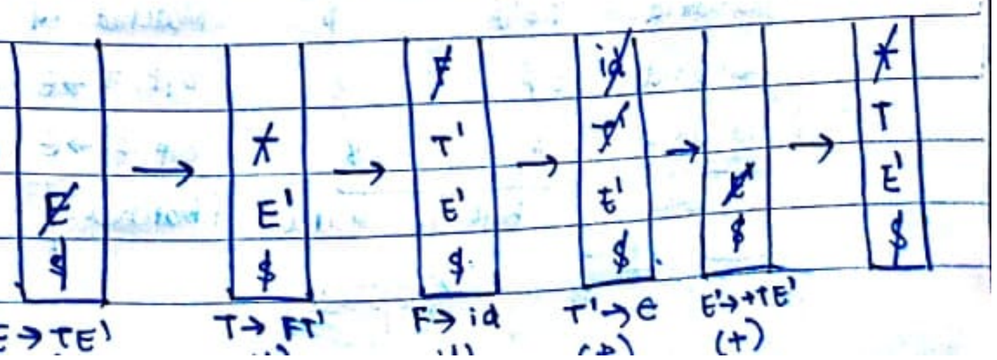
→ Table:

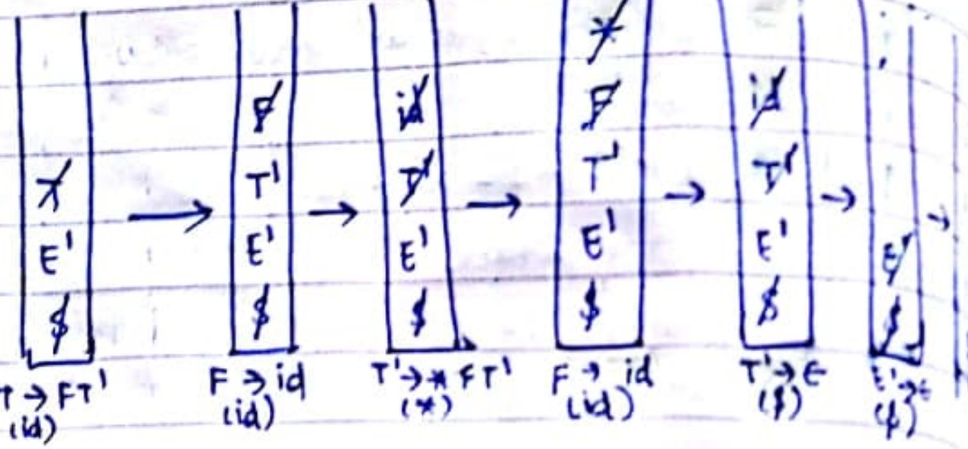
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ϵ	E' → ϵ
T	T → FT'			T → FT'		
T'		T' → ϵ	T' → *FT'		T' → ϵ	T' → ϵ
F	F → id			F → (E)		

* If TOS is a terminal, the current i/p symbol and TOS must match. If TOS is a non terminal, then there must be an entry with non-terminal and i/p symbol in table.

→ id + id * id $

* When current i/p symbol matches TOS, pop TOS





→ Table:

Method	Stack	Input	Action
	E$	id+id*id$	
	TE'$	id+id*id$	O/P: E→TE'
	FT'E'$	id+id*id$	O/P: T→FT'
	id T'E'$	id+id*id$	O/P: F→id
id	T'E'$	+id*id$	Matched id
id	E'$	+id*id$	O/P: T'→∈
id	+TE'$	+id*id$	O/P: E'→+TE'
id+	TE'$	id*id$	Matched +
id+	FT'E'$	id*id$	O/P: T→FT'
id+	id T' E'$	id*id$	O/P: F→id
id+id	T'E'$	*id$	Matched: id
id+id	*FT'E'$	*id$	O/P: T'→*FT'
id+id*	FT'E'$	id$	Matched: *
id+id*	idT'E'$	id$	O/P: F→id
id+id*id	T'E'$	$	Matched: id
id+id*id	E'$	$	O/P: T'→∈
id+id*id	$	$	O/P: E'→∈

Leftmost Derivation

i/P string — Both are equal

⇒ i/P string accepted

Eg: id+id$ id*id id$

→ Table:

Method	Stack	Input	Action
	E$	id+id$	
	TE'$	id+id$	O/P: E→TE'
	FT'E'$	id+id$	O/P: T→FT'
	id T'E'$	id+id$	O/P: F→id
id	T'E'$	+id$	Matched: id
id	E'$	+id$	O/P: T'→∈
id	+TE'$	+id$	O/P: E'→+TE'
id+	TE'$	id$	Matched: +
id+	FT'E'$	id$	O/P: T→FT'
id+	id T'E'$	id$	O/P: F→id
id+id	T'E'$	$	Matched: id
id+id	E'$	$	O/P: T'→∈
id+id	$	$	O/P: E'→∈

i/P string — Both are equal

⇒ i/P string is accepted.

Eq: id*id id*

Method	Stack	Input	Action
	E$	id*id id$	
	TE'$	id*id id$	O/P: E→TE'
	FT'E'$	id*id id$	O/P: T→FT'E'
	id T'E'$	id*id id$	O/P: F→id
id	T'E'$	*id id$	Matched: id
id	*FT'E'$	*id id$	O/P: T'→*FT'
id*	FT'E'$	id id$	Matched: *

id *	FT'E'$	id id $	Matched: *
id *	idT'E'$	id id $	O/P: F→id
id * id	T'E'$	id $	Matched: id
id * id	T'E'$	id $	O/P: T'→ε, error

not i/p string — not equal

⇒ i/p string is not accepted