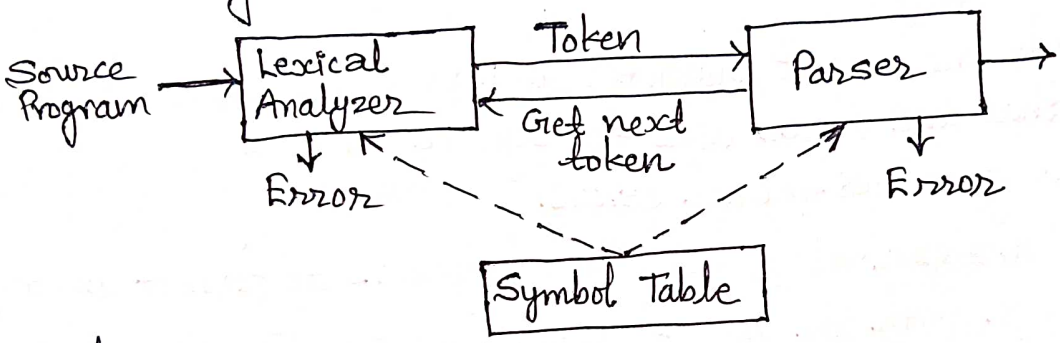
lexical analysis is done for validation of tokens

UNIT-2
Lexical Analyzer

Lexical Analysis: The lexical analysis is the first phase of a compiler where a lexical analyzer acts as an interface between the source program and the rest of the phases of compiler. It reads input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are then sent to the parser for syntax analysis.



Example: Example of Lexical Analysis, Tokens, Non-Tokens

Consider the following code that is fed to Lexical Analyzer.

```
#include <stdio.h>
int Largest (int x, int y)
{
   if (x>y) //This will compare two numbers
      return x;
   else
      return y;
}
```

Examples of Tokens created:

Lexeme	Token
int	Keyword
largest	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
y	Identifier
)	Operator
{	Operator
if	Keyword

Examples of Non-Tokens:

Type	Examples
Comment	// This will compare 2 numbers.
Pre-processor directive	#include <stdio.h>

⊛. Role of Lexical Analyzer:

→ Lexical analyzer helps to identify token into the symbol table.
→ It can either work as a seperate module or as a sub-module.
→ It is responsible for eliminating comments and white spaces from the source program.
→ It reads the input character and produces output sequence of tokens that the Parser uses for syntax analysis.
→ It also generates lexical errors.
→ Lexical analyzer is used by web browsers to format and display a web page with the help of parsed data from Javascript, HTML, CSS.

⊛. Lexemes, Patterns, Tokens:

Lexemes: A lexeme is a sequence of alphanumeric characters that is matched against the pattern for token. A sequence of input characters that make up a single token is called a lexeme. A token can represent more than one lexeme.
Example: The token "String constant" may have a number of lexemes such as "bh", "sum", "area", "name" etc.

Patterns: Patterns are the rules for describing whether a given lexeme belonging to a token or not. The rule associated with each set of string is called pattern. Lexeme is matched against pattern to generate token. Regular expressions are widely used to specify patterns.

Token: Token is word, which describes the lexeme in source program. It is generated when lexeme is matched against pattern. A token is a logical building block of language. They are the sequence of characters having a collective meaning.

Example1: Example showing lexeme, token and pattern for variables.

- Lexeme: A1, Sum, Total
- Pattern: Starting with a letter and followed by letter or digit but not a keyword.
- Token: ID

Example2: Example showing lexeme, token and pattern for floating number.

- Lexeme: 123.45
- Pattern: Starting with digit followed by a digit or optional fraction and or optional exponent.
- Token: NUM

⊛Specifications of Tokens:

Regular Expression is a way to specify tokens. The regular expression represents the regular languages. The language is the set of strings and string is the set of alphabets. Thus, following terminologies are used to specify tokens:

1) **Alphabets:** The set of symbols is called alphabets. Example any finite set of symbols $\Sigma = \{0,1\}$ is a set of binary alphabets.

2) **Strings:** Any finite sequence of alphabets is called a string. Length of string is the total number of occurance of alphabets. e.g, the length of string 'Kanchanpur' is 10. A string of zero length is known as empty string and is denoted by ε (epsilon).

3) **Language:** A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them:

   1) **Union:** It is defined as the set that consists of all elements belonging to either set A or set B or both. In regular expression + symbol is used to represent union operation.

$$A = \{dog, ba, na\} \text{ and } B = \{house, ba\}$$

$$\text{then, } A \cup B = A + B = \{dog, ba, na, house\}$$

ii) Concatenation: It is the operation of joining character strings end-to-end. In regular expression dot operator (.) is used to represent concatenation operation.

$A.B = \{doghouse, dogba, bahouse, baba, nahouse, naba\}$

iii) Kleen closure: The Kleene closure Σ^* gives the set of all possible strings of all possible lengths over Σ including $\{\Phi\}$.

iv) Positive Closure: The set Σ^+ is the infinite set of all possible strings of all possible lengths over Σ excluding $\{\Phi\}$.

Regular Expressions: Regular expressions are the algebraic expressions that are used to describe tokens of a programming language. It uses three regular operations called union, concatenation and star.

Examples: Given the alphabet $A=\{0,1\}$.

1) $1(1+0)^*0$ denotes the language of all strings that begins with a '1' and ends with a '0'.

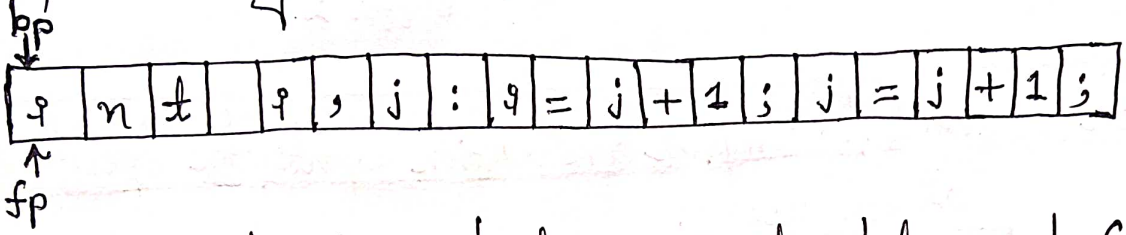
2) $(01)^* + (10)^*$ denotes the set of all strings that describe alternating 1s and 0s. [Note: Practice R.E for identifier in C, one dimensional array, two dimensional array & floating numbers. (6, 10, 11, 12 no. in kec book under RE topic)]

Recognition of Tokens: A recognizer for a language is a program that takes a string w, and answers "Yes" if w is a sentence of that language, otherwise "NO." The tokens that are specified using RE are recognized by using transition diagram or finite automata (FA). Starting from the start state we follow the transition defined. If the transition leads to the accepting state, then the token is matched and hence the lexeme is returned, otherwise other transition diagrams are tried out until we process all the transition diagram or the failure is detected. Recognizer of tokens takes the language L and the string s as input and try to verify whether $s \in L$ or not. There are two types of Finite Automata.

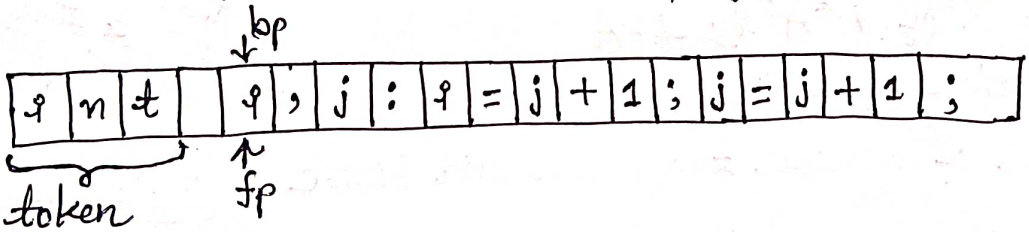
1). Deterministic Finite Automata (DFA).

2). Non Deterministic Finite Automata (NFA).

⊛. **Input Buffering:** Reading character by character from secondry storage is slow process and time consuming as well so, we use buffer technique to eliminate this problem and increase efficiency. The lexical analyzer scans the input from left to right one character at a time. It uses two pointers, begin ptr (bp) and forward ptr (fp) to keep track of the pointer of input scanned. Initially both the pointers point to the first character of the input string as shown below;



The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as fp encounters a blank space the lexeme 'int' is identified. Then both bp and fp are set at next token as shown below and this process will be repeated for the whole program.

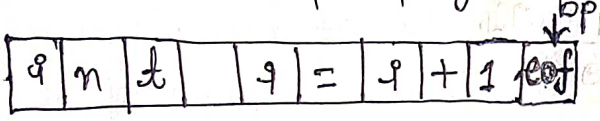


One Buffer Scheme: In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundry, to scan rest of the lexeme the buffer has to be refilled.

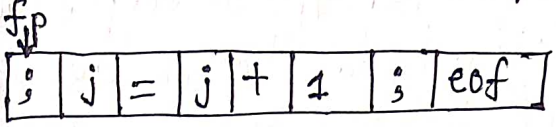
Two Buffer Scheme: In this scheme, two buffers are used to store input string and they are scanned alternately. When end of current buffer is reached the other buffer is filled. The only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves forward in search of

end of lexeme. As soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundry of first buffer end of buffer character (eof) should be placed at the end of first buffer, similarly for second buffer also. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.



Buffer 1



Buffer 2.

⊛ Read Following basic topics from 4th sem note of TOC (Unit-2 and Unit-3):

1) Finite Automata.
2) Deterministic Finite Automata (DFA).
3) Non Deterministic Finite Automata (NFA).
4) Epsilon NFA (ε-NFA).
5) Minimization of DFA. (state partition method) (same TOC में पढ़ा है)
6) Equivalence of Regular Expression and Finite Automata.
7) Reduction of Regular Expression to ε-NFA.
8) Converting ε-NFA to its equivalent DFA.

May not be asked in exam but these topics are necessary topics to understand for upcomming topics. These are basics.

⊛ Design of a Lexical Analyzer: [Imp]

Among these two algorithms one is asked in exam. Here if we see NFA then, it is always ε-NFA

Lexical Analyzer can be designed using following two algorithms:

Algorithm1: Regular Expression → NFA → DFA (two steps: first RE to NFA, then NFA to DFA).

Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA).

Same

#Algorithm1:

This consists of following two steps:

i.e, Reduction of Regular Expression to ε-NFA that we read in TOC Unit-3

1) Regular Expression to NFA (Thomson's Construction):

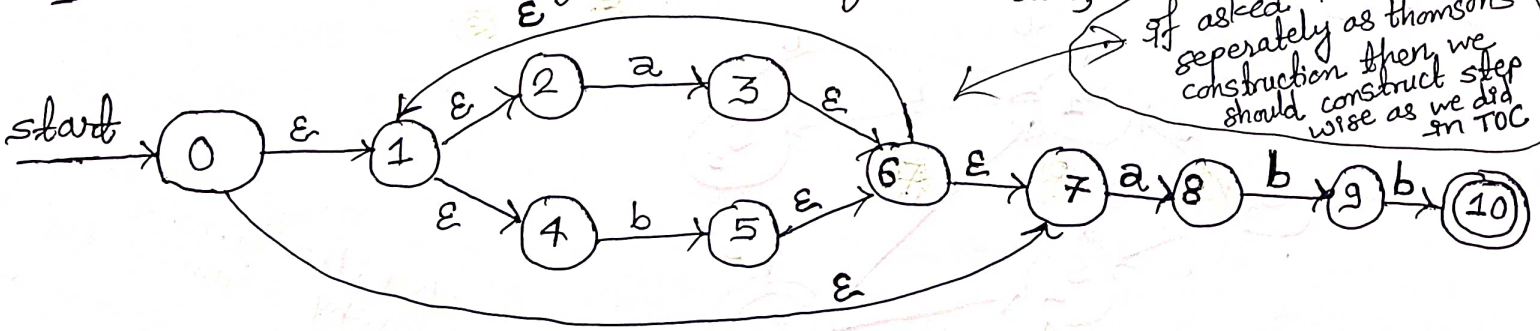
i.e, Converting ε-NFA to its equivalent DFA that we read in TOC Unit-2

2) Conversion from NFA to DFA (Subset Construction Algorithm):

Example: For Regular Expression $(a+b)^*abb$, first convert this RE to NFA, then convert resulting NFA to DFA.

Solution: The NFA of given RE is given below;

If asked in exam seperately as thomson's construction then we should construct step wise as we did in TOC



Now we convert above NFA to DFA as,

The starting state of DFA = S_0 = ε-closure (S_0)

(starting state र start state बाट ε लिएर कुन कुन state सम्म पुगेको छ त्यसको list)

(Since we have two input symbols a, & b so we check for each input) ⇒ $S_0 = \{0, 1, 2, 4, 7\}$

Mark S_0,

($S_0 = 0,1,2,4,7$ so S_0 मा a आउदा 3, र 8 मा गएको छ)

For a: ε-closure($\delta(S_0, a)$) = ε-closure(3,8) = $\{1,2,3,4,6,7,8\} \rightarrow S_1$

For b: ε-closure($\delta(S_0, b)$) = ε-closure(5) = $\{1,2,4,5,6,7\} \rightarrow S_2$

(कुनै नयाँ set आउदा नयाँ state ले denote गर्ने like here S_1)

(just denote गरेको यदि DFA बनाउदा सजिलो होस् भनेर)

(previous step मा आएको नयाँ set check गर्ने next step मा)

Mark S_1,

For a: ε-closure($\delta(S_1, a)$) = ε-closure(3,8) = $\{1,2,3,4,6,7,8\} \rightarrow S_1$

For b: ε-closure($\delta(S_1, b)$) = ε-closure(5,9) = $\{1,2,4,5,6,7,9\} \rightarrow S_3$

Mark S_2,

For a: ε-closure($\delta(S_2, a)$) = ε-closure(3,8) = $\{1,2,3,4,5,7,8\} \rightarrow S_1$

For b: ε-closure($\delta(S_2, b)$) = ε-closure(5) = $\{1,2,4,5,6,7\} \rightarrow S_2$

Mark S_3,

For a: ε-closure($\delta(S_3, a)$) = ε-closure(3,8) = $\{1,2,3,4,6,7,8\} \rightarrow S_1$

For b: ε-closure($\delta(S_3, b)$) = ε-closure(5,10) = $\{1,2,4,5,6,7,10\} \rightarrow S_4$

Mark S_4,

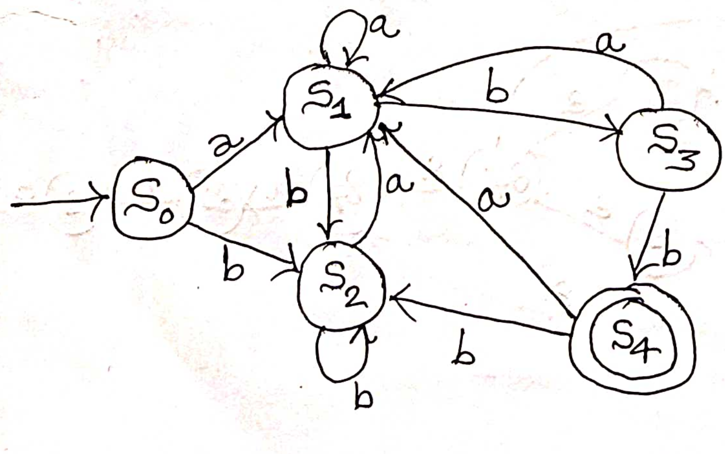
For a: ε-closure($\delta(S_4, a)$) = ε-closure(3,8) = $\{1,2,3,4,6,7,8\} \rightarrow S_1$

For b: ε-closure($\delta(S_4, b)$) = ε-closure(5) = $\{1,2,4,5,6,7\} \rightarrow S_2$.

(No new states here S_1, S_2 are already checked so we stop here. We stop when there are no any new states to check)

S_0 is the starting state. S_4 is an accepting state of DFA since final state of NFA is 10 i.e, states containing 10 are member of final state of DFA. (Since $S_4 = \{1,2,4,5,6,7,10\}$).

Now, constructing DFA using above information:



Note: Practice more examples from kec book

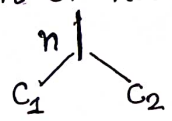
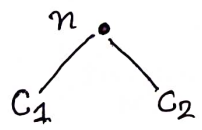
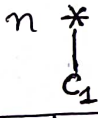
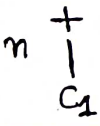
#Algorithm 2 (Conversion from RE to DFA Directly):

Conversion steps:

for exam mostly numericals are asked theory for understanding only

1. Augment the given regular expression by concatenating it with special symbol #.
2. Create the syntax tree for this augmented regular expression.
3. Then each alphabet symbol (including #) with be numbered as position numbers.
4. Compute functions nullable, firstpos, lastpos, and followpos.
5. Finally construct DFA directly from a regular expression by computing the functions nullable(n), firstpos(n), lastpos(n), and followpos(i) from the syntax tree.

Rules for calculating nullable, firstpos and lastpos:

Node n	nullable (n)	firstpos (n)	lastpos (n)
A leaf labelled ε	True	ϕ	ϕ
A leaf with position i	False	{i} (position of leaf node)	{i}
An or node 	Nullable (c_1) or Nullable (c_2).	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
A concatenation node 	Nullable (c_1) and Nullable (c_2).	If (Nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2). else firstpos(c_1)	If(Nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2). else lastpos(c_2).
A star node 	True	firstpos(c_1)	lastpos(c_1)
A +ve closure node 	False	firstpos(c_1)	lastpos(c_1)

Now we calculate followpos for each i we numbered. Now we can construct DFA with starting state S_1 = firstpos(root) from syntax tree and marking each state for each input until no new unmarked states occur.

Example: Convert the regular expression $(a|b)^*a$ into equivalent DFA by direct method.

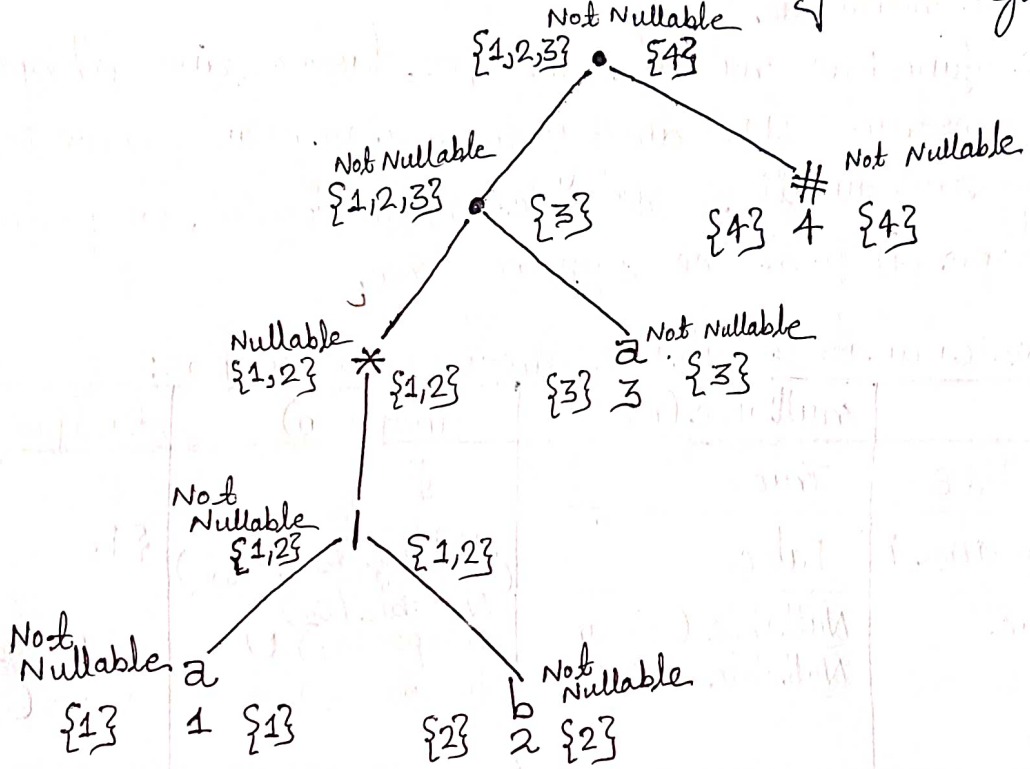
Solution:

Step1: At first we augment the given regular expression as,

$(a|b)^*.a.\#$ ← last मा # थपे augment हुन्छ। अनि operator हरु product form मा हुने भएकाले dot operator हुन्छ।

1 2 3 4 ← alphabets including # numbered

Step2: Now we construct syntax tree of augmented regular expression as,



Step3: Now we compute followpos as,

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {∅} ← empty

Step4: Now we start with starting state as,

S_1 = firstpos(root node of syntax tree) = {1,2,3}

Since we are marking for a for S_1 = {1,2,3}. Now look at step1 where where a occurs in 1,2,3. Here a occurs in 1 & 3 so Union of 1 & 3

Mark S_1,

For a: followpos(1) ∪ followpos(3) = {1,2,3,4} → S_2 ← marked as new state using variable or state name S_2

In step 1 between 1,2,3 b occurs at 2 only

for b: followpos(2) = {1,2,3} → S_1

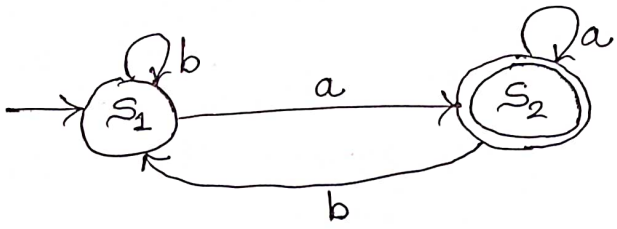
Mark S_2,

For a: followpos(1) ∪ followpos(3) = {1,2,3,4} → S_2

for b: followpos(2) = {1,2,3} → S_1.

No new states occur so we stop marking here. Accepting state means state containing position of # (i.e, 4). So, here S_2 = {1,2,3,4} is accepting state.

Now, based on the above information, resulting DFA of given regular expression is as follows:



Practice these questions also if any confusion or want to try to match answer then refer ref book for solution page no 37.

Q1. Convert the regular expression $(a|\varepsilon)bc^*$ into equivalent DFA by direct method.

Q2. Convert the regular expression $ba(a+b)^*ab$ into equivalent DFA by direct method.

⊛ Flex: An Introduction

possibility of asking in short note for 2.5 marks

Flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, 'lex.yy.c' by default, which defines a routine yylex(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Flex Specification: A flex specification consists of three parts:

Regular definitions, C declarations in %{ %}

%%

Translation rules

This is syntax. %% are opening and closing tags.

%%

User defined auxiliary procedures

The translation rules are of the form:

P_1	{action 1}
P_2	{action 2}
. . .	. . . . .
P_n	{action n}

In all parts of the specification comments of the form /* comment text */ are permitted.

Syntax Analyzer

about 20 marks को सोचके, पढ़ो बाद

Syntax Analysis: All programming languages have certain syntactic structures. We need to verify the source code written for a language is syntatically valid. The validity of the syntax is checked by the syntax analysis. Syntaxes are represented using context free grammar (CFG), or Backus Naur Form (BNF). Parsing is the act of performing syntax analysis to verify an input program's compliance with the source language. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens.

Syntax analyzer and Parser are same thing

Role of Syntax Analyzer/Parser:

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It has to report any syntax errors if occurs.

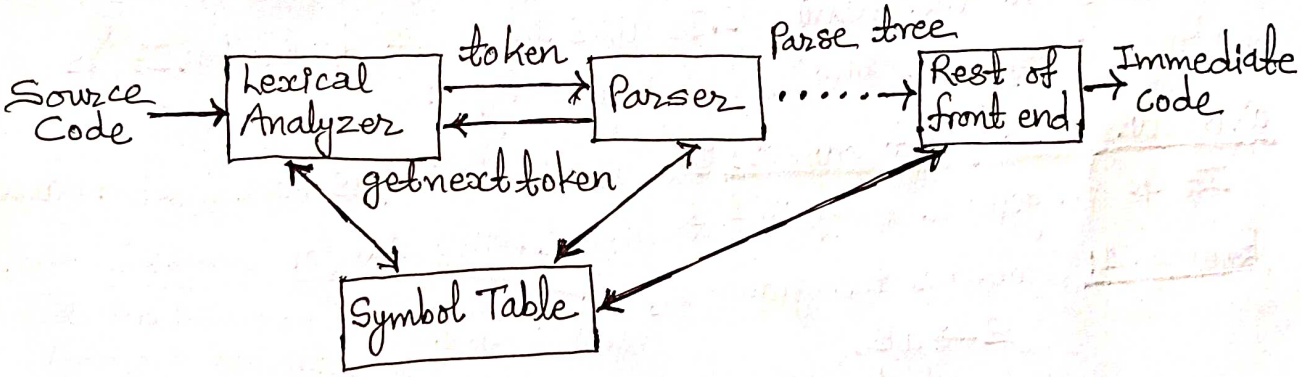


Fig: Role of Parser in a compiler model.

The tasks of parser can be expressed as;
- → Analyzes the context free syntax.
- → Generates the parse tree.
- → Provides the mechanism for context sensitive analysis.
- → Determine the errors and tries to handle them.

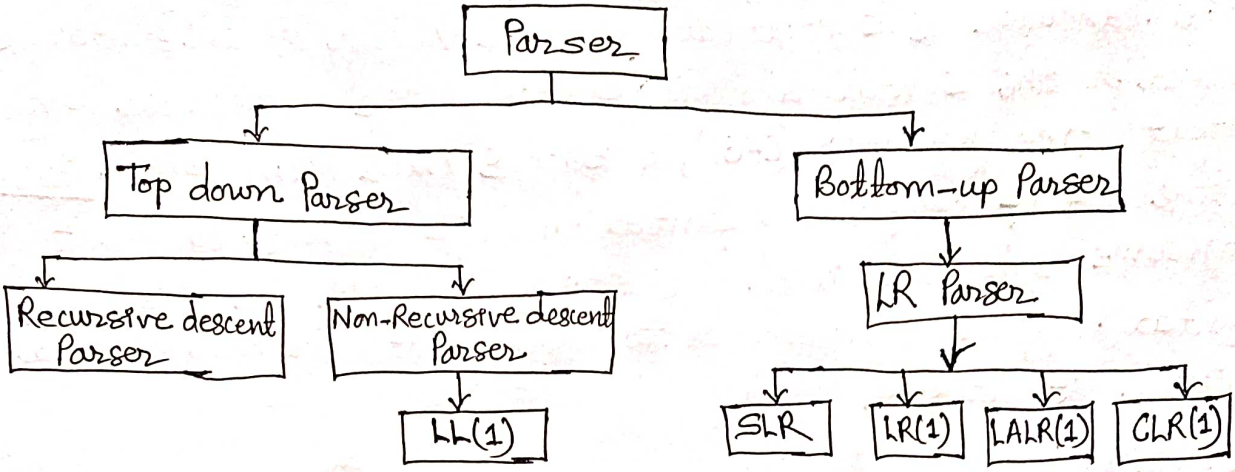
Basic Topics to understand from TOC Unit-4 or Kec Book:

1. Concept of context free grammar including production rules of language.
2. Derivations (leftmost and rightmost)
3. Parse Trees
4. Ambiguity of a Grammar.

These topics are not asked in exam maybe rarely Parse tree asked but concept is needed for upcomming important topic Parsing.

theory related to this are less imp mostly Numericals are imp.

Parsing: Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase. A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.



1) Top down Parser:

Top-down parser is a parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e, it starts from the start symbol and ends on the terminals. It uses left most derivation. It is further classified into following two types:

a) Recursive descent parser: It is also known as backtracking parser. It is a general inefficient technique, only used for small production rules.

Example 1: Consider the grammar,

$S \rightarrow aBc$

$B \rightarrow bc/b$

Input string: abc

Solution:

for solving this every time we use three columns input, output & Rule. सुरुमा input मा दिइएको input string राख्ने output मा production rule मा starting symbol जुन छ त्यो राख्ने / र rule मा starting symbol मा भएको production rule राख्ने

Input	Output	Rule used
abc	S	use $S \rightarrow aBc$
abc	aBc	Match symbol a
bc	Bc	use $B \rightarrow bc$
bc	bcc	Match symbol b
c	cc	Match symbol c
$\emptyset$	c	Dead end, backtrack.
bc	Bc	use $B \rightarrow b$
bc	bc	Match symbol b
c	c	Match symbol c
$\emptyset$	$\emptyset$	Accepted

backtrack गरेको

above step मा S→aBc rule use गरेकोले S लाई aBc ले replace गरेको

left to right corresponding symbol match गर्ने match नभए त्यो symbol delete गर्ने there a is deleted

now we have capital B which is non terminal so it should be reduced to terminal using production rule

दुईवटै $\emptyset$ भए accept हुन्छ अन्यथा backtrack गर्छ।

backtrack गर्दा त्यो भन्दा पहिला production rule जो use भाको छ त्यहाँ गर्ने so हामिले 3rd step मा गाछौं

Example2: Consider the grammar,

$S \rightarrow aBc \mid aaB$

$B \rightarrow bc \mid a$

Input string: aaa

Solution:

Input	Output	Rule used
aaa	S	use $S \rightarrow aBc$
aaa	aBc	Match symbol a
aa	Bc	use $B \rightarrow bc$
aa	bcc	Dead end, backtrack
aa	Bc	use $B \rightarrow a$
aa	ac	Match symbol a
a	c	Dead end, backtrack
aa	Bc	Dead end, backtrack
aa	Bc	Dead end, backtrack
aaa	S	use $S \rightarrow aaB$
aaa	aaB	Match symbol a
aa	aB	Match symbol a
a	B	use $B \rightarrow bc$
a	bc	Dead end, backtrack
a	B	use $B \rightarrow a$
a	a	Match symbol a
ϕ	ϕ	Accepted

① सुरुको दुईवटै symbol terminal (i.e, small) हुँदा र match भएन भने dead end हुन्छ recently used rule मा backtrack हुन्छ

② back track गरेर input र output copy गर्यो recently used production rule को

③ B replaced by a by rule and symbol a matched so will get deleted in next step.

④ dead end आएर recent production rule $B \rightarrow a$ मा गयो जसमा input र output aa र Bc हुन् हामिले B लाई reduce गर्नु पर्ने terminal symbol (i.e, small) मा but $B \rightarrow bc$ र $B \rightarrow a$ already check गरिसक्यौं sab आएन अरु rule छैन B gives so फेरि backtrack गरि माथिल्लो production $S \rightarrow aBc$ मा गयौं,

⑤ $S \rightarrow aBc$ already check गरेको but sab आएन so अब aaB राखेर whole process गर्दा

⊛. Left Recursion:

A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left recursion becomes problem for top-down parsers because left recursion leads to infinite loop so we reduce or eliminate it before solving top-down parsers. A grammar is left recursive if it has a non-terminal A such that there is a derivation:

$$A \rightarrow A\alpha \quad \text{for some string } \alpha.$$

Let we has production as follows:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$$

where, $\beta_1, \beta_2, \ldots \beta_n$ do not start with A

Now we eliminate immediate left recursion as;

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

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

$A\alpha_1$ to $A\alpha_n$ left recursion occurs & β_1 to β_n does not occur

β means it can be terminal or non-terminal symbol

यो general form हो left recursion eliminate गर्ने

माथिबाट left recursion नभएको Part A' पछाडि लिखेको

माथिबाट left recursion symbol (i.e A) हटाई बाकी A' सँग लिखेको last मा ε आउँछ

Example1: Eliminate left recursion from following grammar,

$S \to SBc|SaB|ab|ba|a|b$

$B \to bc|a$

(→ SBc|SaB left recursion part because S→Sα form i.e. A→Aα)

Solution:

(left recursion भएकोको part र S')

$S \to abS'|baS'|aS'|bS'$

$S' \to BcS'|aBS'|\varepsilon$

$B \to bc|a$

(left recursion भएको part with S' recursion symbol हटाएर last मा ε राखेर as in general form)

(यसमा left recursion छैन so copy as it is)

Example2: Eliminate left recursion from following grammar,

$E \to E+T|T$

$T \to T*F|F$

$F \to id|(E)$

Solution:

(Recursive E+T मा E छ so, E skip+T taken and E' added with ε as in general form)

$E \to TE'$

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

$T \to FT'$

(non-recursive part with T')

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

$F \to id|(E)$

Non-Immediate left recursion:

Example: Let's take a grammar with non-immediate left recursion.

$S \to Aa|b$

$A \to Sc|d$

(directly left recursion भएको but कुनै अरु अरु production राखेर check गर्दा left recursion आउने, यस्तो लाई non-immediate left recursion भनिन्द। यसलाई solve गर्न कुनै अरु production rule राखेर पहिला left recursion बनाउने अनि माथि को जस्ते solve गर्ने।)

This grammar is not immediately left-recursive but is still left recursive, So, at first we make immediate recursive as,

$S \to Sca|da|b$

$A \to Sc|d$

(A मा स्क्रोटि Sc राख्दा अर्को चोटि d since A→Sc|d)

Now, eliminate left recursion as,

$S \to daS'|bS'$

$S' \to caS'|\varepsilon$

$A \to Sc|d$

(production को right side rule हरु same symbol बाट start भएको हुने (more than 1 rule) then, it is called left-factorial grammar)

⊛. Left-factoring: If more than one production rules of a grammar have a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string. This grammar is called left factoring grammar.

Let's take an example;

$$A \to \alpha\beta_1|\alpha\beta_2|\alpha\beta_3|\ldots|\alpha\beta_n|\gamma$$

Now eliminate left factoring as;

(General form)

$$A \to \alpha(\beta_1|\beta_2|\beta_3|\ldots|\beta_n)|\gamma$$

(repeat हुने start symbol common हटाएर left-factoring गरिन्द)

The resulting grammar with left factoring free is;

$A \to \alpha A'|\gamma$

$A' \to \beta_1|\beta_2|\beta_3|\ldots|\beta_n$

Example1: Eliminate left factorial from following grammar:

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

$B \rightarrow b$

Solution:

यहाँ maximum मात्र common लिएर फेरि E ले left factorial हुन्छ/so common सबै लिने/so, iEtS पुरै लिदा left factorial रहँदैन

$S \rightarrow iEtS(\varepsilon \mid eS) \mid a$

$B \rightarrow b$

The resulting grammar with left factorial free is,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

$B \rightarrow b$

Example 2: Eliminate left factorial from following grammar:

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

Solution:

$S \rightarrow bSS' \mid a$ ← सबै maximum मा हेर्दा bS तिनवटा मा common, so bS common लिएको

$S' \rightarrow SaaS \mid SaSb \mid b$ ← Sa फेरि common हुन so, फेरि लगाउने

The resulting grammar with left factorial free is,

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaS'' \mid b$

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

b) Non-Recursive Descent Parser:

A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. It is also called LL(1) parsing table technique since we would be building a table for string to be parsed. It has capability to predict which production is to be used to replace input string.

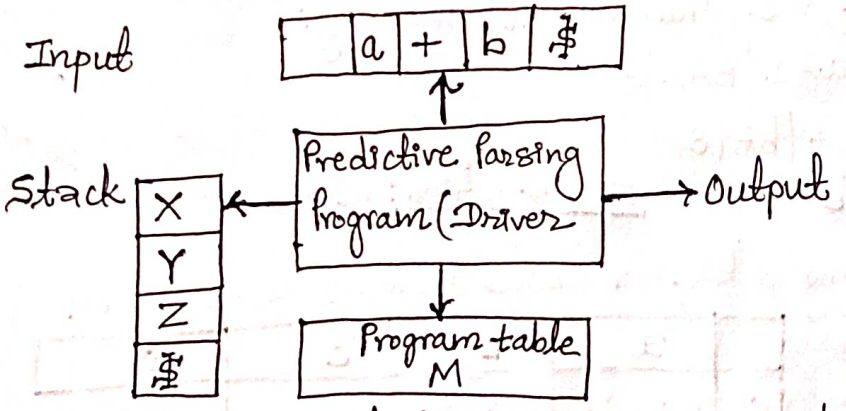


Fig: Model of a non-recursive predictive parser.

- **Input Buffer:** It contains the string to be parsed followed by a symbol $.
- **Stack:** It contains sequence of grammar symbols with $ at bottom.

- **Parsing table:** It is a two dimensional array M[A, a] where 'A' is non-terminal and 'a' is terminal symbol.
- **Output stream:** A production rule representing a step of the derivation sequence of the string in the input buffer.

⊛ Recursive predictive descent parser VS. Non-recursive predictive descent parser:

Recursive predictive descent parser	Non-recursive predictive descent parser
It is a technique which may require backtracking process.	It is a technique that does not require any kind of backtracking.
It uses procedures for every non terminal entity to parse strings.	It finds out productions to use by replacing input string.
It is a type of top-down parsing built from a set of mutually recursive procedures where each procedure implements one of non-terminals of grammars.	It is a type of top-down approach, which is also a type of recursive parsing that does not uses technique of backtracking.
It contains several small functions one for each non-terminals in grammar	The predictive parser uses a look ahead pointer which points to next input symbols to make it parser back tracking free, predictive parser puts some constraints on grammar.
It accepts all kinds of grammars.	It accepts only a class of grammar known as LL(k) grammar.

Example: (To demonstrate Non-recursive descent parser):

Consider the grammar G1 given by:

$S \rightarrow aAa \mid BAa \mid \varepsilon$

$A \rightarrow cA \mid bA \mid \varepsilon$

$B \rightarrow b$

Input string: bcba

Solution:

Parsing table for above grammar is as follows:

	a	b	c	$
S	S→aAa	S→BAa		S→ε
A	A→ε	A→bA	A→cA	
B		B→b		

Parsing Table बनाने का topic LL(1) में construct करने के लिए consider as we know to draw this table or this table is given in question

Now do not focus on how this table is constructed just use this table

Now parsing of input string w=bcba using non-recersive descent parser is as follows:

Stack	Remaining input	Action
\$S	bcba\$	choose S→BAa
\$aAB	bcba\$	choose B→b
\$aAb	bcba\$	match b
\$aA	cba\$	choose A→cA
\$aAc	cba\$	match c
\$aA	ba\$	choose A→bA
\$aAb	ba\$	match b
\$aA	a\$	choose A→ε
\$a	a\$	match a
\$	\$	Accept

① stack को सुरुमा \$ symbol then start symbol of grammar लेख्नुहोस्।

② input को सुरुमा input string र last मा \$ symbol लेख्नुहोस्।

③ मा दिएको table मा stack को top symbol र input को first symbol हुँदा जुन rule बन्छ त्यहि लेख्नुहोस्। Here we have stack top=S, 1^{st} input=b, So, S & b in table gives BAa.

④ We have S in stack top but we choose S→BAa So, S is replaced by BAa but as we know stack is Last in first out (LIFO) so we have to store it in reverse order as aAB.

Now top of stack contains B and start of input contains b so according to table rule is B→b So we choose B→b

⑤ Now top of stack contains b and first input also b so both matched and b is deleted

⑥ Similarly we proceed and finally if we get both stack and input as \$ symbol only then we accept string otherwise reject

Stack मा last character हुँदा input मा first character हटाउनुहोस्।

⊛ Constructing LL(1) Parsing Table: [V. Imp]

The parse table construction requires two functions: FIRST and FOLLOW.
A grammar G is suitable for LL(1) parsing table if the grammar is free from left recursion and left factoring.

[A grammar] —eliminate left recursion→ —eliminate left factor→ [Grammar is suitable now for predictive parsing (LL(1) grammar)]

To compute LL(1) parsing table, at first we need to compute FIRST and FOLLOW functions.

Compute FIRST: FIRST(α) is a set of terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.

Rules:

↳ If 'a' is terminal, then FIRST(a) is {a}.

↳ If $A \to \varepsilon$ is a production, then FIRST(A) is ε.

↳ For any non-terminal A with production rules $A \to \alpha_1 | \alpha_2 | \alpha_3 | \dots | \alpha_n$ then $FIRST(A) = FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \dots\dots \cup FIRST(\alpha_n)$.

↳ If the production rule of the form, $A \to \beta_1 \beta_2 \beta_3 \dots \beta_n$ then,

$$FIRST(A) = FIRST(\beta_1 \beta_2 \beta_3 \dots \beta_n).$$

Example1: Find FIRST of following grammar symbols,

$R \to aS | (R)S$

$S \to +RS | aRS | *S | \varepsilon$

Solution:

We calculate FIRST for each symbol on production

terminal symbol so first of that symbol is same that symbol by rule

opening symbol first comes and these symbols are treated as terminal

$FIRST(aS) = FIRST(a) = \{a\}$ $\quad FIRST((R)S) = FIRST(() = \{(\}$

$FIRST(+RS) = FIRST(+) = \{+\}$ $\quad FIRST(aRS) = FIRST(a) = \{a\}.$

$FIRST(*S) = FIRST(*) = \{*\}$ $\quad FIRST(\varepsilon) = FIRST(\varepsilon) = \{\varepsilon\}.$

$FIRST(R) = \{FIRST(aS) \cup FIRST((R)S)\} = \{a, (\}$

first of aS and (R)S already found so now we can just place them.

$FIRST(S) = \{FIRST(+RS) \cup FIRST(aRS) \cup FIRST(*S) \cup FIRST(\varepsilon)\}$

$= \{+, a, *, \varepsilon\}.$

Example2: Find FIRST of following grammar symbols,

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$

Solution:

opening brace

$FIRST(F) = \{FIRST((E)) \cup FIRST(id)\} = \{(, id\}$ $\quad FIRST(id) = \{id\}$

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

$FIRST(T') = \{*, \varepsilon\}$

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

$FIRST(TE') = FIRST(T) = \{(, id\}$ $\quad FIRST(T) = FIRST(F) = \{(, id\}$

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

$FIRST(\varepsilon) = \{\varepsilon\}.$

$FIRST(+TE') = FIRST(+) = \{+\}$

$FIRST(*FT') = FIRST(*) = \{*\}$

$FIRST(FT') = FIRST(F) = \{(, id\}$

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

Compute FOLLOW: FOLLOW(A) is the set of terminals that can immediately follow non terminal A except ε.

Rules:

↳ If A is a starting symbol of given grammar then FOLLOW(A) = {$}.

↳ For every production $B \to \alpha A \beta$, where α and β are any string of grammar symbols and A is non terminal, then everything in FIRST(β) except ε is FOLLOW(A).

↳ For every production $B \to \alpha A$, or a production $B \to \alpha A \beta$, FIRST(β) contains ε, then everything in FOLLOW(B) is FOLLOW(A).

Example 1: Compute FOLLOW of the following grammar.

$R \to aS \mid (R)S$

$S \to +RS \mid aRS \mid *S \mid \varepsilon$

Follow compute गर्दा left side को symbol को मात्र लिए पुग्छ

Solution:

follow find गर्दा जसको follow find गर्दैछौं, right side मा त्यो letter जसको production मात्र लिन्छे। Here R को निकाल्दा $R \to (R)S$, $S \to +RS, S \to aRS$ मात्र लिएको

$FOLLOW(R) = \{FOLLOW(R)S \cup FOLLOW(+RS) \cup FOLLOW(aRS)\}$

according to 2nd rule

$= \{FIRST(\,)S \cup FIRST(S) \cup FOLLOW(S)\}$

We have aRS but $S \to \varepsilon$ so if we put ε in place of S we get aR. So, rule 3rd used

$= \{\$,), +, a, *\}$

R starting symbol भएकोले $ थपिएको

FIRST( )S is)

Since we don't know follow of S now so we calculate it below separately

FIRST(S) is already computed in FIRST section example 1 so, we directly use here FIRST(S) = {+, a, *, ε}

aRS मा ε नभएको भए 2nd rule use हुन्थ्यो FIRST(S) हुन्थ्यो but FIRST(S) set मा already आइसकेको थियो so repeat हुने थिएन FIRST( )S ∪ FIRST(S) यो दुई term मात्र हुन्थ्यो।

we do not use ε in FOLLOW

$FOLLOW(S) = \{FOLLOW(R)\}$

$= \{\$,), +, a, *\}$

Follow(S) same as follow(R) आयो so माथि थपेनैं किनकि set मा value repeat हुँदैनन। फरक आको भए माथि FOLLOW(S) को ठाउँमा value हरु थपथ्यौं।

Example 2: Compute FOLLOW of the following grammar.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

left side symbols E, E', T, T', र F को Follow निकाल्दै

right side E भएको production हरुमा मात्र ध्यान दिने so त्यो मात्र लिएको $F \to (E)$

Solution:

$FOLLOW(E) = FIRST(\,)$

E लाई follow गर्ने closing brace which is terminal. Terminal को First terminal नै हुन्छ।

$= \{\$,)\}$

E starting symbol पनि हो so start मा $ add गर्दै

We have two productions for this $E \to TE'$ and $E' \to +TE'$

E' को follow निकाल्दै छैन, $E' \to \varepsilon$ छ but E' मै राख्न मिलेन i.e, निकाल्ने symbol र ε राख्ने symbol same नाए ε राख्दैनौं।

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

$E \to TE'$ मा E' पछि केहि छैन so, FOLLOW(E) similar for E'

$= \{\$,)\}$

follow(E) को value लिएको मात्रिक्वाट। FOLLOW of E' निकाल्दैछौं so Follow(E') को निकाल्दैनौं आफ्नै follow भएर

$FOLLOW(T) = \{FOLLOW(TE') \cup FOLLOW(+TE') \cup FOLLOW(T) \cup FOLLOW(E')\}$

we have following productions for this.
$E \to TE'$, $E' \to +TE'$
But we also have $E' \to \varepsilon$
so putting in above two we get two new productions as
$E \to T$, $E' \to +T$

$E \to TE'$
$E' \to +TE'$
gives FIRST(E') by 2nd rule

$= \{FIRST(E') \cup FOLLOW(E) \cup FOLLOW(E')\}$

$= \{+, \$,)\}$

first of E' is + already calculated in FIRST Section

FOLLOW(E) मा $ र) already calculated above, FOLLOW(E') मा $ already in set so not necessary to include

$E \to T$ gives FOLLOW(E) by 3rd rule

by 3rd rule $E' \to +T$ gives FOLLOW(E')

$FOLLOW(T') = \{FOLLOW(FT') \cup FOLLOW(*FT')\}$

we have two productions for this: $T \to FT'$ and $T' \to *FT'$
Also we have $T' \to \varepsilon$ but T' के निकाल्दिदैन follow so ε राख्नु पर्दैन as we did for E' above

$= \{FOLLOW(T) \cup FOLLOW(T')\}$

by 3rd rule

by 3rd rule. but we can neglect this, we are calculating Follow(T) itself so.

$= \{+,), \$\}$

value of FOLLOW(T) that we already calculated above

$FOLLOW(F) = \{FOLLOW(FT') \cup FOLLOW(*FT') \cup FOLLOW(F) \cup FOLLOW(*F)\}$

$= \{FIRST(T') \cup FOLLOW(T')\}$

by 2nd rule

$= \{+, *,), \$\}$

We have two production for FOLLOW(F) as;
$T \to FT'$ and $T' \to *FT'$
but we also have $T' \to \varepsilon$ in grammar so, we put and get two more:
$T \to F$ and $T' \to *F$

Note: LL(1) Parsing table construct को लागि FIRST र FOLLOW find गर्दा At first left recursion र left factorial छ कि छैन check गर्ने। अनि directly FIRST र FOLLOW निकाल्ने। भए पहिला left recursion र left factorial reduce गर्ने (that we already read) यसपछि FIRST र FOLLOW को लागि same process proceed गर्ने,

FOLLOW निकाल्दा कुनै symbol को, त्यो symbol production को right side मा कुनै ठाउँमा पनि नभेटिए त्यसको follow start symbol भए $ मात्र हुन्छ अरु भए केहि पनि हुँदैन empty.

We have calculated FIRST and FOLLOW, now we can construct LL(1) parsing table easily as in the following examples:

Example1: Construct LL(1) parsing table of following grammar.

$R \to aS \mid (R)S$

$S \to +RS \mid aRS \mid *S \mid \varepsilon$

Solution:

Non-terminals	Terminal Symbols					
	a	(	)	+	*	$
R	R→aS	R→(R)S				
S	S→aRS S→ε		S→ε	S→+RS, S→ε	S→*S, S→ε	S→ε

① माथिको production हरु हेर्ने non-terminal जति सबै यो side लेख्ने

② माथिको production or grammar हेर्ने terminal हरु सबै लेख्ने र $ पनि आफ्नो last मा

③ Now fill गर्न सुरु गर्‍यौं, so R ले दिने production हेर्‍यौं माथिको grammar मा, $R \to aS$ र $R \to (R)S$ हुन्। Production मा दुई case आउन सक्दछन् एउटा ε भनेको आर्को ε नभएको, ε नभएको आर (i.e, $A \to \alpha$ form where α can be terminals and non-terminals 1 or more than one) त्यतिबेला FIRST(α) हेर्ने हुन्। For eg. $R \to aS$ मा ε नभएको case हुन, so FIRST(aS) हेर्‍यौं जुन हामिले पहिले नै निकालेका थियौं जसको value {a} हुन so $R \to aS$, 'R' row and 'a' column मा हुने भयो। Same process for $R \to (R)S$, $S \to +RS$, $S \to aRS$ and $S \to *S$.

ε भएको case हुन भने, (i.e, $A \to \varepsilon$) त्यतिबेला FOLLOW(A) हेर्ने हुन्। For eg. $S \to \varepsilon$ हुन last production हाम्रो so we see FOLLOW(S) which we already calculated and it has value {$,), +, a, *} हुन, so S row अनि यी सबै symbol भएको तल $S \to \varepsilon$ production हुने भयो।

④ हामिले LL(1) parsing table त बनायौं but here एउटै cell मा more than 1 production पनि हुन so in this type of case, the given grammar is not feasible for LL(1) parser. एउटा cell मा एउटा मात्र production भएको बेला LL(1) parser को लागि feasible हुन्छ।

Example2: Construct LL(1) parsing table for following grammar.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

Solution:

Non-terminals	Terminal Symbols					
	+	*	(	)	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→(E)		F→id	

2. Bottom-up Parser:

→ यो topic बाट पनि कतिमा दुई question सोध्न सकिन्छ 3 पनि सोध्न / यो topic मा Shift-reduce parsing, SLR, LR(1), LALR(1) पढ्नुहोला

Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by starting from input string and ends on the start symbol, by the reduction process. Reduction is the process of replacing a substring by a non-terminal in bottom-up parsing. It is the reverse process of production.

Example: Consider the grammar

S→aABe
A→Abc|b
B→d

→ Production मा left side को symbol लाई right side को symbol ले replace गरिन्छ / but Reduction is just opposite. rightside को symbol लाई left side को symbol ले replace गरिन्छ

Now, the sentence abbcde can be reduced to S as follows:

abbcde
aAbcde (replacing b by using A→b)
aAde (replacing Abc by using A→Abc)
aABe (replacing d by using B→d)
S (replacing aABe by using S→aABe)

Hence, S is the starting symbol of grammar.

(short note मा सोध्न सक्छ यो)
(basic terminology हो bottom-up parsing मा use हुन)
(right side को terminal symbol जुन non-terminal symbol मा reduce हुन्छ, त्यो symbol लाई handle भनिन्छ।)

⊛ Handle:

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a handle. If the grammar is unambigous, then every right-sentential form of the grammar has exactly one handle.

Example: Let we have production as: $E \rightarrow E+T \mid T$, Now if we reduce E+T to E then, E+T is handle. Similarly let we have production as; $F \rightarrow (E) \mid id$, Now if we reduce id to F then, id is handle.

a) Shift-Reduce Parsing:

A shift reduce parser tries to reduce the given input string into the starting symbol. At each reduction step, a substring of the input matching to the right side of a production rule is replaced by non-terminal at the left side of that production rule. The process of reducing the given input string into the starting symbol is called shift-reduce parsing.

A string $\xrightarrow{\text{Reduced to}}$ the starting symbol.

Stack Implementation of Shift-Reduce Parser:

(यो theory हरु understand गर्नु मात्र exam मा numerical आउँछ can be escaped)

There are mainly following four basic operations used in shift reduce parser:

Shift: This involves moving of symbols from input buffer onto the stack.

Reduce: If the handle appears on top of the stack then, it's reduction by using appropriate production rule is done i.e, RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

Accept: If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. Accept means successful parsing is done.

Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Algorithm:

1. Initially stack contains only the sentinel \$, and input buffer contains the input string w\$.

2. While stack not equal to $S do
   a. While there is no handle at the top of the stack, do shift input buffer and push the symbol onto the stack.
   b. If there is a handle on the top of the stack, then pop the handle and reduce the handle with its non-terminal and push it onto stack.
3. Done.

Note that S→E and S→(E) are different not same

Example 1: Use the following grammar

$$S \rightarrow S+S \mid S*S \mid (S) \mid id$$

⇒ production rule को right side हिर्दा जे जे हुन् त्यो सबै handle हुन् like S+S is one, S*S is another, (S) and id are also handle.

Perform Shift Reduce parsing for input string: id+id+id

Solution:

↳ सुरुमा stack empty हुन्छ $ मात्र हुन्छ ↔
↳ Input buffer मा input string राख्छौं last मा $ symbol थपेर

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift id
$id	+id+id$	Reduce by S→id
$S	+id+id$	Shift +
$S+	id+id$	Shift id
$S+id	+id$	Reduce by S→id
$S+S	+id$	Shift +
$S+S+	id$	Shift id
$S+S+id	$	Reduce by S→id
$S+S+S	$	Reduce by S→S+S
$S+S	$	Reduce by S→S+S
$S	$	Accept

Parsing action मा algorithm अनुसार mostly shift र Reduce operation हुन्छ।

Handle देखे stack मा र input buffer empty भयो भने त्यतिबेला ERROR हो त्यो ERROR लेखेर त्यहिं stop गर्ने।

Stack को top मा handle देखे भने shift गर्ने handle भए Reduce गर्ने

shift/reduce conflict
↳ यो case मा S handle होइन but S+S handle हो so यो फिर reduce गर्दा नि हुन्छ S→S+S, फरक पर्दैन यो ques मा

shift गर्दा input buffer मा भएको first symbol stack को top मा move गर्ने

Reduce गर्दा terminal symbol लाई non-terminal ले replace गर्ने rule अनुसार

accept हुने stack मा $ र start symbol मात्र भएर पर्दा र input buffer मा $ symbol मात्र

Example 2: Use the grammar, and perform shift reduce parsing.

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

Input string: id+id*id

Solution:

Stack	Input Buffer	Parsing Action.
$	id+id*id$	Shift id
$id	+id*id$	Reduce by F→id
$F	+id*id$	Reduce by T→F
$T	+id*id$	Reduce by E→T
$E	+id*id$	Shift +
$E+	id*id$	Shift id
$E+id	*id$	Reduce by F→id
$E+F	*id$	Reduce by T→F
$E+T	*id$	Shift * (OR Reduce by E→T) CONFLICT
$E+T*	id$	Shift id
$E+T*id	$	Reduce by F→id
$E+T*F	$	Reduce by T→T*F
$E+T	$	Reduce by E→E+T
$E	$	Accept

we find possible ways to accept string in this case

T alone is handle whole E+T is also handle so, this is case of reduce/reduce conflict

we have production E→T so T is handle but if we reduce T by E it will be E+E but later E+E can't be reduced which is not in production and may lead to error so we did shift operation in this type of conflict case.

Practice more questions from text book

⊛. Conflicts in Shift-Reduce Parsing:

There are two kinds of shift-reduce conflicts:

exam मा आए example पनि गरेर देखाउने

1) **Shift/Reduce Conflict:** Here, the parser is not able to decide whether to shift or to reduce. (like in 6th step of example 1)

2) **Reduce/Reduce Conflict:** Here, the parser cannot decide which sentential form to use for reduction. (like in 9th step of example 2).

b) LR Parser:

shift reduce मा shift/reduce र reduce/reduce conflict आउँदा यो method use गर्दा conflict आउँदैन।

LR parsing is one type of bottom up parsing. It is used to parse large class of grammars. In the LR parsing, L stands for left-to-right scanning of the input. R stands for constructing a right most derivation in reverse. We will study SLR, LR(1) and LALR(1) here in this section.

LR Parsers: General Structure

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output, and stack are same but parsing table is different.

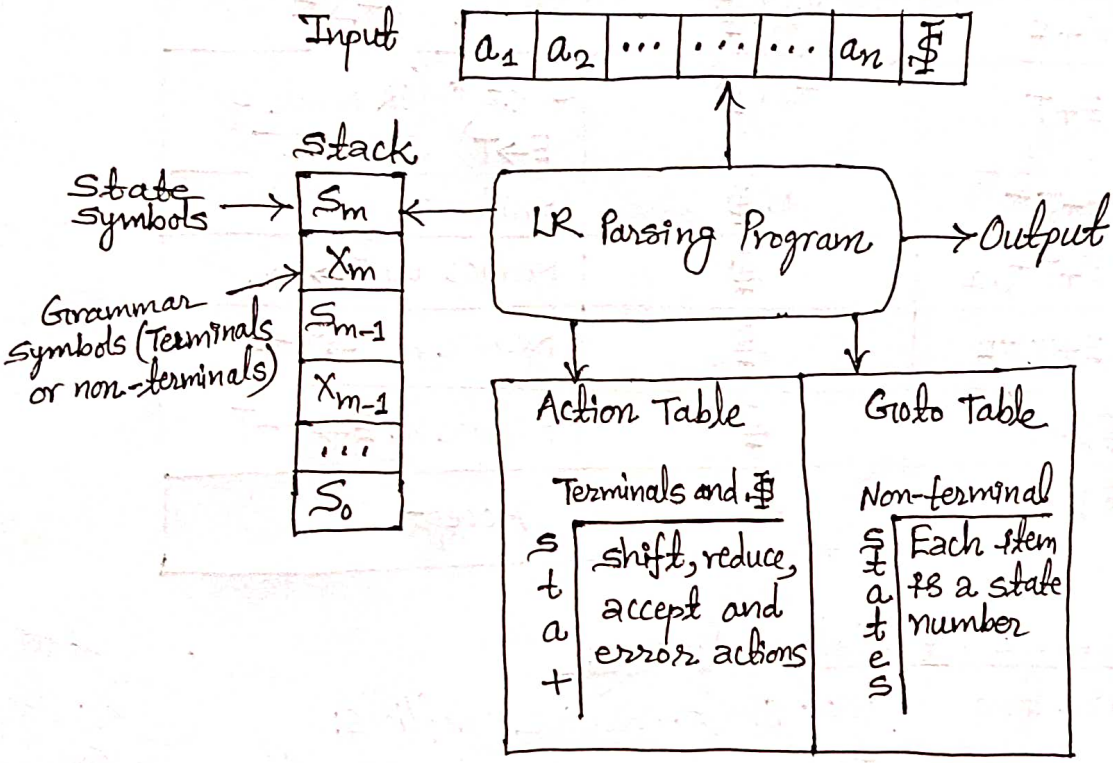


Fig: Block diagram of LR parser.

यो basic terminologies short note मा 5 marks को आउँछ। example।

a) SLR (Simple LR Parser):

Basic terminologies used for LR parsing table:

Augmented grammar: If G is a grammar with start symbol S, then the augmented grammar G' of G is a grammar with a new start symbol S' and production S' → S.

Example: the grammar,

E → E+T | T
T → T*F | F
F → (E) | id

same

It's augmented grammar is,

E' → E
E → E+T | T
T → T*F | F
F → (E) | id

starting symbol को dash gives starting symbol given grammar को सुरुमा भएमा त्यसलाई augmented grammar भनिन्छ।

↳ **LR(0) Item:** An 'item' (LR(0) item) is a production rule that contains a dot (•) somewhere in the right side of the production. For example, the production $A \to \alpha A \beta$ has four items:

$A \to \bullet \alpha A \beta$
$A \to \alpha \bullet A \beta$
$A \to \alpha A \bullet \beta$
$A \to \alpha A \beta \bullet$

↔ कुनै production को right side मा कुन कुन ठाउँमा • राख्न सकिन्छ त्यो सबै possible outcome लाई LR(0) item भनिन्छ। • सबै ठाउँमा राख्न मिल्छ terminal, non-terminal symbol को मतलब नराख

A production $A \to \varepsilon$, generates only one item $A \to \bullet$.

↳ **Closure Operation:** If I is a set of items for a grammar G, then closure(I) is the set of LR(0) items constructed from I using the following rules:

→ Symbol like B which can be terminal or non-terminal

1. Initially, every LR(0) item in I is added to closure(I).
2. If $A \to \alpha \bullet B\beta$ is in closure(I) and $B \to \gamma$ is a production rule of G then add $B \to \bullet\gamma$ in the closure(I) repeat until no more new LR(0) items added to closure(I).

Example: Consider the grammar:

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

It's augmented grammar is;

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

If $I = \{[E' \to \bullet E]\}$, then closure I contains the items,

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

↔ closure निकाल्दा • को पछाडि कुन symbol छ त्यसैले लिन सक्ने OR त्यो बाट हामि कुन कुन production मा पुग्न सक्छौं, सुरुमा • लेखेर त्यसै सबै production लेख्दौं। Foreg we have $E' \to \bullet E$ initially. We have E after • so we write productions of E using dot at start as $\bullet E+T$ and $\bullet T$ again we have T which can produce again and so on.

↔ अरु पनि लिन सक्छौं but example को लागि $E' \to E$ लिएको

↳ **Goto Operation:**

If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I, X) is defined as follows:

If $A \to \alpha \bullet X\beta$ in I then every item in closure $(\{A \to aX \bullet \beta\})$ will be in goto(I, X).

↔ goto निकाल्दा सुरुमा • symbol, 1 step पछाडि सार्ने then त्यो • पछिको symbol को साथि जस्ते गरि closure operation गर्ने त्यहि हो।

Example:

Let, $I = \{E' \to \cdot E, E \to \cdot E+T, E \to \cdot T, T \to \cdot T*F, T \to \cdot F, F \to \cdot(E), F \to \cdot id\}$ ← Set of LR(0) item,

Now,

$goto(I, E) = closure(\{[E' \to E^\cdot, E \to E^\cdot + T]\})$

$= \{E' \to E^\cdot, E \to E^\cdot + T\}$

Now we are finding goto of E, so • E produce गर्ने term हरु मात्र लिएर त्यो set को goto गर्दा। goto means shift • one step right then do closure

E' → E• मा • पछि कुनै term छैन so no closure or no other new production occur. E → E•+T मा • पछि + symbol छ so, terminal symbol को closure हुन्छ non-terminal को मात्र so same term as closure of that

$goto(I, T) = closure(\{[E \to T^\cdot, T \to T^\cdot * F]\})$

•T पछिका term मात्र हेर्ने

$= \{E \to T^\cdot, T \to T^\cdot * F\}$

$goto(I, F) = closure(\{[T \to F^\cdot]\})$

$= \{T \to F^\cdot\}$

$goto(I, ()) = closure(\{[F \to (\cdot E)]\})$

• पछि E छ so E को closure निकाल्ने निकालिछ, means E बाट जुन जुन production पुग्न सक्दो त्यो सबै नुरुमा • लगेर

$= \{F \to (\cdot E), E \to \cdot E+T, E \to \cdot T, T \to \cdot T*F, T \to \cdot F, F \to \cdot(E), F \to \cdot id\}$

• पछि terminal non-terminal जे आए पनि सबको goto निकाल्ने छोड्ने। F → •(E) मा • पछि opening brace आयो so यसको goto

$goto(I, id) = closure(\{[F \to id^\cdot]\})$

$= \{F \to id^\cdot\}$

↳ Canonical LR(0) collection:

To construct canonical LR(0) collection of grammar we require augmented grammar, closure, and goto functions.

यो algorithm solving को process बुझ्नका लागि हो रटनु पर्दैन।

Algorithm:

1. Start
2. Augment the grammar by adding production S' → S.
3. $C = \{closure(\{S' \to S\})\}$
4. Repeat the followings until no more set of LR(0) items can be added to C.

i.e, each LR(0) item को goto निकाल्ने क्रम हो।

for each I in C and each grammar symbol X
if goto(I, X) is not empty and not in C
add goto(I, X) to C.

5. Repeat step 4 until no new sets of items are added to C.
6. Stop.

Example 1: Find canonical collection of LR(0) items of following grammar.

$C \to AB$
$A \to a$
$B \to a$

Solution:

The augmented grammar of given grammar is;

$C' \to C$
$C \to AB$
$A \to a$
$B \to a$

Next, We obtain the canonical collection of sets of LR(0) items as follows;

$I_0 = closure(\{C' \to \cdot C\}) = \{C' \to \cdot C, C \to \cdot AB, A \to \cdot a\}$

$I_1 = goto(I_0, C) = closure(C' \to C\cdot) = \{C' \to C\cdot\}$

$I_2 = goto(I_0, A) = closure(C \to A\cdot B) = \{C \to A\cdot B, B \to \cdot a\}$

$I_3 = goto(I_0, a) = closure(A \to a\cdot) = \{A \to a\cdot\}$

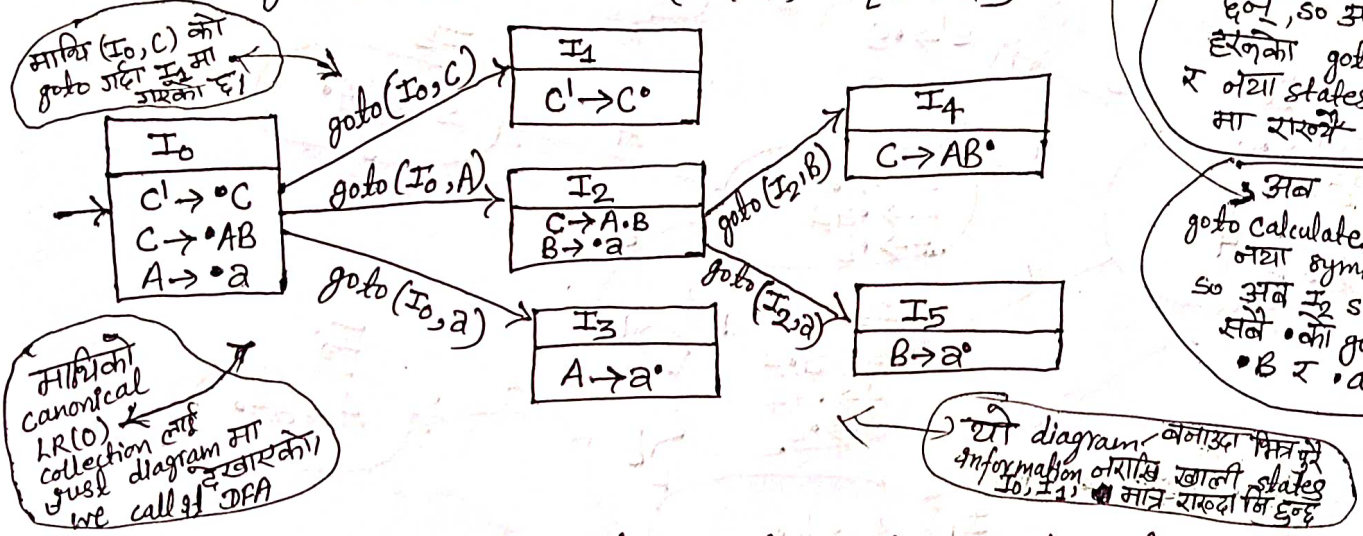
$I_4 = goto(I_2, B) = closure(C \to AB\cdot) = \{C \to AB\cdot\}$

$I_5 = goto(I_2, a) = closure(B \to a\cdot) = \{B \to a\cdot\}$

सुरुको state लाई हामिले I_0 मान्यौं। सुरुको state मा augmented grammar की first production को closure निकाल्दै।

Now, I_0 मा • भाको symbol •C, •A, र •a छन्, so अब यो symbol हरुको goto निकाल्ने र नया states I_1, I_2, I_3 मा राख्ने

अब I_2 मा goto Calculate गर्नुपर्ने नया symbol •B भयो so अब I_2 set मा भएका सबै • को goto निकाल्ने •B र •a को



Example 2: Find canonical collection of LR(0) items of following grammar

$S \to AA$
$A \to aA \mid b$

Solution:

The augmented grammar of given grammar is;

$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

Now we obtain the canonical collection of sets of LR(0) items as follows;

$I_0 = closure(\{S' \to \cdot S\}) = \{S' \to \cdot S, S \to \cdot AA, A \to \cdot aA, A \to \cdot b\}$

$I_1 = goto(\{I_0, S\}) = closure(\{S' \to S\cdot\}) = \{S' \to S\cdot\}$

$I_2 = goto(\{I_0, A\}) = closure(\{S \to A\cdot A\}) = \{S \to A\cdot A, A \to \cdot aA, A \to \cdot b\}$

$I_3 = goto(\{I_0, a\}) = closure(\{A \to a\cdot A\}) = \{A \to a\cdot A, A \to \cdot aA, A \to \cdot b\}$

$I_4 = goto(\{I_0, b\}) = closure(\{A \to b\cdot\}) = \{A \to b\cdot\}$

$I_5 = goto(\{I_2, A\}) = closure(\{S \to AA\cdot\}) = \{S \to AA\cdot\}$

$I_3 = goto(\{I_2, a\}) = closure(\{A \to a\cdot A\}) = \{A \to a\cdot A, A \to \cdot aA, A \to \cdot b\}$

(goto calculate गर्दा first state मा चाहि assign गर्दा because यस्तो case नि आउछ)

(This is same as I_3 state so no need to assign to new state I_6. Now, the transition goto (I_2, a) will go to I_3 as it is same to I_3)

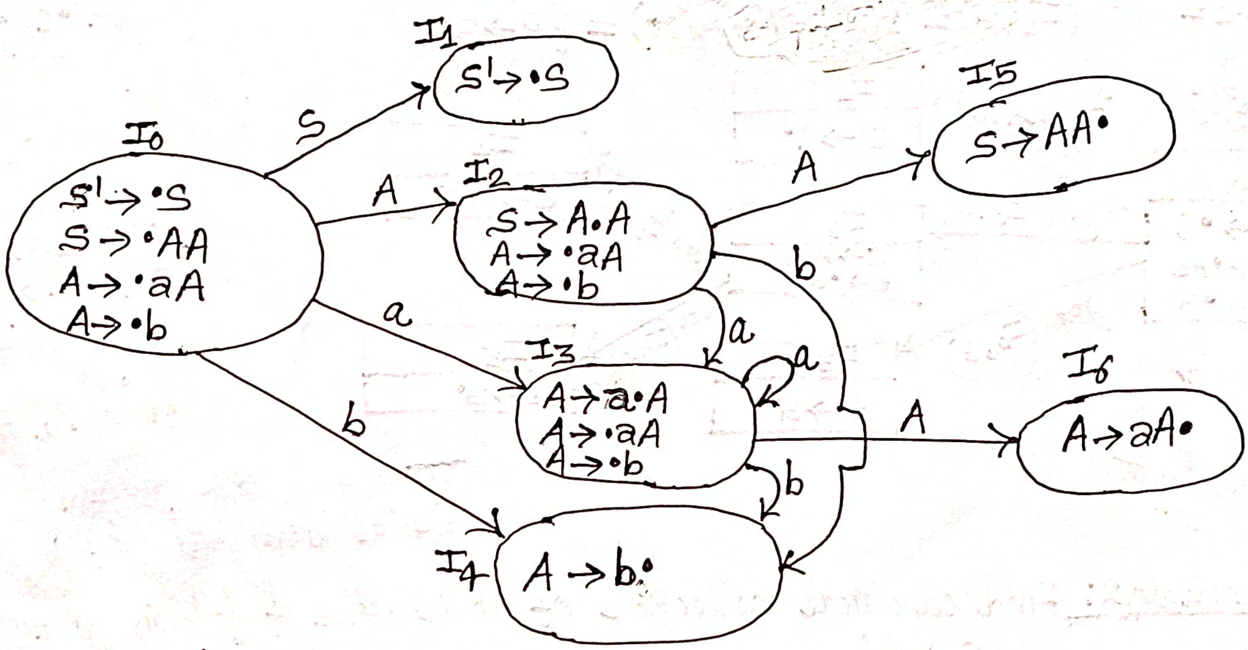
$I_4 = goto(\{I_2, b\}) = closure(\{A \to b\cdot\}) = \{A \to b\cdot\}$ (Same as I_4 so assigned to I_4 instead of new state)

$I_6 = goto(\{I_3, A\}) = closure(\{A \to aA\cdot\}) = \{A \to aA\cdot\}$

$I_3 = goto(\{I_3, a\}) = closure(\{A \to a\cdot A\}) = \{A \to a\cdot A, A \to \cdot aA, A \to \cdot b\}$ (Same as I_3)

$I_4 = goto(\{I_3, b\}) = closure(\{A \to b\cdot\}) = \{A \to b\cdot\}$ (Same as I_4)

Drawing DFA: For DFA of above cannonical LR(0) collection we have 7 states I_0 to I_6 as follows:



⊛ Constructing SLR Parsing Tables:

Algorithm: (just to understand)

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0, I_1, \ldots, I_n\}$
2. Create the parsing action table as follows
   - If $A \to \alpha\cdot a\beta$ is in I_i and $goto(I_i, a) = I_j$ then set action $[i, a]$ = shift j.
   - If $A \to \alpha\cdot$ is in I_i, then set action $[i, a]$ to "reduce $A \to \alpha$" for all 'a' in FOLLOW(A). where, $A \neq S'$.
   - If $S' \to S\cdot$ is in I_i, then action $[i, \$]$ = accept.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table.
   - for all non-terminals A, if goto $(I_i, A) = I_j$ then goto $[i, A] = j$
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow \cdot S$.

Example 1: Construct SLR parsing table of following grammar.

$C \rightarrow AB$
$A \rightarrow a$
$B \rightarrow a$

Solution:

The augmented grammar of given grammar is;

$C' \rightarrow C$
$C \rightarrow AB$
$A \rightarrow a$
$B \rightarrow a$

(these process are same as we already did before in examples of canonical collection of LR(0) items all same)

Next, we obtain the canonical collection of sets of LR(0) items as follows,

$I_0 = \text{closure}(\{C' \rightarrow \cdot C\}) = \{C' \rightarrow \cdot C, C \rightarrow \cdot AB, A \rightarrow \cdot a\}$

$I_1 = \text{goto}(I_0, C) = \text{closure}(C' \rightarrow C\cdot) = \{C' \rightarrow C\cdot\}$

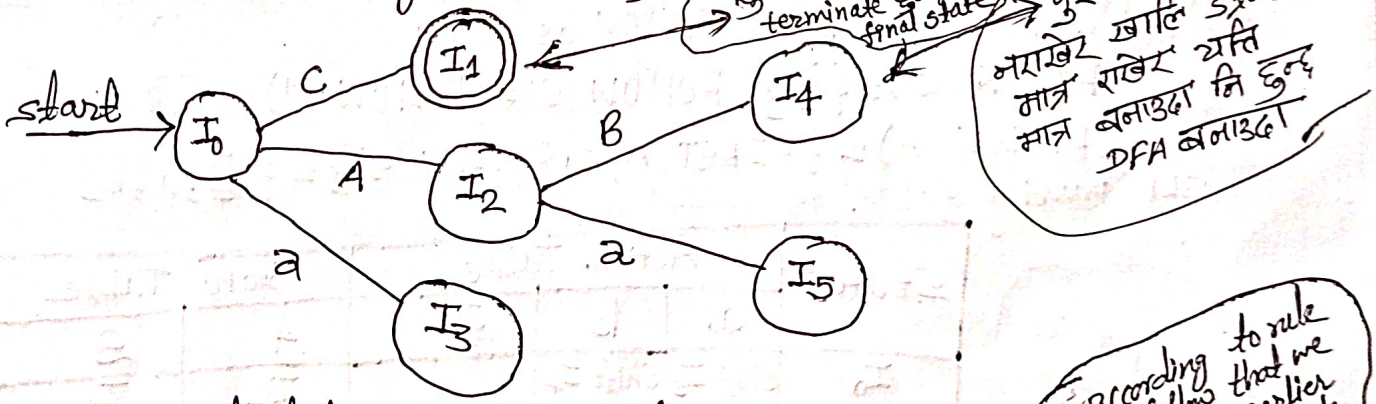
$I_2 = \text{goto}(I_0, A) = \text{closure}(C \rightarrow A \cdot B) = \{C \rightarrow A \cdot B, B \rightarrow \cdot a\}$

$I_3 = \text{goto}(I_0, a) = \text{closure}(A \rightarrow a\cdot) = \{A \rightarrow a\cdot\}$

$I_4 = \text{goto}(I_2, B) = \text{closure}(C \rightarrow AB\cdot) = \{C \rightarrow AB\cdot\}$

$I_5 = \text{goto}(I_2, a) = \text{closure}(B \rightarrow a\cdot) = \{B \rightarrow a\cdot\}$

The DFA of above grammar is,



Now we calculate FOLLOW function as;

according to rule of follow that we studied earlier have a look

$\text{FOLLOW}(C') = \{\$\}$ $\quad \text{FOLLOW}(C) = \{\text{FOLLOW}(C')\} = \{\$\}$

grammar मा rightside C' भएको हुँदा यसि दैन so empty only $ symbol

$\text{FOLLOW}(A) = \{\text{FIRST}(B) \cup a\} = \{a\}$

$\text{FOLLOW}(B) = \{\text{FOLLOW}(C)\} = \{\$\}$

Now finally we construct SLR parsing table as below:

States	Action Table		Goto Table		
	a	$	C	A	B
I_0	Shift I_3		I_1	I_2	
I_1		Accept			
I_2	Shift I_5				I_4
I_3	Reduce R_2				
I_4		Reduce R_1			
I_5		Reduce R_3			

(1) Action table मा terminals a र $, GOTO table मा non-terminals C, A, B

(2) $I_0, I_1 \ldots I_n$ को set हेनमा भएको production हेर्ने mainly 3 case हुन्छन् ।
1) $A \to \alpha \cdot a\beta$ form (i.e, • पछि terminal) आए shift गर्ने ।
2) $A \to \alpha \cdot$ form आए (last मा • भाको) र augmented grammar को 1st production नभए reduce $A \to \alpha$ गर्ने i.e, FOLLOW(A) हेर्ने ।
3) Augmented grammar को first production भए Accept गर्ने ।
यो 3 case मध्ये कुनै पनि नभए हामि neglect गर्ने table मा राख्देनौ

(3) सुरुमा I_0 को set हेर्यौं । $C \to \cdot C$ र $C \to \cdot AB$ ले कुनै पनि 3 case satisfy गरेन so we neglect them । $A \to \cdot a$ first case सँग match गर्छ । So, I_0 मा a को तल shift हुन्छ । DFA मा I_0 मा a आउदा I_3 मा जान्छ so we write Shift I_3
Now we are in I_1 set, we have $C' \to C \cdot$ which satisfies 3rd case so, $ को लागि accept गर्ने according to rule/algorithm, and so on.....

(4) state हेर्ने non-terminal आउदा कता जाको छ त्यो i.e, goto

Short मा shift लाई S, Accept लाई Acc, र Reduce लाई R लेखिन्छ

3rd Reduce वाला case आए, जसले produce गरेको छ त्यसको FOLLOW निकाल्ने । त्यसपछि FOLLOW मा भएको symbol को तल Reduce लेख्ने ।

Example 2: Construct the SLR parsing table for the grammar.

$S \to AA$
$A \to aA \mid b$

Solution:

The augment and cannonical set of LR(0) item as well as DFA is same as we solved before section in example 2, so we directly construct table here calculating FOLLOW as below:

$FOLLOW(S') = \{\$\}$ $FOLLOW(S) = \{FOLLOW(S')\} = \{\$\}$

$FOLLOW(A) = \{FIRST(A) \cup FOLLOW(S)\} = \{\$, a, b\}$

SLR parsing table is as follows:

States	Action Table			Goto Table	
	a	b	$	A	S
I_0	Shift I_3	Shift I_4		I_2	I_1
I_1			Accept		
I_2	Shift I_3	Shift I_4		I_5	
I_3	Shift I_3	Shift I_4		I_6	
I_4	Reduce I_3	Reduce I_3	Reduce I_3		
I_5			Reduce I_1		
I_6	Reduce I_2	Reduce I_2	Reduce I_2		

we can practice more examples from kec book's page no. 46 onwards

table बनाउदा हरेक cell मा more than one action आए जस्तै shift र shift, shift र reduce etc. possibility भएको बेला parsable हुँदैन grammar यसलाई ambiguity in SLR भनिन्छ ।

LR(1) Grammars:

→ 2nd type of LR parser (also called General LR) after shift-reduce parsing. यसमा canonical set of LR(1) निकालेँ instead of LR(0) अरु table draw गर्नु, goto निकाल्नु, closure निकाल्नु सबै same as we did in shift-reduce.

LR(1) parsing uses look-ahead to avoid unnecessary conflicts in parsing table.

LR(1) item = LR(0) item + look-ahead.

LR(1) item भनेको LR(0) जस्तै हो अप यसमा look-ahead भन्नि हुन्छ।

LR(0) item	LR(1) item
$[A \to \alpha \cdot \beta]$	$[A \to \alpha \cdot \beta, a]$

LR(0) item को form को हुन्छ

LR(1) item को form को हुन्छ where a is assumed look-ahead symbol.

Example 1: Construct LR(1) parsing table of following grammar,

$S \to AA$
$A \to aA$
$A \to b$

Solution: The augmented grammar of given grammar is;

$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

The canonical collection of LR(1) items of augmented grammar is:

$I_0 = \text{Closure}(S' \to \cdot S) = \{S' \to \cdot S, \$$
$S \to \cdot AA, \$$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b\}$

same as LR(0) item. comma(,) पछिको look-ahead मात्र फरक यसमा

सुरुको production मा look-ahead $ राखेँ अनि यसलाई rule अनुसार $A \to [\alpha \cdot B\beta, a]$ सँग compare गर्दै dot पछिको एउटा non-terminal symbol छोडेर बाँकि रहेको string को FIRST निकालेँ, त्यो value 2nd production को look-ahead value हुन्छ and so on...
For e.g. $[S' \to \cdot S, \$]$ compared with $[A \to \alpha \cdot B\beta, a]$, now $FIRST(\$) = \{\$\}$ which is look-ahead of $S \to \cdot AA$

• पछिको एउटा symbol छोडि बाँकि को FIRST

Rule for look ahead symbol
If $A \to [\alpha \cdot B\beta, a] \in$ closure(I) then, add the item $[B \to \cdot\gamma, b]$ to I if not already in I. where, $b \in FIRST(\beta a)$

similarly $A \to \cdot aA$ production is given by $S \to \cdot AA, \$$, so, $FIRST(A\$) = FIRST(A) = \{a, b\}$. Hence, look-ahead is a/b.
similarly $A \to \cdot b$ production पनि $S \to \cdot AA, \$$ बाट आएको हो so, same $FIRST(A) = \{a, b\}$

goto गर्दा • पछाडि भएको rule को form मा हेर्ने so दायाँ look-ahead नया calculate नगरि माथिकै राख्ने

$I_1 = \text{Goto}(I_0, S) = \text{closure}(S' \to S^\cdot, \$) = \{S' \to S^\cdot, \$\}$

$I_2 = \text{Goto}(I_0, A) = \text{closure}(S \to A \cdot A, \$) = \{S \to A \cdot A, \$$ (copy)
$A \to \cdot aA, \$$
$A \to \cdot b, \$\}$

यो production $S \to A \cdot A, \$$ बाट गर्दा आको so $FIRST(\$) = \$$ is look-ahead by rule similarly this.

$I_3 = \text{Goto}(I_0, a) = \text{closure}(A \to a \cdot A, a/b) = \{A \to a \cdot A, a/b$
$A \to \cdot aA, a/b$
$A \to \cdot b, a/b\}$

β absent भए माथि भने झै look-ahead नल पनि look-ahead new calculate गर्नु पर्ने

$I_4 = Goto(I_0, b) = closure(A \to b^\bullet, a/b) = \{A \to b^\bullet, a/b\}$

$I_5 = Goto(I_2, A) = closure(S \to AA^\bullet, \$) = \{S \to AA^\bullet, \$\}$

$I_6 = Goto(I_2, a) = closure(A \to a^\bullet A, \$) = \{A \to a^\bullet A, \$$
$A \to {}^\bullet aA, \$$
$A \to {}^\bullet b, \$\}$

β absent here so same $, $ in look-ahead as before

$I_7 = Goto(I_2, b) = closure(A \to b^\bullet, \$) = \{A \to b^\bullet, \$\}$

$I_8 = Goto(I_3, A) = closure(A \to aA^\bullet, a/b) = \{A \to aA^\bullet, a/b\}$

$Goto(I_3, a) = closure(A \to a^\bullet A, a/b) = \{A \to a^\bullet A, a/b$
$A \to {}^\bullet aA, a/b$
$A \to {}^\bullet b, a/b\}$ same as I_3

can be assigned to I_3 or left without assigned but understand Goto$(I_3, a) = I_3$

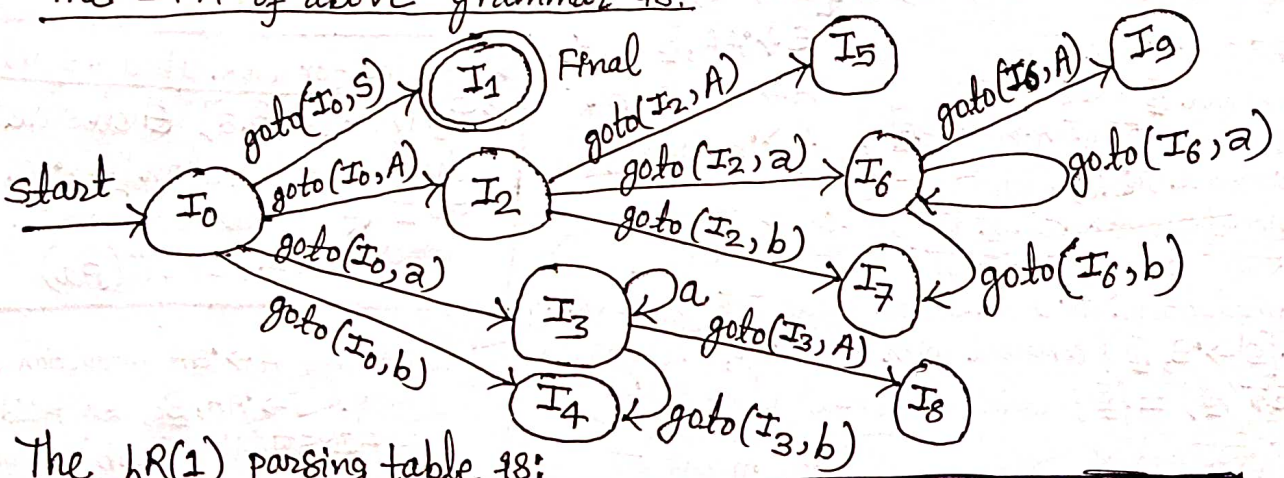
$Goto(I_3, b) = closure(A \to b^\bullet, a/b)$ same as I_4.

$I_9 = Goto(I_6, A) = closure(A \to aA^\bullet, \$) = \{A \to aA^\bullet, \$\}$

$Goto(I_6, a) = closure(A \to a^\bullet A, \$)$ same as I_6.

$Goto(I_6, b) = closure(A \to b^\bullet, \$)$ same as I_7.

The DFA of above grammar is:



The LR(1) parsing table is:

States	Action table			Goto Table	
	a	b	$	S	A
0	Shift I_3	Shift I_4		I_1	I_2
1			Accept		
2	Shift I_6	Shift I_7			I_5
3	Shift I_3	Shift I_4			I_8
4	Reduce I_3	Reduce I_3			
5			Reduce I_1		
6	Shift I_6	Shift I_7			I_9
7			Reduce I_3		
8	Reduce I_2	Reduce I_2			
9			Reduce I_2		

iii) LALR(1) Grammars:

everything same as LR(1) only difference here is we combine states having same productions, but can have different look-ahead symbol.

Example1: Construct LALR parsing table for following grammar,

$S \to L=R$

= is terminal symbol and L=R is one single string. Only next string when seperated by |

$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Solution:

The augmented grammar of above grammar is,

$S' \to S$
$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

At first we find the canonical collection of LR(1) items of the given augmented grammar as;

$I_0 = closure(S' \to \cdot S, \$)$
$= \{ S' \to \cdot S, \$$
$S \to \cdot L=R, \$$
$S \to \cdot R, \$$
$L \to \cdot *R, \$$
$L \to \cdot id, =$
$R \to \cdot L, \$ \}$

$I_1 = closure(goto(I_0, S))$
$= closure(S' \to S \cdot, \$)$
$= \{ S' \to S \cdot, \$$
$R \to L \cdot, \$ \}$

$I_2 = closure(goto(I_0, L))$
$= closure(\{S \to L \cdot = R, \$),$
$(R \to L \cdot, \$)\}$
$= \{ S \to L \cdot = R, \$ \}$

$I_3 = closure(goto(I_0, R))$
$= closure(S \to R \cdot, \$)$
$= \{ S \to R \cdot, \$ \}$

$I_4 = closure(goto(I_0, *))$
$= closure(L \to * \cdot R, =)$
$= \{(L \to * \cdot R, =), (R \to \cdot L, =),$
$(L \to \cdot *R, =), (L \to \cdot id, =)\}$

$I_5 = closure(goto(I_0, id))$
$= closure(L \to id \cdot, =)$
$= \{ L \to id \cdot, = \}$

$I_6 = closure(goto(I_2, =))$
$= closure(S \to L= \cdot R, \$)$
$= \{ S \to L= \cdot R, \$$
$R \to \cdot L, \$$
$L \to \cdot *R, \$$
$L \to \cdot id, \$ \}$

$I_7 = closure(goto(I_4, R))$
$= closure(L \to *R \cdot, =)$
$= \{ L \to *R \cdot, = \}$

$I_8 = closure(goto(I_4, L))$
$= closure(R \to L \cdot, =)$
$= \{ R \to L \cdot, = \}$

$I_{12} = closure(goto(I_6, id))$
$= closure(L \to id \cdot, \$)$
$= \{ L \to id \cdot, \$ \}$

$I_{13} = closure(goto(I_{11}, R))$
$= \{ L \to *R \cdot, \$ \}$

यहाँ 12 र 13, 11 पनि last मा हुने उता last मा गर्ने, दुईको हेड्ड so यता राखेको

$I_9 = closure(goto(I_6, R))$
$= closure(S \to L = R^\bullet, \$)$
$= \{S \to L = R^\bullet, \$\}$

$I_{10} = closure(goto(I_6, L))$
$= \{R \to L^\bullet, \$\}$

$I_{11} = closure(goto(I_6, *))$
$= closure(L \to *^\bullet R, \$)$
$= \{L \to *^\bullet R, \$$
$R \to {}^\bullet L, \$$
$L \to {}^\bullet *R, \$$
$L \to {}^\bullet id, \$\}$

यो सम्म सब same हो as before अब यो combine गर्ने मात्र different

Now we combine states as follows:

Combine गर्दा दुईवटै state मा same no. of productions हुने पर्यो र core production same हुने पर सबै look-ahed symbol छोडेर बाकी बचेको लाई core production भनिन्छ।

Combine state 4 and 11 as:

$I_4: \{(L \to *^\bullet R, =), (R \to {}^\bullet L, =), (L \to {}^\bullet *R, =), (L \to {}^\bullet id, =)\}$
$I_{11}: \{(L \to *^\bullet R, \$), (R \to {}^\bullet L, \$), (L \to {}^\bullet *R, \$), (L \to {}^\bullet id, \$)\}$
$I_{4,11}: \{(L \to *^\bullet R, =/\$), (R \to {}^\bullet L, =/\$), (L \to {}^\bullet *R, =/\$), (L \to {}^\bullet id, =/\$)\}$

Combine गरेको दुईवटै state को symbol look-ahead slash गरी लेखेको

Combine state 5 and 12 as;

$I_5: \{L \to id^\bullet, =\}$
$I_{12}: \{L \to id^\bullet, \$\}$
$I_{5,12}: \{L \to id^\bullet, =/\$\}$

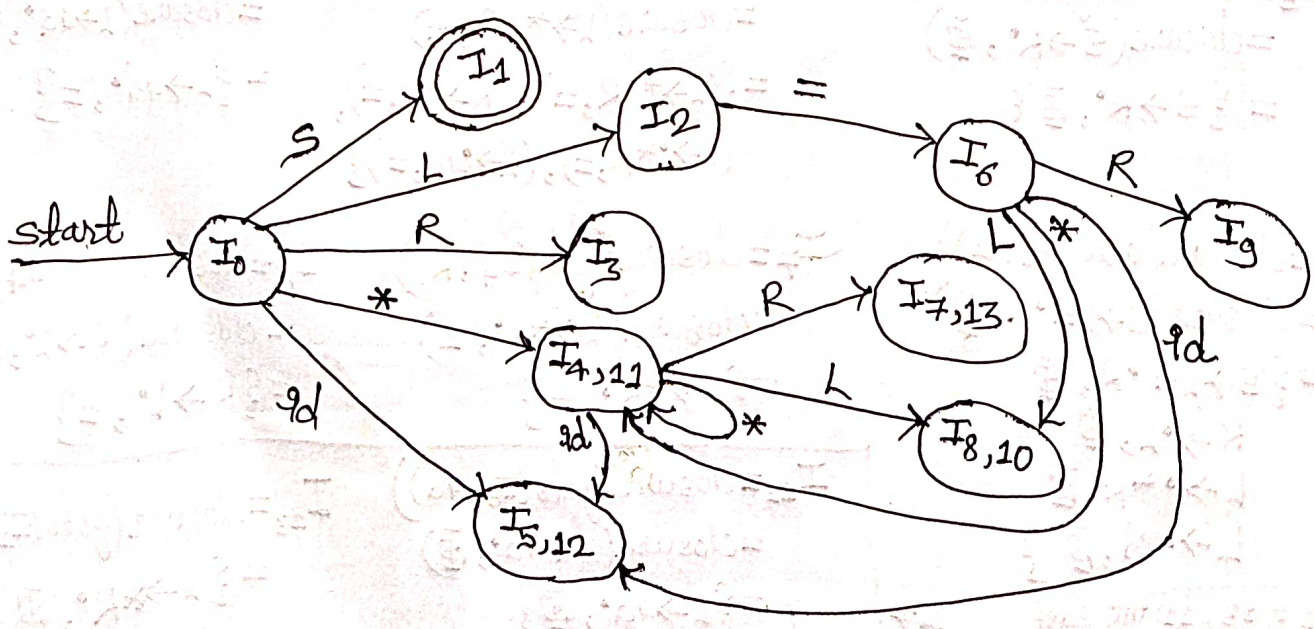
Combine state 7 and 13 as;

$I_7: \{L \to *R^\bullet, =\}$
$I_{13}: \{L \to *R^\bullet, \$\}$
$I_{7,13}: \{L \to *R^\bullet, =/\$\}$

Combine state 8 and 10 as;

$I_8: \{R \to L^\bullet, =\}$
$I_{10}: \{R \to L^\bullet, \$\}$
$I_{8,10}: \{R \to L^\bullet, =/\$\}$

Now the DFA of LALR parsing is:



The LALR parsing table is:

States	Action Table				Goto Table		
	id	*	=	$	S	L	R
0	Shift I_5	Shift I_4			I_1	I_2	I_3
1				Accept			
2			Shift I_6	Reduce I_5			
3				Reduce I_2			
4	Shift I_5	Shift I_4				I_8	I_9
5			Reduce I_4	Reduce I_4			
6	Shift I_{12}	Shift I_{11}				I_{10}	I_9
7			Reduce I_3	Reduce I_3			
8			Reduce I_5	Reduce I_5			
9				Reduce I_1			

table also same method as before ↔ यो I_4 भनेको अब $I_{4,11}$ हो combine गरेको भएर similarly others which are combined.

⊛. Kernel and Non-Kernel Items:

Kernel item includes the initial items, $S' \to .S$ and all items whose dot are not at the left end. Similarly non-kernel items are those items which have their dots at the left end except $S' \to .S$

Example: Find the kernel and non-kernel items of following grammar,

$C \to AB$
$A \to a$
$B \to a$

Solution:

The augmented grammar of given grammar is,

$C' \to C$
$C \to AB$
$A \to a$
$B \to a$

Now, we obtain the canonical collection of sets of LR(0) items as follows,

$I_0 = \text{closure}(\{C' \to \cdot C\}) = \{C' \to \cdot C, C \to \cdot AB, A \to \cdot a\}$

$I_1 = \text{goto}(I_0, C) = \text{closure}(C' \to C\cdot) = \{C' \to C\cdot\}$

$I_2 = \text{goto}(I_0, A) = \text{closure}(C \to A\cdot B) = \{C \to A\cdot B, B \to \cdot a\}$

$I_3 = \text{goto}(I_0, a) = \text{closure}(A \to a\cdot) = \{A \to a\cdot\}$

$I_4 = goto(I_2, B) = closure(C \to AB^\bullet) = \{C \to AB^\bullet\}$

$I_5 = goto(I_2, a) = closure(B \to a^\bullet) = \{B \to a^\bullet\}$

List of kernel and non-kernel items are listed below:

States	Kernel Items	Non-kernel items
I_0	$C' \to {}^\bullet C$	$C \to {}^\bullet AB$ $A \to {}^\bullet a$
I_1	$C' \to C^\bullet$	
I_2	$C \to A^\bullet B$	$B \to {}^\bullet a$
I_3	$A \to a^\bullet$	
I_4	$C \to AB^\bullet$	
I_5	$B \to a^\bullet$	

Augmented grammar को सुरुको production S सुरुमा dot (•) भएको सबै kernel item. सुरुको production $C' \to {}^\bullet C$ बाहेक सुरुमा dot (•) भएको सबै non-kernel item

⊛ Top down Parsing vs. Bottom up Parsing:

S.N.	Top down parsing	Bottom up parsing
1.	It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
2.	This parsing technique uses left most derivation.	This parsing technique uses right most derivation.
3.	Its main decision is to select what production rule to use in order to construct the string.	Its main decision is to select when to use a production rule to reduce the string to get the starting symbol.
4.	Error detection is easy	Error detection is difficult.
5.	Parsing table size is small.	Parsing table size is bigger.
6.	Less Power	High Power.

Semantic Analysis

यो बाट 1 short question आउन 5 marks को लागि

Introduction: Semantic Analysis is the third phase of compiler which provides meaning to its constructs, like tokens and syntax structure.

CFG + semantic rules = Syntax Directed Definitions.

Examples of Semantic Errors:

Example1: Use of a non-initalized variable.

```
int i;
i++; //the variable i is not initialized.
```

Example2: Type incompatibility

```
int a="hello"; //the types String and int are not compatible.
```

Example3: Errors in expressions

```
char s='A';
int a=15-s; //the '-' operator does not support type char.
```

Example4: Array index out of range.

```
int a[10];
a[12]=25; //12 is not legal index, we have index 0 to 9
          for array of size 10.
```

Example5: Unknown references

```
int a;
printf("%d", a); //a is not initialized.
```

Type Checking: Compiler must check that the source program follows both the syntatic and semantic conventions of the source language. Type checking is the process of checking the data type of different variables. The design of a type checker for a language is based on information about the syntatic construct in the language, the notation of type, and the rules for assigning types to the language constructs.

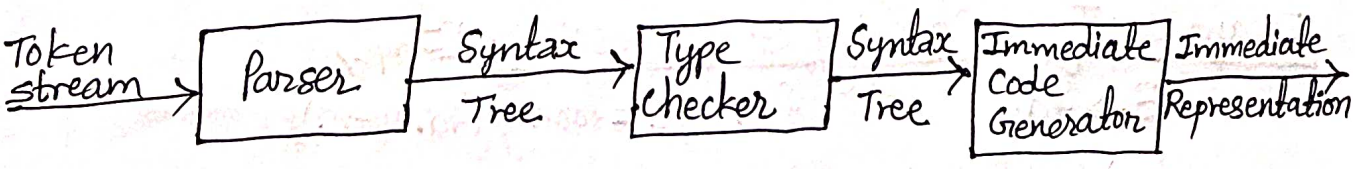


Fig: Position of Type Checker.

⊛ **Type Systems:** The collection of different data types and their associated rules to assign types to programming language constructs is known as type systems. It is an informal type system rules, for example "if both operands of addition are of type integer, then the result is of type integer." A type checker implements type system.

⊛ **Type Expressions:** The type of a language construct will be denoted by a "type expression". A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked.

Constructors include: ← just understand only

- **Arrays:** If T is a type expression then array(I,T) is a type expression denoting the type of an array with elements of type T and index set I. Example: array(0...99, int).
- **Products:** If T1 and T2 are type expressions, then their Cartesian product T1 × T2 is a type expression. Example: int × int.
- **Records:** The record type constructor will be applied to a tuple formed from field names and field types.
- **Pointers:** If T is a type expression, then pointer(T) is a type expression denoting the type "pointer to an object of type T." For example, var p: ↑ row declares variable p have type pointer(row).
- **Functions:** A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D→R. Example: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Example: Type Checking of Expressions: [Imp]

E → id	{E.type = lookup(id.entry)}
E → charliteral	{E.type = char}
E → intliteral	{E.type = int}

$E \rightarrow E1 + E2$	{E.type = (E1.type == E2.type) ? E1.type : type_error}
$E \rightarrow E1 \uparrow$	{E.type = (E1.type == pointer(t)) ? t : type_error}
$E \rightarrow E1 [E2]$	{E.type = (E2.type == int and E1.type == array(s,t)) ? t : type_error}
$S \rightarrow id = E$	{S.type = (id.type == E.type) ? void : type_error}

(This bracket denotes index of array)

$S \rightarrow$ if E then S1	{S.type = (E.type == boolean) ? S1.type : type_error}
$S \rightarrow$ while E do S1	{S.type = (E.type == boolean) ? S1.type : type_error}
$S \rightarrow S1 ; S2$	{S.type = (S1.type == void and S2.type == void) ? void : type_error}

⊛Static vs. Dynamic Type Checking: [Imp]

Static Type Checking	Dynamic Type Checking
The type checking at compilation time is known as static type checking.	The type checking at runtime is known as dynamic type checking.
A language is statically-typed if the type of variable is known at compile time.	A language is strongly-typed, if every program it's compiler accepts will execute without type errors.
It includes languages such as C, C++, C#, Java, FORTRAN, Pascal etc.	It includes languages such as JavaScript, LISP, PHP, Python, Ruby etc.

Typical examples of static checking are:
→ Type checks
→ Flow-of-control checks
→ Uniqueness checks
→ Name-related checks
	Example: array out of bounds as below;
	`int a[10];`
	`for(int i=0; i<20; i++)`
	`a[i]=i;`

It is the process of reducing possibilities for bugs in programs and then checking that parts of program have been connected in a consistent way.	It is the process of verifying the type safety of a program at runtime.

⑧. Type Casting vs. Type Conversion:

Type Casting	Type Conversion (Coercion)
In type casting, a data type is converted into another data type by a programmer using casting operator.	In type conversion, a data type is converted into another data type by a compiler.
Type casting can be applied to compatible data types as well as incompatible data types.	Type conversion can only be applied to compatible data types.
In type casting, casting operator is needed in order to cast the data type to another data type.	In type conversion there is no need for a casting operator.
In type casting, the destination data type may be smaller than the source data type, when converting the data type to another data type.	In type conversion, the destination data type can't be smaller than source data type.
Type casting takes place during the program design by programmers.	Type conversion is done at the compile time.

⊛ Syntax-Directed Translation (STD):

(STD भनेर describe गर्नु आए यसको defn र SDD, SDTS short मा describe गर्ने)

Syntax-Directed Translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser. Almost all modern compilers are syntax-directed. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in same order. There are two ways to represent semantic rules associated with grammar symbols.

→ Syntax-Directed Definitions (SDD).
→ Syntax-Directed Translation Schemes (SDTS).

(explain this with suitable example भनेर सोधेमा normally so अलि explain गर्ने र 1,1 example पुग्छ।)

i) Syntax-Directed Definitions (SDD): [Imp]

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. A SDD is a generalization of a context-free grammar in which each grammar symbol is associated with a set of attributes.

Example: The syntax directed definition for a simple desk calculator.

Production	Semantic Rules
L → E return	Print (E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → digit	F.val = digit.lexval

(digit: any variable that does not produce further)

ii) Syntax-Directed Translation Schemes (SDTS): [Imp]

SDTS embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed. It is used to evaluate the order of semantic rules.

Syntax, $A \rightarrow \{\ldots\} X \{\ldots\} Y \{\ldots\}$

where, within the curly brackets we define semantic actions.

Example: A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

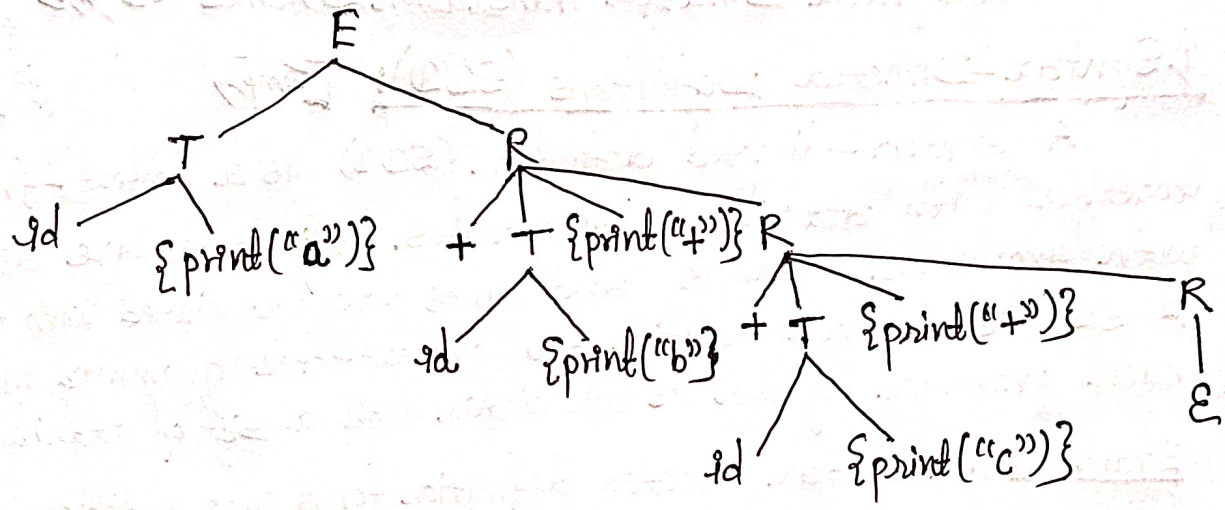
$E \rightarrow TR$

$R \rightarrow +T \{print("+")\} R_1$

$R \rightarrow \varepsilon$

$T \rightarrow id \{print(id.name)\}$

Infix expression (a+b+c) ⟶ postfix expression (ab+c+).



⊛. ATTRIBUTE TYPES:

1) Synthesized Attribute (↑): If the value of the attribute only depends upon it's children then it is synthesized attribute. Simply we can also say that the attributes of a node that are derived from it's children nodes are called synthesized attributes. Let we have $S \rightarrow ABC$ as our production, now if S is taking values from it's child nodes (A, B, C) then it is said to be a synthesized attribute.

Example for Synthesized Attributes:

Production	Semantic Rules
L→E return	print (E.val)
E→E1+T	E.val = E1.val + T.val
E→T	E.val = T.val
T→T1*F	T.val = T1.val * F.val
F→digit	F.val = digit.lexval

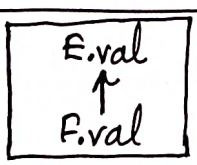
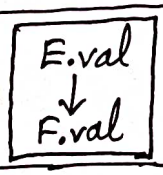
Same example as SDD

2) **Inherited Attribute (→, ↓):** A node in which attributes are derived from the parent or siblings of node is called inherited attribute of that node. If the value of the attribute depends upon it's parent or siblings then it is inherited attribute. Let we have production as $S \rightarrow ABC$, now if 'A' gets values from S, B and C, similarly if B can take values from S, A, C, likewise C can take values from S, A, and B.

Example for Inherited Attributes:

Production	Semantic Rules
D → TL	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L_1, id	L_1.in = L.in; addtype (id.entry, L.in)
L → id	addtype (id.entry, L.in).

⊛. Synthesized attributes vs. Inherited attributes:

Synthesized attribute	Inherited attributes.
Attributes of a node that are derived from its children nodes are called synthesized attributes.	Attributes which are derived from the parent or siblings of node are called inherited attribute.
The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
Synthesized attributes can be contained by both the terminals and non-terminals.	Inherited attributes can be only contained by non-terminals.
Example: E → F, E.val = F.val 	Example: E → F, E.val = F.val 

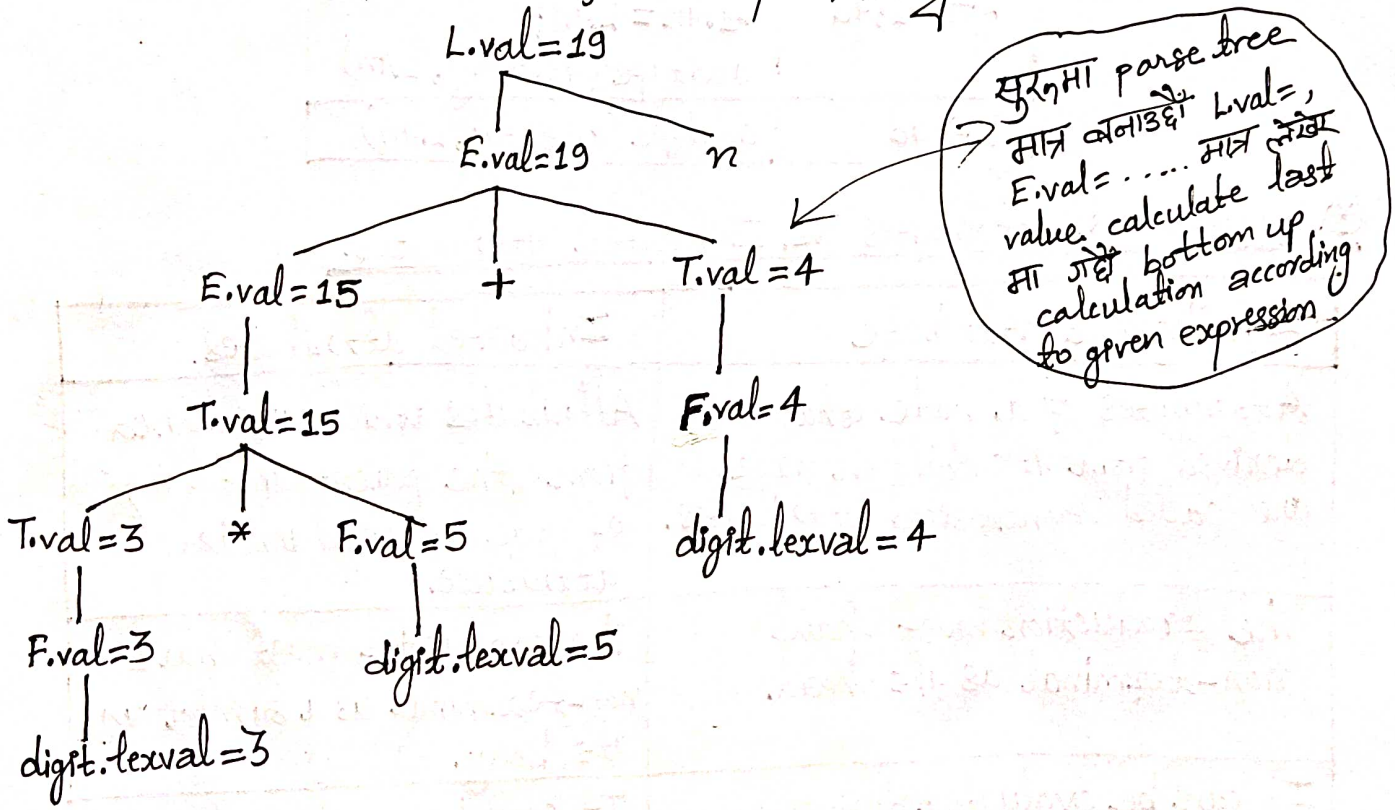
⊛ Annoted Parse Tree:

A parse tree constructing for a given input string in which each node showing the values of attributes is called an anoted parse tree. It is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annoting or decorating the parse tree.

Example 1: Lets take a grammar,

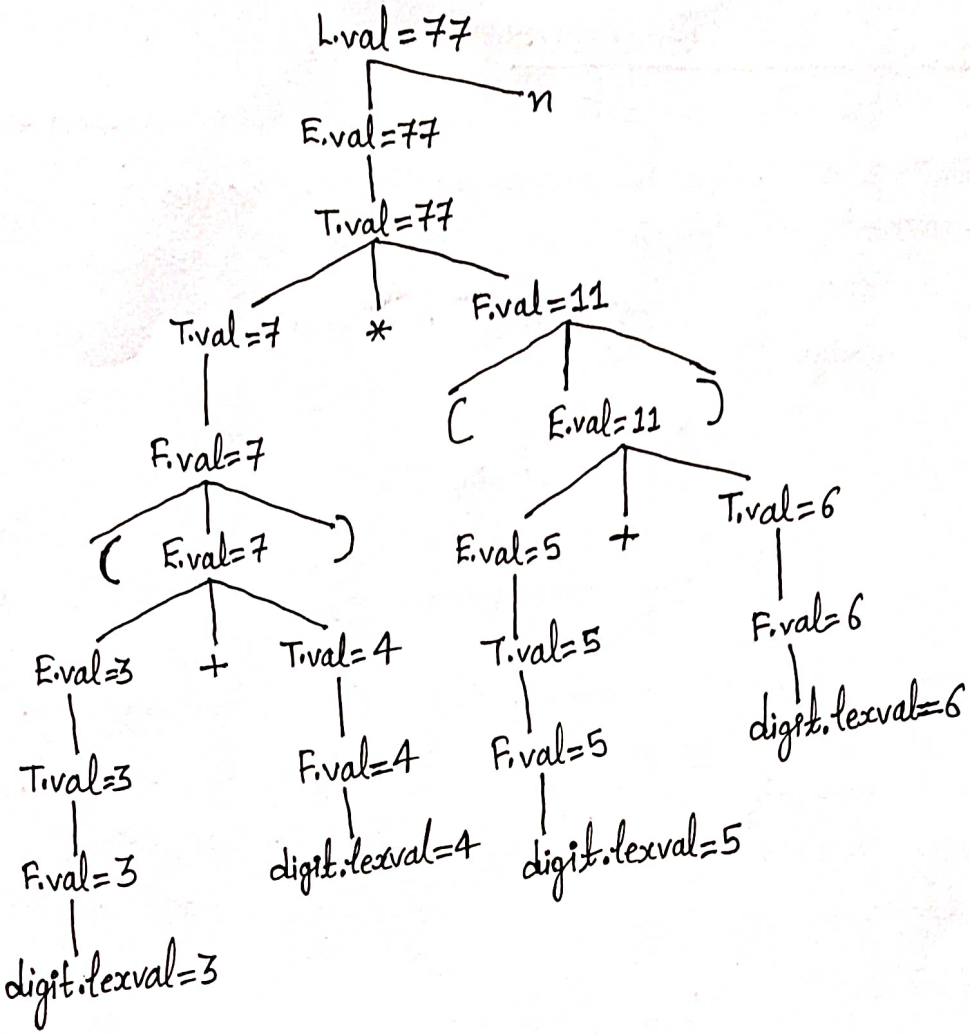
$L \to E\,n$
$E \to E1 + T$
$E \to T$
$T \to T_1 * F$
$T \to F$
$F \to (E)$
$F \to digit.$

Now the annoted parse tree for the input string 3*5+4 is,



Example 2: For the Syntax-Directed Definitions (SDD) of the grammar, $(3+4)*(5+6)n$, give annoted parse tree.

Solution:



⊛. S-Attributed Definitions:

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. Attribute values for the non-terminal at the head is computed from the attribute values of the symbols at the body of the production. The attributes of an S-attributed SDD can be evaluated in bottom up order of nodes of the parse tree.

Example for S-attributed definitions:

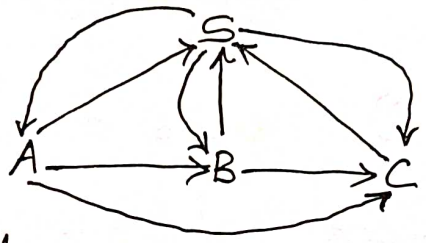
Production	Semantic Rules
L→E return	print (E.val)
E→E1+T	E.val= E1.val +T.val
E→T	E.val=T.val
T→F	T.val= F.val
F→digit	F.val =digit.lexval

Same example as before

⊛ L-Attributed Definitions:

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes, as in the following production:

$S \rightarrow ABC$



S can take values from A, B, and C (synthesized). 'A' can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right like A does not take value from B or C. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Example:

Production	Semantic Rules
$T \rightarrow FT'$	T'.inh = F.val
$T' \rightarrow *FT_1'$	T_1'.inh = T'.inh * F.val