Teacher's Sign / Remarks

Assignment 0:-

⇒ How many subjects have you been teaching?
(Set zero to the zero)
Name, Code, Credits

⇒ Which two subjects you liked the most

⇒ Which two subjects you scored the best

⇒ Name 2 teachers of that subject you liked the most

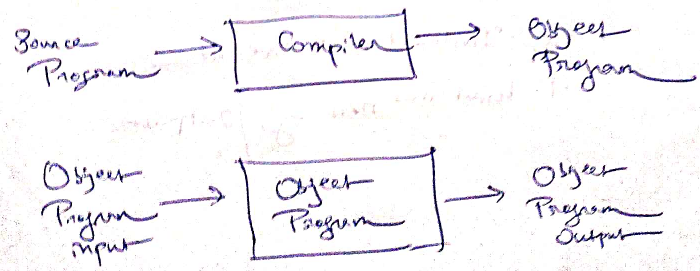
⇒ Go to website and count research paper of all faculty available.

How many SCI / SCI journal paper (optional)
Science citation index

Book:-
Principles of Compiler Design
Aho & Ullman

Class sheet
Class test
Assignment (10)
feedback
download syllabus
Google classroom

(Remember password for entry in CSE dept)

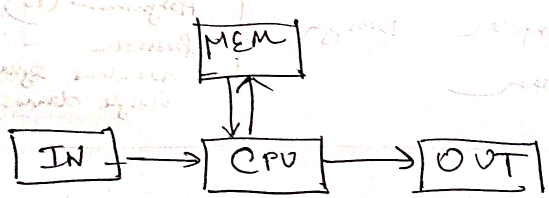


SYNTEX
SEMANTICS

History of the Computer → Quiz / Assignment
History of Compiler →

x := a + b * c → Pascal notation

x = a + b * c → Assignment Statement in C



Computer Block Diagram

1. Instruction Fetch
2. Data Fetch
3. Instruction Execute
4. Result in memory / output

Assignment 1

1. => What are the main phases of the Compiler with a schematic diagram explain

25/01/24

begin
If x=10 then
y := 6
else
y := 10;
end;

Lexical analysis
Syntax (Process)
Intermediate Code
Code optimization
Code generation

Phases of Compiler design

First step in lexical analysis is token identification.

begin, if, x, =, 10, then, y, :=, 6, else, end

these are tokens

; is a delimiter

Convert the signal translate HLL to machine code

Translates the whole program into machine code before program is run

⇒ Difference between converter and compiler. Compare all these.

and translator.

Converts one language to another language

translates HLL to machine code

Infix, Prefix, Postfix

Tokens

The string representing a program can be partitioned at the lowest level into a sequence of substrings called tokens.

Tokens are lowest level of string which has some meaning to the compiler. Each of the token is a sequence of characters whose significance is possessed collectively rather than individually.

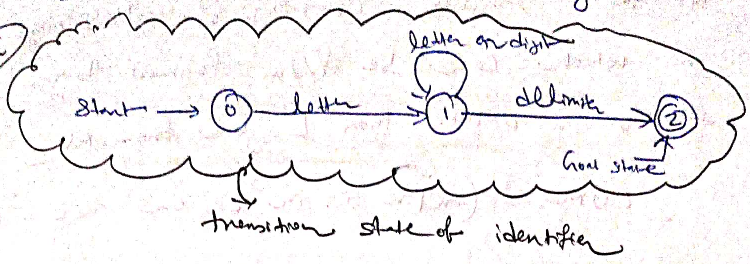
The token depends on the language.

All languages have such type of tokens:—

(i) Constant (eg. 6, 10, 5 etc.)
(ii) identifiers (eg. a, b, c, d etc.)
(iii) Operators (eg. +, −, /, % etc.)
(iv) Keywords (eg. if, else, for, while, int etc.)
(v) Punctuations (eg. ;, (), , etc.)

Identifier may contain any letter followed by any number of letters or digits.

Identifier



String

A string is a finite sequence of symbols taken from some alphabet or character class.

Prefix

If x is a string formed by discarding zero or more trailing symbols of x is called prefix of x.
eg. abc is a prefix of abcd.

Suffix

A suffix of x is a string formed by zero or more of leading symbols of x,
eg. cde is suffix of abcde

Language

A language is a set of strings from some specific alphabet.

If L and M are language then L.M or simply LM is the language consisting of all string xy which can be formed by selecting a string x from L, a string y from M and concatenating them in that order.
$LM = \{xy \mid x \text{ is in } L \text{ and } y \text{ is in } M\}$.

If L and M are languages then LUM is the language consisting of all strings of w which can be ~~the~~ formed by selecting w either from L or from M

$$L \cup M = (w \mid w \text{ is in } L \text{ or } w \text{ is in } M)$$

Regular Expression

Regular expression over alphabet Σ are exactly those expression which may be constructed using following rules:—

(i) ε (NULL) is regular expression denoting $\{\varepsilon\}$

(ii) For each a in Σ a is a regular expression denoting $\langle a \rangle$, the language with only one string the string containing ~~Symbol~~ single symbol a.

(iii) If R and S are regular expression denoting Languages L_R and L_S respectively then

⇒ $(R) \mid (S)$ is a regular expression denoting $L_R \cup L_S$

⇒ $(R).(S)$ " " " " " $L_R \cdot L_S$

⇒ $(R)^*$ " " " " " L_R^*

⇒ $(R)^+$ " " " " " L_R^+

eg. If $R = a$, then,

$$R^* = \varepsilon | R | RR | RRR | \ldots$$
$$R^+ = R | RR | RRR | \ldots$$

⇒ Axioms of regular expression

⇒ $R|S = S|R$ (| is commutative)

⇒ $R|(S|T) = (R|S)|T$ (| is associative)

⇒ $R.(S.T) = (R.S).T$ (. is associative)

⇒ $R.(S|T) = R.S | R.T$ (. is distributive over |)

⇒ $(S|T).R = S.R | T.R$ (. is distributive over |)

---

eg Keyword = BEGIN | END | IF | THEN | ELSE

identifier = letter (letter | digit)*

Constant = digit+

relop = < | > | <= | = | <> | >=

↓

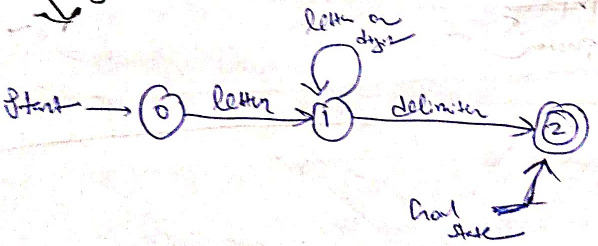
relational operator
~~Boolean~~
operator

Non-deterministic (behaviour if has multiple transitions) Finite Automata (NFA)

It is a state transition diagram derived from regular expression. Every NFA has a start state, final state. The intermediate states are defined for components of regular expression, the NFA for regular expression letter (letter | digit)* is represented as

⇒ How to define Infinite automata:

An infinite automata is a state machine that can process infinite inputs such as infinite words or infinite trees. It is an extension of finite automata which can only process finite inputs. Infinite automata are useful for verifying systems with infinite state spaces or studying formal languages that are not regular.



eg. ab1c2n

a*

a* = ε,

a+ =

ab*

(ab)+

ab

(a|b)

Start

State

0

1

2

a^* a^+ ab^* $(ab)^+$

$a^* = \varepsilon, a, aa, aaa, aaaa, aaaaa$

$a^+ = aa, aaa, aaaaaa, aa, aaaa$

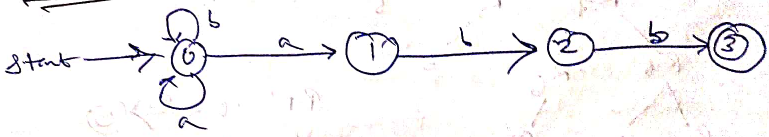
$ab^* = ab, a, abb, abbb, abbbb$

$(ab)^+ = ab, abab, ababab, abababab, ababababab$

---

abb, cf, a1d, abcd, a234, b23

$\Rightarrow (a|b)^* abb$



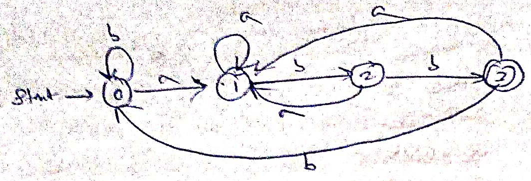
State	a	b
0	0, 1	0
1		2
2		3

State	Remaining input
0	aabb
0	abb
1	bb
2	b
3	ε

aabb, abb, babb, bbabb, abbabb

DFA

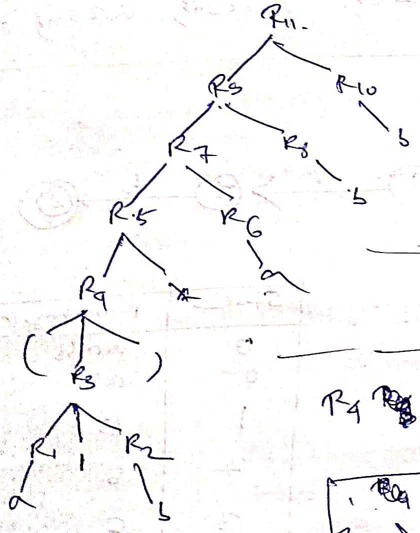
(Deterministic Finite Automata)



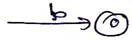
State	Remaining input
0	aabb
1	abb
1	bb
2	b
3	ε

State	a	b
0	1	0
1	1	2
2	1	3
3	1	0

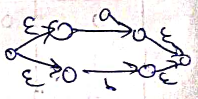
⇒ regular expression of (a|b)* abb may be decomposed into a tree structure

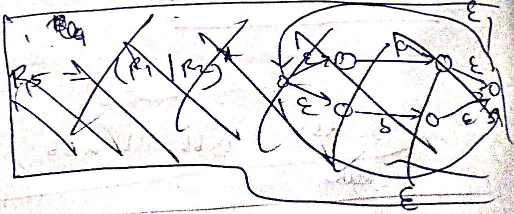


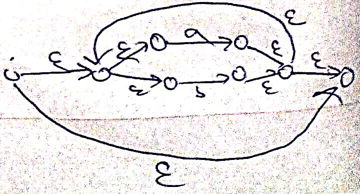
R1 : 

R2 : 

R3 :

R4 = (R1 | R2) 



R5 = (R1|R2)* 

R6 0 —a→ ◎

R7 = (R1|R2)* R6

R8 : 0 —b→ ◎

R9 = R7 R8

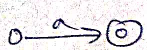
R10 : 0 —b→

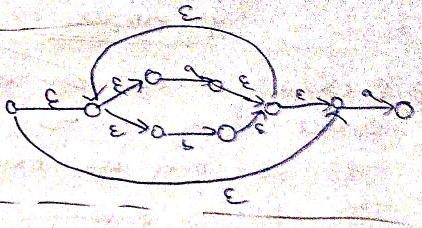
R11 = R9 R10

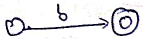
start → 0

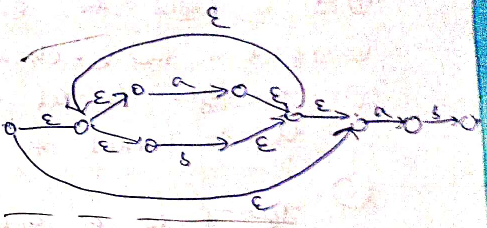
Important ... least o...

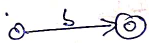
Give...

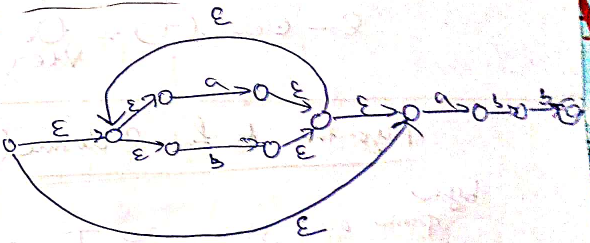
R_6 

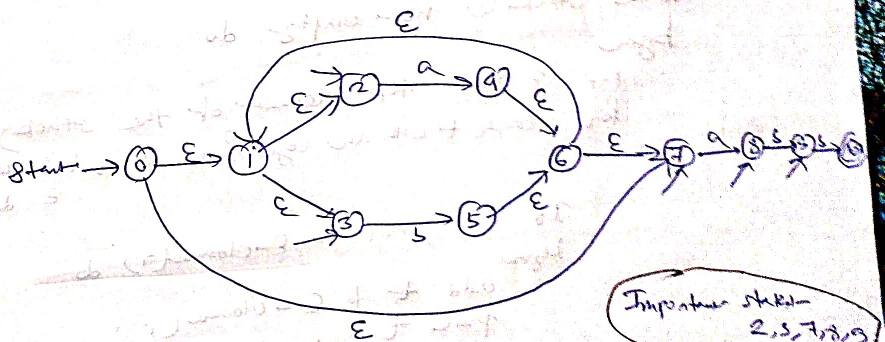
$R_7 = (R_1 | R_2)^* R_6$ 

R_8 : 

$R_9 = R_7 R_8$ 

R_{10}^* 

$R_{11} = R_9 R_{10}$ 



Important states are those states from which at least one non ε transition exists.

⊛ Given expression, define the grammar draw NFA

ε- closure

Starting state

⇒ ε-closure is the set of sets that can be reached from S on ε-transition alone. This set must be obtained by applying the following rules—

(i) S is added to ε-closure

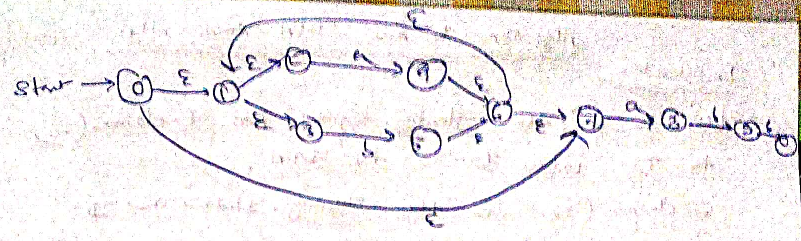
(ii) If t is in ε-closure then if there is an edge labelled ε from t to u then u is added to ε-closure, i.e.

(iii) If T is a set of states

$$\varepsilon\text{-closure}(T) = \bigcup_{\forall t \in T} \varepsilon\text{-closure}(t)$$

Algorithm to find ε-closure(T)

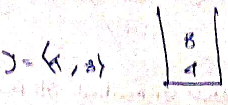
```
begin
  push all states in T onto stack;
  ε-closure(T) := T;
  while stack is not empty do
  begin
    pop S the top element of the stack;
    for each t with an edge from S to t labeled
                                        ε do
      If t is not in ε-closure(T) do
      begin
        add t to ε-closure(T);
        push t onto stack;
      end;
  end;
end;
```

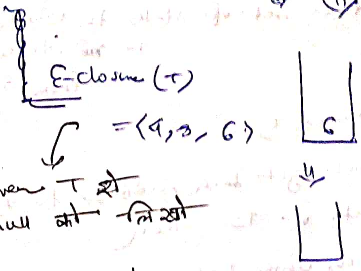


T = ⟨4, 8⟩

ε-closure(T) = ⟨4, 8⟩

(null मिलता चालते रहो, a मारत करो)





ε-closure(T)
= ⟨4, 8, 6⟩

Given T को null का लिखो

ε-closure(T)
= ⟨4, 8, 6, 1⟩

ε-closure(T)
= ⟨4, 8, 6, 1, 7⟩

ε-closure(T) = ⟨4, 8, 6, 1, 7, 2, 3⟩

If T = ⟨5⟩

ε-closure(T) = ⟨5⟩

⟨5, 6⟩

⟨5, 6, 7⟩

⟨5, 6, 7, 1⟩

⟨5, 6, 7, 1, 2, 3⟩

If T = ⟨0⟩

ε-closure(T)

= ⟨0⟩

⟨0, 1⟩

⟨0, 1, 2, 3⟩

⟨0, 1, 2, 3, 7⟩

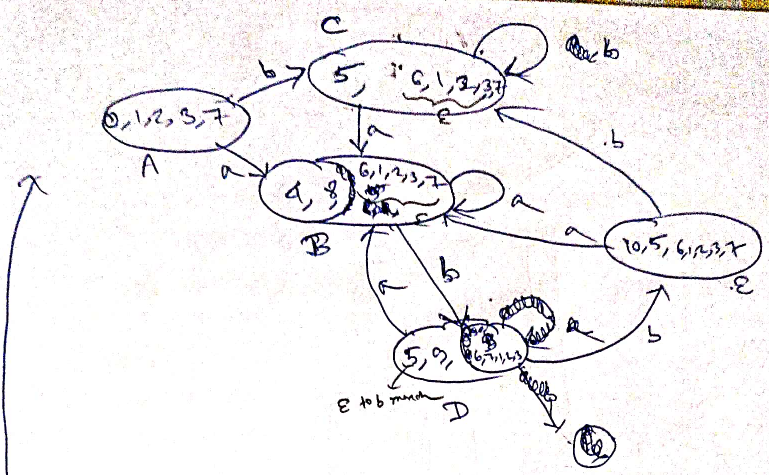
Algorithm to find DFA from NFA

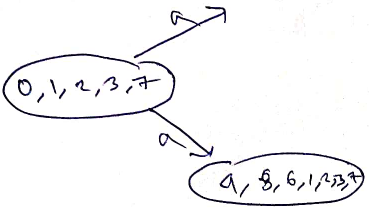
Initialization

Let S_0 be the starting symbol. Add ε-closure(S_0) to D, where D is the DFA.

ε-closure(S_0) is the starting state in D and is initially unmarked.

While there is an unmarked state $n = (S_1, S_2, \ldots S_n)$ do
begin
  mark n;
  For each input a do
  begin
    let T be the set of states to which there is a transition on a from S_i in n;
    y := ε-Closure(T)
    if y has not yet been added to the set of states of D then
    mark y as unmarked state of D;
    add a transition from n to y labeled 'a' if not already present
  end;
end;


C
A
B
D
E
a
b
ε


a
a

PASCAL

```
Begin
While B do
P := Construct
end
```

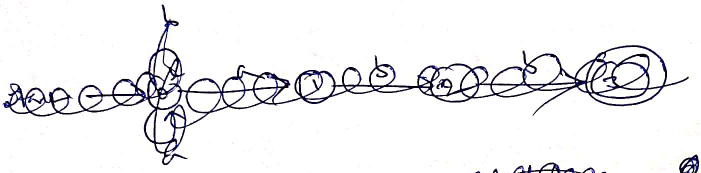
Disadvantage → Assign same construct repeatedly

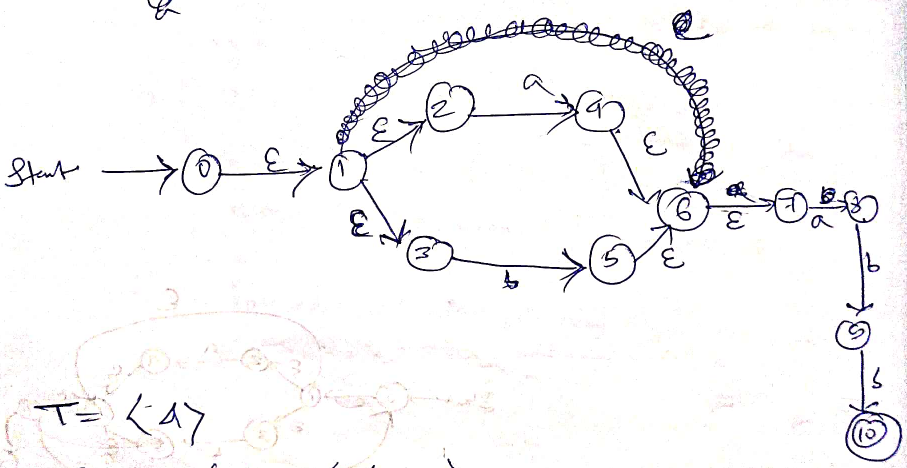
assign

Loop optimization ⟹

```
P := Construct
begin
While B do
end
```

$(a|b)^+ abb$



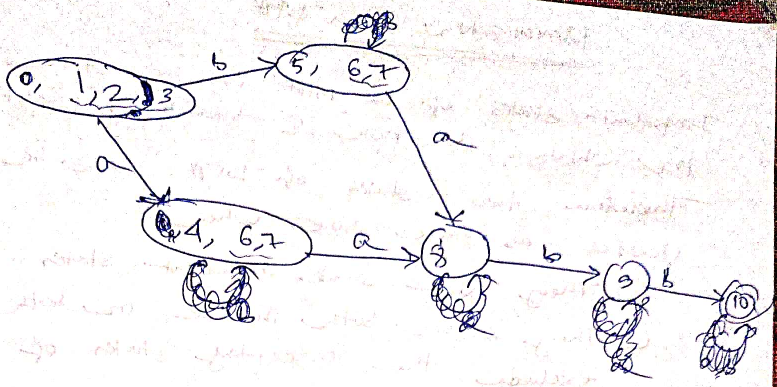


$T = \{1\}$

ε-closure$(T) = \{4, 6\}$

⇓

ε-closure$(T) = \{4, 6, 7\}$



i= and | n(a|b|c)*

Assignment

decompose
NFA
ε-closure
DFA

eg. Example of regular expression

photo

Dance

Dance it

wed

Keyword = BEGIN | END | IF | THEN | ELSE

identifier = letter (letter | digit)*

Dance DFA & NFA
from this

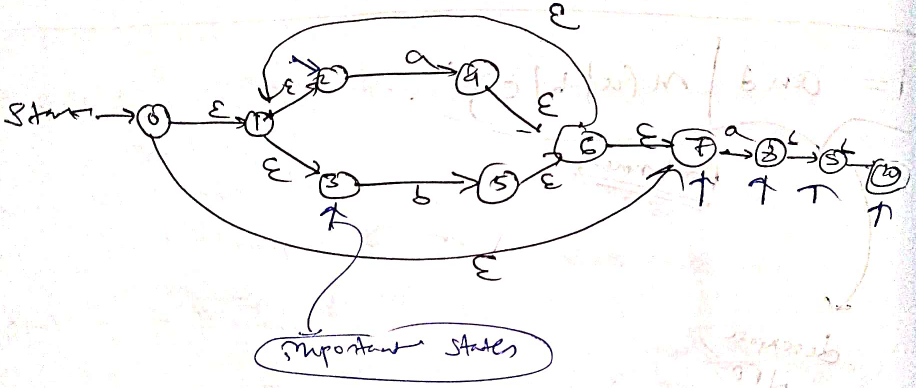
Minimization of a DFA

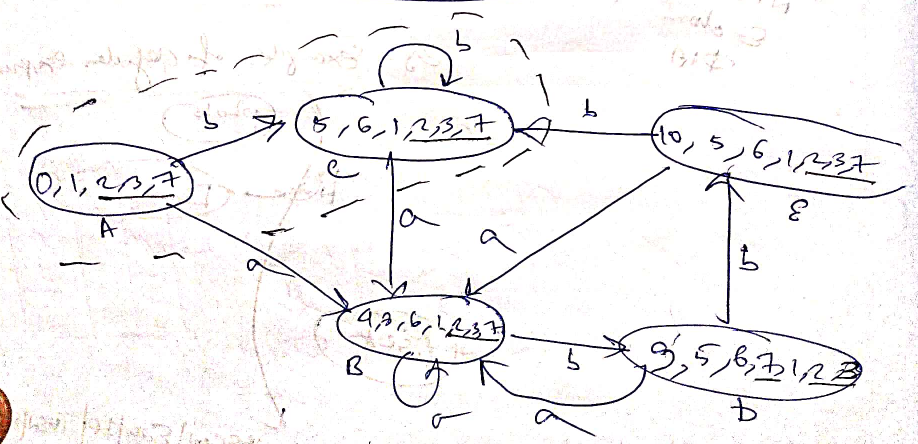
Important states of a NFA is a state for which a non ε transition exists.

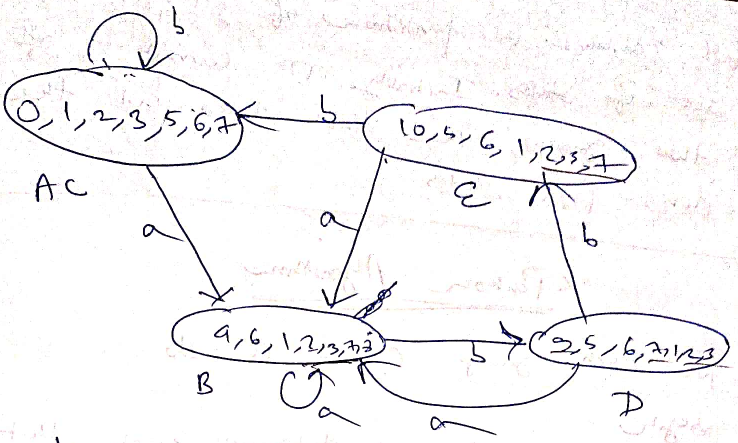
Therefore two states of DFA can be identified as single state when

(i) They have same important states

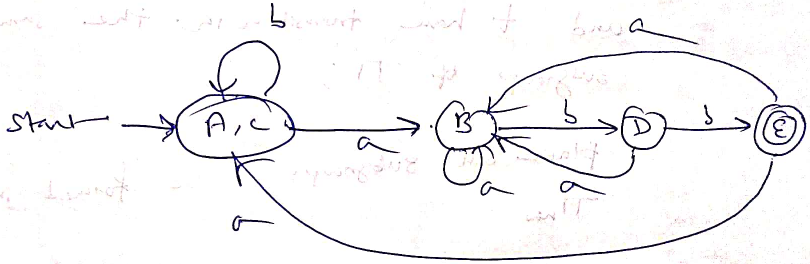
(ii) They either both include or both exclude the accepting states of the corresponding NFA.







∴ Minimised DFA



A and C are merged because they contain same important states.

We construct a partition of Π of the set of states. Initially Π consists of two groups, the final states and the non-final states.

Partition Algorithm

for each group G of Π do

begin

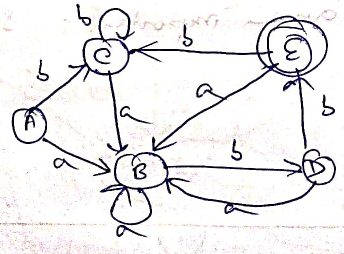
partition G into subgroups such that two states s and t of G are in same subgroup if and only if for all input symbol 'a' states s and t have transition in the same subgroup of Π;

place all subgroups so formed in Π_{new}

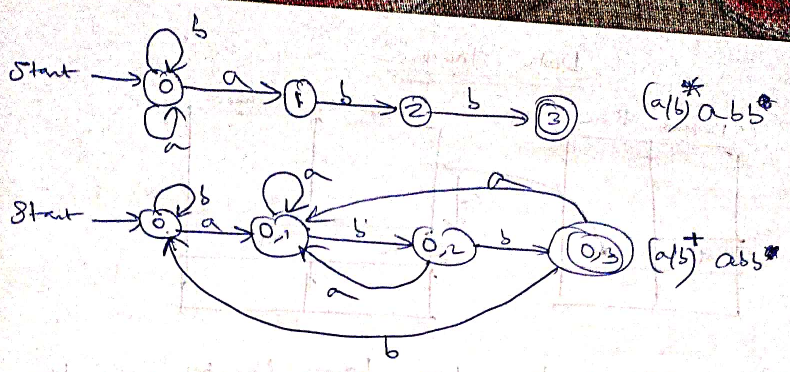
end.

Example DFA

State	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



$\Pi_0 = (ABCD)(E)$



$\Pi_1 = (A B C D)(E)$

Transition on 'a': $A \to B$, $B \to B$, $C \to B$, $D \to B$

" " 'b': $A \to C$, $B \to D$, $C \to C$, $D \to E$

$\Pi_2 = (A B C)(D)(E)$

Transition on 'a': $A \to B$, $B \to B$, $C \to B$

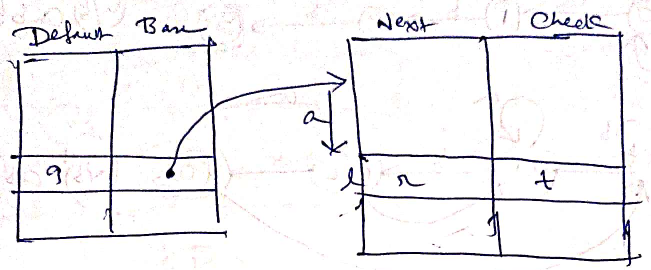
" " 'b': $A \to C$, $B \to D$, $C \to C$

$\Pi_3 = (A C)(B)(D)(E)$

Transition on 'a': $A \to B$, $C \to B$

" " 'b': $A \to C$, $C \to C$

Data Structure



BASE array stores the base location of the entries for the states in NEXT and CHECK arrays. DEFAULT array gives alternative base location in case the previous base location is invalid.

$l = BASE[s] + a$ if $CHECK[l] = s$ then $NEXT[l]$ is the next state for s on input a. If not then find $q = DEFAULT[s]$ and repeat the entire procedure recursively, using q in place of s.

Syntactic Specification of a Programming language

Context free grammar

Grammar

A grammar gives a precise, yet easy to understand, syntactic specification for the programs of a particular language.

Context free grammar ⟷ BACKER NAUR FORM

Syntactic analysis

Parser

An efficient parser can be constructed automatically from a properly designed grammar. Certain parser construction procedures can reveal syntactic ambiguous and other difficult to parse constructs which might otherwise go undetected in the initial design phase of a language and its compiler.

A C compiler has been designed using C language.

A grammar imparts structure to a program that is useful for its translation into object codes and for detection of errors.

Context free grammar

⇒ If S_1 and S_2 are statements and E is an expression, then "if E then S_1 else S_2" is a statement. — (I)

⇒ If $S_1, S_2, \ldots, S_n$ are statements then "begin $S_1, S_2, \ldots, S_n$ end" is a statement. — (II)

⇒ If E_1 and E_2 are expressions then $E_1 + E_2$ is an expression — (III)

⇒ (I) can be expressed as
Statement → if <expression> then <statement> else <statement> — (IV)

Backus Naur form

⇒ (3) Can be written as
<expression> → <expression> + <expression> — (V)

⇒ (2) can be written as
<statement> → begin < Statement>, <statement>, ...,
<statement> end.

⇒ We require the grammar rule should contain known number of symbols.

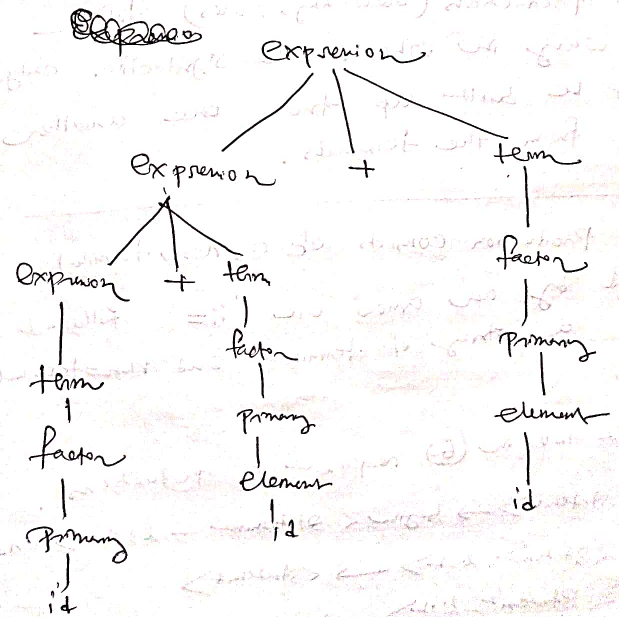
⇒ Introduce a new syntactic item called <statement_list>
<statement> → begin < statement_list> end
<statement_list> → <statement> | <statement> ; <statement_list> — (VI)

⇒ Vertical bar means 'or'

⇒ A <Statement_list> is either a <Statement> or a <Statement_list> followed by a semicolon followed by a <Statement>

⇒ The set of rules (iv), (v), (vi) are examples of grammar.

⇒ Grammar involves four quantities — terminals, nonterminals, a start symbol, productions.



⇒ The basic symbols of which strings in the language are composed is known as terminals.

(denoted by uppercase (T)) eg. begin, end, else, then

⇒ non terminals are special symbols that denote sets of strings.

(denoted by lowercase (t)) eg. <statement>, <statement_list>, <expression>

⇒ One non terminal is selected as the start symbol and it denotes the language in which we are truly interested.

⇒ The productions (rewriting rules) define the ways in which the syntactic categories may be built up from one another and from the terminals.

⇒ Each production consists of a non terminal followed by an arrow or "::=" followed by a string of terminals and non-terminals.

(How productions are written)

The rule in (6) represent 3 productions.

<statement> → begin <statement_list> end

<statement_list> → <statement>

<statement_list> → <statement> ; <statement_list>

---

eg. Non-terminals: - <expression>, <operator>

Start symbol: <expression>

terminals: id, +, −, *, /, ↑, (,)

$S \rightarrow i\ C\ t\ S\ S' \mid a$ (i: if, t: then, a: statement)

$S' \rightarrow es \mid a$ (e: else, a: statement)

This is a grammar.
It has two Productions.

Capitals = S, S', C → Non-terminals
Small = i, t, a, e → terminals

The productions are

<expression> → <expression> <operator> <expression>
(ε: produces; A: <operator>)

$E \rightarrow E\ A\ E$

<expression> → (<expression>)
$E \rightarrow (E)$

<expression> → − <expression>
$E \rightarrow -E$

<expression> → id
$E \rightarrow id$

<operator> → +
<operator> → −
<operator> → *
<operator> → /
<operator> → ↑

<expression> → <expression> <operator> <expression>
| (<expression>)
| − <expression>
| id

$E \rightarrow E\ A\ E \mid \varepsilon \mid -\varepsilon \mid id$
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$ (exponential)

Terminals

⇒ Single lowercase letter a, b, c

⇒ Operator symbols such as +, −, etc.

⇒ punctuation symbols such as parenthesis, comma etc

⇒ the digits 0, 1, ... 9

⇒ boldface strings such as id or if

$E \longrightarrow E A E$

$E \longrightarrow (E)$

$E \longrightarrow \overset{\text{unary minus}}{-} E$

$E \longrightarrow \underset{\text{identifier}}{id}$

$A \longrightarrow + \mid - \mid * \mid / \mid \uparrow$ (↑ : exponential)

Total 9 Productions

Nonterminals

⇒ lowercase names such as expression, statement, operator etc.

⇒ Italic capital letters near the beginning of the alphabet

⇒ The letter S, which, when it appears is usually the start symbol

---

⇒ if not ended in terminal then incomplete

⇒ Capital symbol near the end of the alphabet chiefly X, Y, Z represent grammar symbol that is either non-terminal or terminal

⇒ Small letters near the end of the alphabet chiefly u, v, ..., z represent terminals.

⇒ Lowercase Greek letters α, β, γ, for example, represent strings of grammar symbols.

Thus generic production could appear as $A \to \alpha$ indicating that there is a single non-terminal A on left of arrow and a string grammar symbol α to the right of arrow.

⇒ if $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$, then we may write

$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$

⇒ Unless otherwise stated the left side of the first production is the start symbol.

eg $E \to EAE$

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

E-production

A-production

E and A are non-terminals and remaining are terminals

⇒ ~~$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$~~

⇒ $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

$-E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

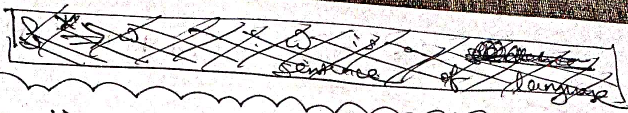
This is the derivation of $-(id)$ from E and provides proof that one single instance of an expression is $-(id)$

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production and α and β are arbitrary string of grammar symbols.
- If $\alpha_1 \Rightarrow \alpha_2, \Rightarrow \dots \Rightarrow \alpha_n$, we can say α_1 derives α_n
- The symbol $\overset{*}{\Rightarrow}$ means "derives in zero or more steps"
- The symbol $\overset{+}{\Rightarrow}$ means "derives in one or more steps"
- Given CFG G and start symbol S, we can use the relation $\overset{+}{\Rightarrow}$ to define $L(G)$, the language generated by G.

  Strings in $L(G)$ may contain only terminal symbols of G. We can say a string of terminals w is in $L(G)$ if $S \overset{+}{\Rightarrow} w$

  The string w is called a sentence of G. If $S \overset{*}{\Rightarrow} \alpha$, where α may contain nonterminals, we say α is a sentential form of G.

Language → Grammar → Production



if $S \Rightarrow^* w$ then w is a sentence of language

but if $S \not\Rightarrow^* w$ then w is not a sentence as it contains nonterminals w is a sentential form of language

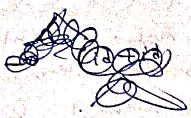
Parse Tree

⇒ A graphical representation of the derivations that filters out the choice regarding replacement order is called parse tree.

⇒ Each interior node of the parse tree is labelled by some nonterminal A and the children node are labelled from left to right by the symbols in the right side of production by which this A was replaced in derivation.

eg. $A \rightarrow XYZ$ is a production used at some step of a derivation then the parse tree for the derivation will have the subtree.

Parse tree for $-(id+id)$



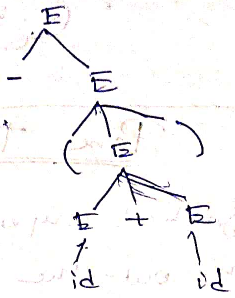
$E \Rightarrow -E$

$\Rightarrow -(E)$

$\Rightarrow -(E+E)$

$\Rightarrow -(id+E)$

$\Rightarrow -(id+id)$



$id + id * id$

$E \Rightarrow E+E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

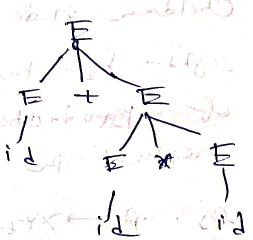
$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$



OK

$E \Rightarrow E*E$

$\Rightarrow E*id$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$

$3+5*7 = 3+35 = 38$

$3+5*7$

$= 8*7$

$= 56$

Operator precedence

- (Unary minus)
↑
* /
+ −

Associativity relation

$a \uparrow b \uparrow c = a \uparrow (b \uparrow c)$

$a - b - c = (a - b) - c$

$a + -b \uparrow c + d * e$

$= (a + ((-b) \uparrow c)) + (d * e)$

<element> → (<expression>) | id

<primary> → − <primary> | <element>

<factor> → <primary> ↑ <factor> | <primary>

<term> → <term> * <factor>
| <term> / <factor>
| <factor>

<expression> → <expression> + <term>
| <expression> − <term>
| <term>

Production

<expression> → <expression> + <term>
| <expression> − <term>
| <term>

<term> → <term> * <factor>
| <term> / <factor>
| <factor>

<factor> → <primary> ↑ <factor> | <primary>

<primary> → − <primary> | <element>

<element> → (<expression>) | id

<expression> ⇒ <expression> + <term>

⇒ <term> + <term>

⇒ <factor> + <term>

⇒ <primary> + <term>

⇒ <elem> + <term>

⇒ id + <term>

⇒ id + <term> * <factor>

⇒ id + <factor> * <factor>

⇒ id + <elem> * <factor>

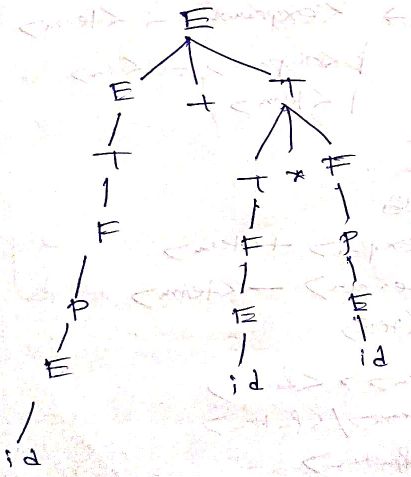
⇒ id + id * <factor>

⇒ id + id * <primary>

⇒ id + id * <elem>

⇒ id + id * id

Derivation for arithmetic string



E ⇒ E + T

⇒ T + T

⇒ F + T

⇒ P + T

⇒ E + T

⇒ id + T

⇒ id + F * F

⇒ id + F * F

⇒ id + E * F

⇒ id + id * P

⇒ id + id * E

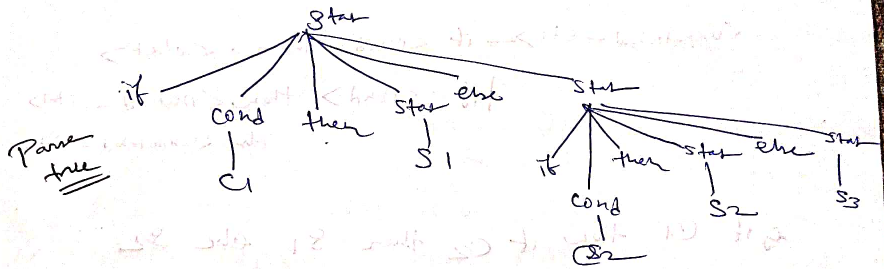
⇒ id + id * id

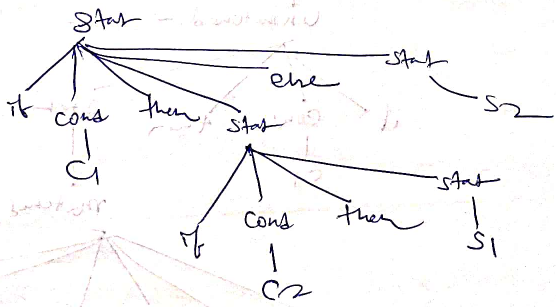
<stat> ⟶ if <cond> then <stat>
| if <cond> then <stat> else <stat>
| <other stat>

Prodⁿ for logical expⁿ

eg. if C1 then S1
else if C2 then S2 else S3

Parse tree





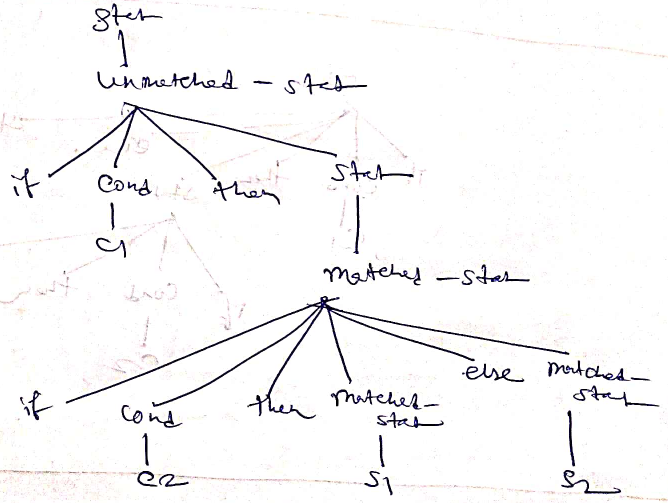
Unambiguous grammar

<stat> → <matched – stat>
<unmatched – stat>

<matched – stat> → if <cond> then <matched – stat>
else <matched – stat>
| other – stat>

<unmatched – stat> → if <cond> then <stat>
| if <cond> then <matched – stat>
else <unmatched – stat>

eg if C_1 then if C_2 then S_1 else S_2



Ch – Capabilities of context free grammar

(c) $L_2 = \{a^n b^m c^n d^m \mid n \geq 1$ and $m \geq 1\}$

It is not a context-free grammar because its NFA can't be obtained

L_2 is embedded in languages which requires that procedures be declared with the same number of formal parameters as there are actual parameters in their use is a^n and b^m could represent the formal parameter lists in two procedures declared to have m and n arguments respectively. Then c^n and d^m represent the actual parameter used in calls to these two procedures.

Again notice that typical syntax of procedure definitions and uses does not concern with self with counting the number of parameters.

Statement → call id (expression list)

Expression list → expression | expression list

If $a^n b^n c^n$, $n \geq 0$ then in string a, b, c we have equal no. of a, b, c that are not at all context free

⇒ Discuss the non-context free language constructs

⇒ There are three langs L_1, L_2, L_3. In this context three langs may be referred. Finite automata cannot count $L_1 \cap L_2 \cap L_3$

(a) left most & right most derivation → 3m
4+9

(b) Assignment

(c) How many right most derivation possible?

Grammar { S→ A→ B→

→ finite automata is used to find tokens in lexical analysis

Trace (actions)

{ What is handle?
What is handle pruning?

Give reasons for grammar to be unambiguous?

Operator precedence grammar

→ Shift reduce parsing

Grammar:
S → aACBe
A → Ab | b
B → d

1 prod
2 prod
3 prod

Consider a string "abbcde"

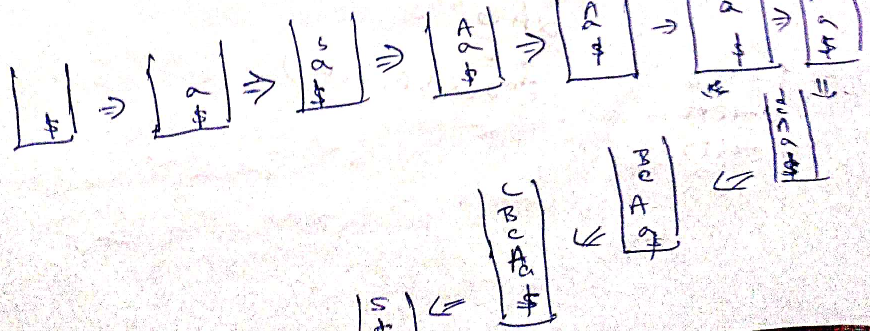
$

bottom up parsing (string to start symbol)

Stack	Input	Action
$	abbcde$	shift
$a	bbcde$	shift
$ab	bcde$	reduce A → b
$aA	bcde$	shift
$aAb	cde$	reduce A → Ab handle
$aA	cde$	shift
$aAc	de$	shift
$aAcd	e$	reduce B → d
$aAcB	e$	shift
$aAcBe	$	reduce S → aACBe
$S	$	accept

trace (actions)

DS used = stack
to obtain = string



Example Grammar

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

Consider a string "abbcde"

Right most derivation

$S \Rightarrow aAcBe$

$\Rightarrow aAcde$

$\Rightarrow aAbcde$

$\Rightarrow abbcde$

Left most Derivation

$S \Rightarrow aAcBe$

$\Rightarrow aAbcBe$

$\Rightarrow abbcBe$

$\Rightarrow abbcde$

Def of handle

(i) If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$ (* denotes zero or more steps; rm = right most) then $A \rightarrow \beta$ in the pos^n following α is a handle of $\alpha\beta w$.

(ii) If the Grammar is unambiguous then every right sentential form of the Grammar has exactly one handle.

αAw (A: non terminal)

string (α, β, A, S, w)

Eg: $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Consider the Right most derivation

$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E * id$ (Handle)
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

$E \Rightarrow E * E$
$\Rightarrow E * id$
$\Rightarrow E + E * id$ (Handle)
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

identical right sentential form

Right Sentential form	Handle	Reducing product
$id_1 + id_2 * id_3$	id_1	$E \to id_1$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

⇒ Ambiguity problem can be solved using a grammar called operator grammar

⇒ Operator grammar

Defn: A grammar which has no production whose right side is ε or has two adjacent non-terminals.

Original grammar

$E \to EAE \mid (E) \mid -E \mid id$
$A \to + \mid - \mid * \mid / \mid \uparrow$

Operator grammar

$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$

Representation of precedence relation

We represent the following three relational operators to represent precedence relations

⇒ $A \doteq B$ means A has higher precedence than B

⇒ $A \lessdot B$ means A has lower precedence than B

⇒ $A \doteq B$ means A and B have same Precedence

Rules for finding operator precedence relation

⇒ (i) If operator θ_1 has higher precedence than θ_2 then $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$

⇒ (ii) If θ_1 and θ_2 have same precedence then if the operators are left associative then

$\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$ and if they are right associative then

$\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$

⇒ (iii) $\theta \lessdot id$, $id \gtrdot \theta$,

$\theta \lessdot ($, $(\lessdot \theta$,

$) \gtrdot \theta$, $\theta \gtrdot)$,

$\theta \gtrdot \$$ and $\$ \lessdot \theta$

Also,

$(\doteq)$, $\$ \lessdot ($, $\$ \lessdot id$, $(\lessdot ($,

$id \gtrdot \$$, $) \gtrdot \$$, $(\lessdot id$, $id \gtrdot)$,

$) \gtrdot)$

⇒ If operator O_1 has higher precedence than O_2, then $O_1 \cdot > O_2$ and $O_2 < \cdot O_1$.

e.g. $* \cdot > +$ and $+ < \cdot *$

These relations ~~Assume~~ Presume that in an expression of the form $E+E*E+E$, the central $E*E$ is the handle that will be reduced first.

---

	id	+	*	$
id		$\cdot >$	$\cdot >$	$\cdot >$
+	$< \cdot$	$\cdot >$	$< \cdot$	$\cdot >$
*	$< \cdot$	$\cdot >$	$\cdot >$	$\cdot >$
$	$< \cdot$	$< \cdot$	$< \cdot$	

~~$ $ $.~~

~~$ id + id * id $~~

$ id + id * id $

$ <· E + E ·> $

Parser Actions

- Shift
- Reduce
- Accept
- Report Error

⇒ In a shift action the next input symbol is shifted to the top of the stack.

⇒ In a reduce action the parser knows that the right end of the handle is at the top of the stack. It must locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.

⇒ In an accept action, the parser announces successful completion of parsing.

⇒ In an error action, the parser discovers that a syntax error has occured and calls an error recovery routine.

repeat forever
if only $ is on the stack and only $ is on the input then accept and break
else begin
let a be the topmost terminal symbol on the stack and let b be the current input symbol.
if a <· b or a ≐ b then shift b on the stack
else if a ·> b then
repeat
pop the stack until top stack terminal is related by <· to the terminal most recently popped;
Store each Popped item in sequence and reduce else call the error correcting routine;
end.

Operator precedence parsing

Stack	Relation	Input	Action
$	<·	id + id * id $	Shift
$ id	·>	+ id * id $	reduce E→id
$ E	<·	+ id * id $	Shift
$ E +	<·	id * id $	Shift
$ E + id	·>	* id $	reduce E→id
$ E + E	<·	* id $	Shift
$ E + E *	<·	id $	Shift
$ E + E * id	·>	$	reduce E→id
$ E + E * E	·>	$	reduce E→E*E
$ E + E	·>	$	reduce E→E+E
$ E		$	accept

Stack	Relation	Input	Action
$	<	id * id + id $	Shift
$ id	>	* id + id $	reduce E → id
$ E	<	* id + id $	Shift
$ E *	<	id + id $	Shift
$ E * id	>	+ id $	reduce E → id
$ E * E	>	+ id $	Reduce E → E*E
$ E	<	+ id $	Shift
$ E +	<	id $	Shift
$ E + id	>	$	Reduce E → id
$ E + E	>	$	Reduce E → E+E
$ E		$	Accept

Operator Precedence Grammar

⇒ A grammar which has no production whose right side is ε or has two adjacent nonterminals is an operator grammar.

⇒ Operator precedence grammar is an operator grammar in which precedence relations $<\cdot$ $\doteq$ and $\cdot>$ are disjoint.

Rules for finding operator precedence relation

(1) If $S \to \alpha a \beta b \gamma$ where β is either single nonterminal or ε then $a \doteq b$

$S \to i C t S$
$S \to i C t S e S$
$S \to a$
$S \to b$

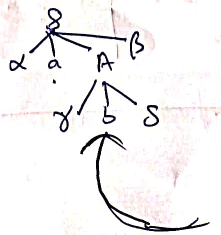
$b \to i$ → because b has C (beginning of i)
$t <\cdot a$

$S \to i C t S e S$
$i \doteq t$, $t \doteq i$

(2) If $S \to \alpha a A \beta$ and $A \overset{+}{\Rightarrow} \gamma b S$ such that γ is either ε or single nonterminal then $a <\cdot b$, also $\$ <\cdot b$ if $S \overset{+}{\Rightarrow} \gamma b S$ where γ is either ε or a single nonterminal.

$S \to i C t S$ and $C \overset{+}{\Rightarrow} b$ so $i <\cdot b$

$S \to i C t S$
$S \to i C t S e S$
$S \to a$
$S \to b$



γbS should be reduced first to A and then αAβ should be reduced to S
Hence $a <\cdot b$

b is the leftmost terminal of a handle

Let the grammar be

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

Consider the production

$E \rightarrow E+E$ which is the form $A \rightarrow \alpha a A \beta$

where $\alpha = E$, $a = +$, $A = E$, $\beta = E$

Again $E \Rightarrow E+E$ which is of the form

$\gamma b \delta$, where $\gamma = E$, $b = +$ and $\delta = E$

Then by rule (2) $+ \lessdot +$

Here $E \rightarrow E+E$ stands for $A \rightarrow \alpha A b \beta$

where $\alpha = \varepsilon$, $A = E$, $b = +$, $\beta = E$

and $E \Rightarrow E+E$ stands for

$A \Rightarrow \gamma a \delta$

where $\gamma = E$, $a = +$, $\delta = E$

then by rule (3)

$+ \gtrdot +$

Hence the above grammar is not an Operator precedence grammar.

(6)

Grammar

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

⇒ All derivations from F will have id as the first symbol or id as last symbol.

⇒ A derivation from T could begin $T \Rightarrow T*F$ showing that * could be the first and last terminal

⇒ A derivation could begin $T \Rightarrow F$ meaning that every first and last terminal of F is also a first or last terminal derivable from T. Thus *, (and id can be first and *,) and id can be last in derivation of T.

⇒ (, +, *, (or id can be first and +, *,) or id can be last in derivation of E

Non terminal	First terminal	Last terminal
E	+, *, (, id	+, *,), id
T	*, (, id	*,), id
F	(, id	), id

Leading(A)

Leading(A) = {a | $A \Rightarrow \gamma a \delta$ where γ is ε or single nonterminal}

Concept for computing computation of LEADING(A):-

(i) a is in LEADING(A) if $A \rightarrow \gamma a \delta$ where γ is ε or a non terminal

(ii) a is in LEADING(A) if $A \rightarrow B\alpha$ and a is in LEADING(B)

```
Procedure Insll(A,B)
if not L[A,a] then
begin
    L[A,a] := true;
    Push (A,a) onto Stack
end;
```

```
begin
for ∀A, a   L[A,a] := false
for each production of the form
A → aα or A → Baα do Insll(A,a);
While stack is not empty do
begin
    pop top pair (B,a) from Stack;
    for each A → Bα do Insll(A,a);
end;
end;
```

Grammar

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

$Leading(E) = \{+, id, (, *\}$ (from F)
$Leading(T) = \{*, id, (\}$ (from F)
$Leading(F) = \{(, id\}$

(F, id)
(E, ()
(T, *)
(E, +)

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

Rules for finding operator precedence relation

(1) If $A \rightarrow \alpha a \beta b \gamma$, where β is either a single nonterminal or ϵ then $a \doteq b$

Here a and b are terminal symbols which are considered as operators. The production rules suggest that the string "$\alpha a \beta b \gamma$" is reduced to S in one step therefore there must not be any "do it before" relation between them. Hence they must have equal precedence ie, $a \doteq b$

(iv) If $S \rightarrow \alpha A a \beta$ and $A \Rightarrow^{+} \gamma b \delta$ such that γ is either ε or a single non-terminal then $a < \cdot b$ also $\$ < \cdot b$ if $S \Rightarrow^{+} \gamma b \delta$ where γ is either ε or a single non terminal. b is in Leading A.

(v) If $S \rightarrow \alpha A b \beta$ and $A \Rightarrow^{+} \gamma a \delta$ where δ is either ε or a single nonterminal then $a \doteq b$, also $a \doteq \$$ if $S \Rightarrow^{+} \gamma a \delta$ where δ is either ε or a single nonterminal a is in Trailing A.

Baap ~~Baap~~ ke presence be dacche
thi baap ban jaate hai

⇒ Concept for finding operator precedence

In a production of the form $A \to x_1 x_2 \ldots x_i x_{i+1} \ldots x_n$

(i) If x_i and x_{i+1} are terminals then set $x_i \doteq x_{i+1}$ (rule 1)

(ii) If x_i and x_{i+2} are terminals and x_{i+1} is a nonterminal then set $x_i \doteq x_{i+2}$ (rule 1)

(iii) If x_i is a terminal and x_{i+1} is a ~~terminal~~ nonterminal then for all a, a is in ~~[illegible]~~ LEADING(x_{i+1}) set $x_i < a$ (rule 2)

(iv) If x_i is a nonterminal and x_{i+1} is a terminal then for all a a is in TRAILING(x_i); set $a > x_{i+1}$ (rule 3)

(v) Set $\$ < a$ for all a in LEADING(S) and set $b > \$$ for all b in TRAILING(S) (rule 2 and 3)

$S \to aAcBe$
$A \to Ab$
$A \to b$
$B \to d$

$a \doteq e$; $c \doteq e$
$a < b$; $e < d$
$a > c$; $d > e$
$b > c$; $d > e$
$b > b$

LEADING(S) = {a}
LEADING(A) = {A, b}
LEADING(B) = {d}

TRAILING(S) = {e}
TRAILING(A) = {A, b}
TRAILING(B) = {d}

Automata — Backtracking Parser

Work down also and write procedures for this

Eg.

$S \rightarrow aACBe$

$A \rightarrow Ab$

$A \rightarrow b$

$B \rightarrow d$

	a	b	c	d	e	$
a		$\lessdot$	$\doteq$			
b		$\gtrdot$	$\gtrdot$			
c				$\lessdot$	$\doteq$	
d					$\gtrdot$	
e						$\gtrdot$
$	$\lessdot$					

LEADING(S) = {a}

LEADING(A) = {b}

LEADING(B) = {d}

TRAILING(S) = {e}

TRAILING(A) = {b}

TRAILING(B) = {d}

⇒ Write down an algorithm Computing Operator Precedence relationship.

*** 4m

(2)

$S \rightarrow aAcBe$

$A \rightarrow Ab \mid b$

$B \rightarrow d$

Consider "abbcde"

	a	b	c	d	e	$
a	—	$\doteq$	$\doteq$			
b		$>$	$>$			
c				$<\cdot$	$\doteq$	
d					$>$	
e						$>$
$	$<\cdot$					

Ans.

Stack		Input	
$	$<\cdot$	abbcde$	Shift
$a	$<\cdot$	bbcde$	Shift
$ab	$>$	bcde$	Reduce A→b
$aA	$<\cdot$	bcde$	Shift
$aAb	$>$	cde$	Reduce A→Ab
$aA	$\doteq$	cde$	Shift
$aAc	$<\cdot$	de$	Shift
$aAcd	$>$	e$	Reduce B→d
$aAcB	$\doteq$	e$	Shift
$aAcBe	$>$	$	reduce S→aAcBe
$S			accept

Q

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

LEADING(E) = {+, *, (, id}
LEADING(T) = {*, (, id}
LEADING(F) = {(, id}
TRAILING(E) = {+, *,), id}
TRAILING(T) = {*,), id}
TRAILING(F) = {), id}

Write down the operator precedence table by finding leading & trailing first for the above operator precedence grammar

Assignment

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	
)	⋗	⋗		⋗		⋗
id	⋗	⋗		⋗		⋗
$	⋖	⋖	⋖		⋖	

Operator Precedence Parsing

(i) (ii) (iii) (iv) (v)

No:5

$+ \gtrdot \$$
$* \gtrdot \$$
$id \gtrdot \$$
$) \gtrdot \$$

$\$ \lessdot +$
$+ \lessdot +$
$+ \lessdot id$
$\$ \lessdot ($

$\$ \lessdot$ LEADING(S)
and TRAILING(S) $\gtrdot \$$

Starting Symbol

No case

None

$(a_i \doteq a_{i+1} \in T)$
$a_i \doteq a_{i+1}$

$F \to (E)$

$(\doteq)$

$a_i, a_{i+2} \in T$
$a_{i+1} \in NT$
$\to F \to (E)$

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

$(\lessdot +, (\lessdot *, (\lessdot id,$
$+ \lessdot (, * \lessdot (, + \lessdot id, + \lessdot ($
$* \lessdot id, * \lessdot ($

$a_i \in T, a_{i+1} \in NT$
$a_i \lessdot$ LEADING(a_{i+1})

$a_i \in T$ TRAILING(a_{i+1})
$a_{i+1} \in$

F → (T) { *>), *id>),)>> }
E → E+T { +>+, +*>+, +id>+,)>+ }
T → T*F { *>*, id>*,)>* }

∀i: ∈NT
∀a ∈ T
Table (ai) > ai+1

~~Operator~~ Grammar → Operator Grammar → Leading, Trailing

Operator Precedence Relation

(i) $ <· LEADING(S)
TRAILING(S) ·> $

(ii) ai, ai+1 ∈ T
· ai ≐ ai+1

(iii) ai, ai+2 ∈ T
ai+1 ∈ NT
∴ ai ≐ ai+2

(iv) ai ∈ T
ai+1 ∈ NT
∴ ai <· LEADING(ai+1)

(v) ai ∈ NT
ai+1 ∈ T
∴ TRAILING(ai) ·> ai+1

(i) A → γaβ
LEADING(A) = {a}
if γ is ε or NT

(ii) if A → Bα
and ~~LEADING(A)~~
LEADING(A) = {a}
then LEADING(B) = {a}

(i) A → γaβ
TRAILING(A) = {a}
if β is ε or NT

(ii) if A → αB
and TRAILING(A) = {a}
then TRAILING(B) = {a}

⑧ (id + id) * id

parse this string and write down Tables

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Top Down Parsing

(Recursive - descent parsing)

(a) Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab$ → Starting symbol is non function

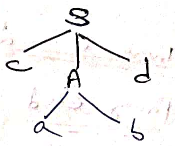
$A \rightarrow a$

Let the input string be cad

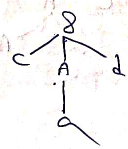
Example String: c a d

input c a d

Failure — back track



Match is obtained



Backtracking was necessary because we did not know we should expand ab or c

---

Left factoring

If $A \rightarrow \alpha\beta \mid \alpha\gamma$ are two A productions and the input begins with a non empty string derived from α we do not know whether to expand A to $\alpha\beta$ or to $\alpha\gamma$. We may defer the decision by expanding A to $\alpha A'$. Then after seeing the input derived from α we expand A' to β or to γ. That is the original grammar is left factored. After left factoring the original grammar we get

A prime

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

$S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

Changed Grammar

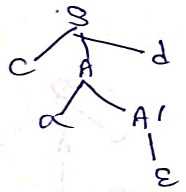
$S \rightarrow cAd$

$A \rightarrow aA'$

$A' \rightarrow b \mid \varepsilon$

Input c a d

Parse tree



$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

Changed Grammar

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

$A \rightarrow$ (common) A'

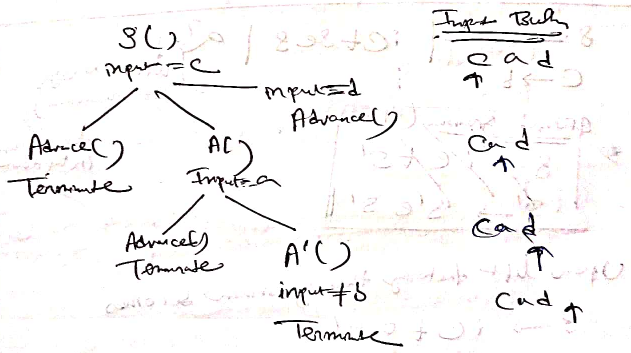
$A' \rightarrow$ (remaining) $\mid \varepsilon$

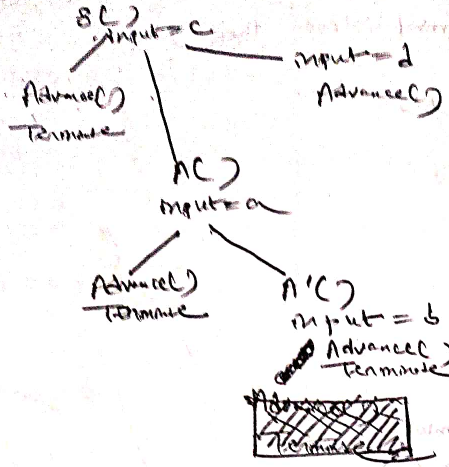
Recursive Descent Parsing

```
Procedure S( )
begin
    if input symbol = 'c' then
    begin
        Advance( );
        A( );
        if input symbol = 'd' then
        Advance( ) else error( ).
    end;
end;
```

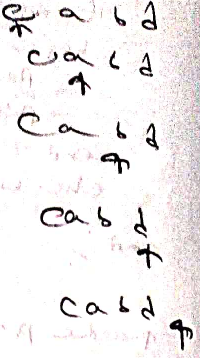
```
Procedure A()
begin
    if input symbol = 'a' then
    begin
        Advance();
        A'();
    end
    else error();
end;

Procedure A'()
begin
    if input symbol = 'b'
    then Advance();
end;
```





Input buffer



$S \rightarrow iCtS \mid iCtSeS \mid a$ — Left factoring, Left recursion — two problems in top-down

$C \rightarrow b$

Original Grammar (Upon left factoring)

$S \rightarrow iCtS'$

$S' \rightarrow SeS'$

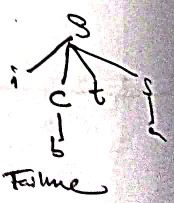
Upon left factoring the grammar becomes

$S \rightarrow iCtSS'$

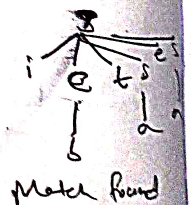
$S' \rightarrow eS \mid \epsilon$

$C \rightarrow b$

ibtaea

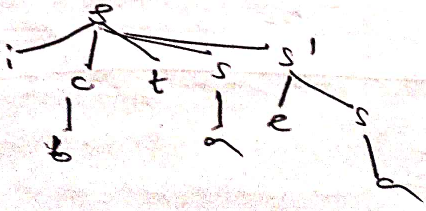


Failure



Match found

ibtaea



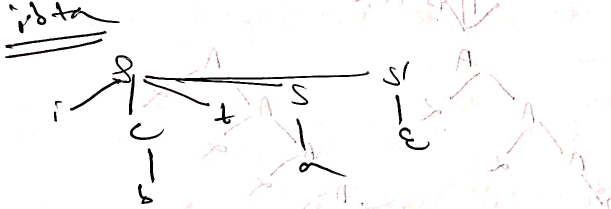
Assignment

④ Which of the grammar req. left factoring. How

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

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

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

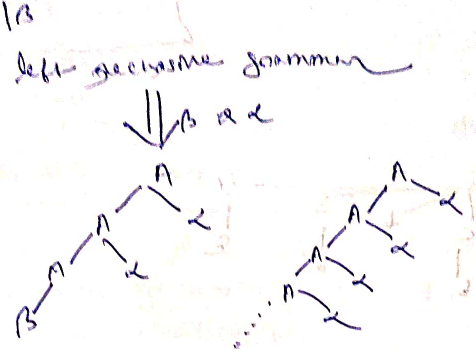


Left Recursion

Top-down → left ancestry
Bottom-up → data to right

Consider the grammar
$A \rightarrow A\alpha \mid \beta$

This is a left recursive grammar



This can be modified to

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

$A \rightarrow$ (non-recursive) A'
$A' \rightarrow$ (recursive) $A' \mid \varepsilon$

Method of Eliminating Immediate Left Recursion

Consider the grammar

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where β_i begins with A

we can eliminate left recursion as follows

$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

Non-immediate left recursion

Consider the grammar

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid e$

The non-terminal is left recursive because

$S \Rightarrow Aa \Rightarrow Sda$

The following algo will eliminate all left recursion if the grammar has no cycle i.e., it has a derivation of the form $A \Rightarrow^{+} A$ or has an ε production.

1. arrange the nonterminals of G in some order $A_1, A_2, \ldots, A_n$
2. for $i := 1$ to n do
   begin
   for $j := 1$ to $i-1$ do
   replace each production of the form $A_i \rightarrow A_j\gamma$ by the production $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
   where, $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current A_j productions.
   eliminate the immediate left recursion among the A_i productions.

   End;

Example grammar

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid e$

We order the non terminal S, A ie, $A_1 = S$ and $A_2 = A$

⇒ We do not have any production of the form

$S \rightarrow S\gamma$

⇒ Set $i = 2$, $j = 1$

⇒ Consider $A \rightarrow Sd$ we have following A1 productions available

$S \rightarrow Aa$ and $S \rightarrow b$

⇒ After replacing S in $A \rightarrow Sd$ we get

$A \rightarrow Aad$ and $A \rightarrow bd$

⇒ Next we eliminate immediate left recursion among each A production which are

$A \rightarrow Ac \mid Aad \mid bd \mid e$

⇒ The resulting productions are

$A \rightarrow bdA' \mid eA'$

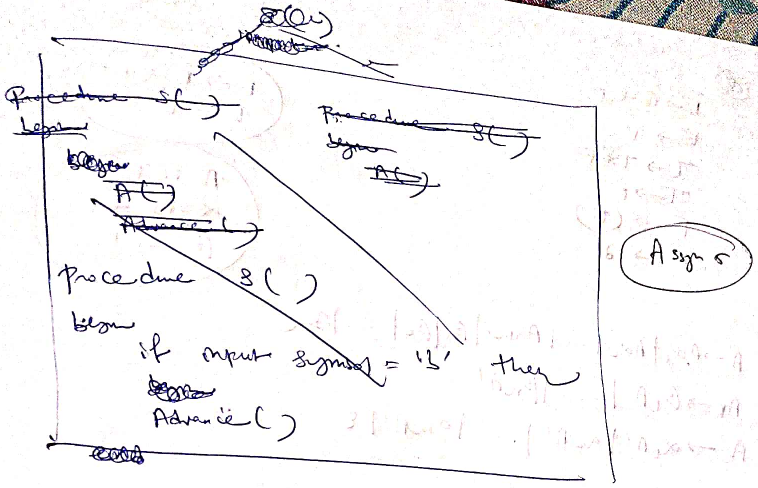
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

And the complete grammar is

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid eA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$




```
Procedure B( )
begin
    if input symbol = 'b'
    then Advance( )
    else
    begin
        A( )
        if input symbol = 'a'
        then Advance( )
    end
    else error( )
end
```

```
Procedure A( )
begin
    if input = 'b' then
    begin
        Advance( )
        if input = 'd' then
        begin
            Advance( )
            A'( )
        end
        else error( )
    end
    else if input = 'e'
    begin
        Advance( )
        A'( )
    end
    else error( )
end
```

```
Procedure A'( )
begin
if input = 'e' then
begin
    Advance( )
    A'( )
end
else
    if input = 'a' then
    begin if input = 'd' then
        begin
            Advance( )
        end A'( )
    else error( )
end
else error( )
end
```

(a)

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

$E \to E+T$
$E \to T$

A is E
α is +T
β is T

$A \to A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$

$A \to \beta_1 A' | \dots | \beta_n A'$

$A' \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \varepsilon$

Using the immediate left recursion

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$

```
Procedure E()
begin
    T();
    E'();
end;
```

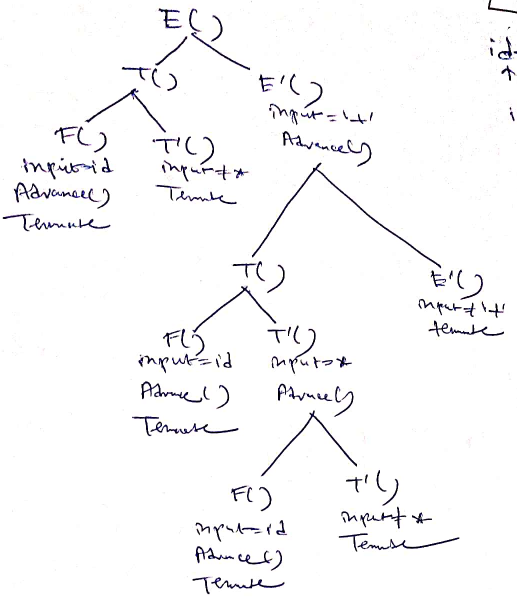
```
Procedure T()
begin
    F();
    T'();
end;
```

```
Procedure F()
begin
    if input='id' then
    Advance()
    else
    if input='(' then
    begin
        Advance();
        E();
        if input=')' then
        Advance();
        else error();
    end
    else error();
end;
```

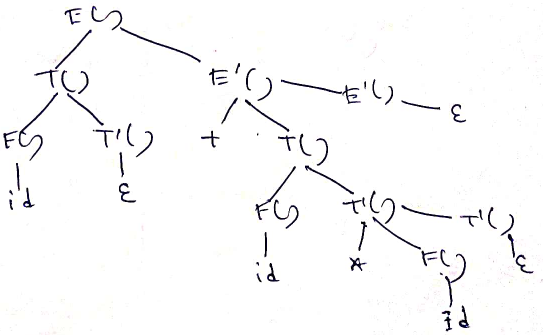
```
Procedure E'();
begin
    if input='+' then
    begin
        Advance()
        T()
        E'()
    end
end
```

```
Procedure T'()
begin
    if input='*' then
    begin
        Advance()
        F()
        T'()
    end
end
```

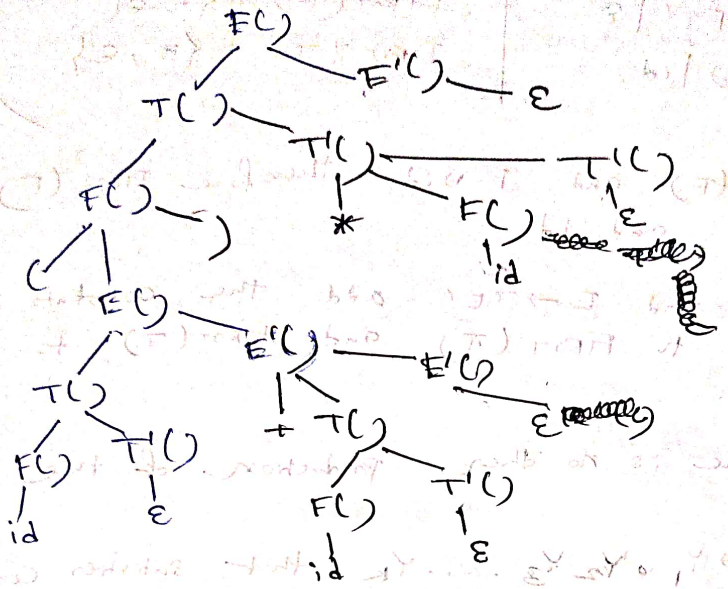
Input buffer id + id * id



id + id * id
↑
id + id * id
↑
id + id * id
↑
id + id * id
↑
id + id * id
↑
id + id * id
↑



(id + id) * id



FIRST

① If n is a terminal then FIRST(n) is $\{n\}$

② If X is a nonterminal and $X \rightarrow a\alpha$ is a production then add a to FIRST(X).
If $X \rightarrow \varepsilon$ then add ε to FIRST(X)

③ If $X \rightarrow Y_1 Y_2 \ldots Y_k$ then for all i such that all of $Y_1, Y_2, \ldots Y_{i-1}$ are nonterminal and FIRST(Y_1) contains ε for $j = 1, 2, \ldots, i-1$ add every non ε symbol in FIRST(Y_1) to FIRST(X).
If ε is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ then add ε to FIRST(X)

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

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

⇒ Since $F \to (E)$ and $F \to id$ therefore FIRST(F) includes (and id

$T \to FT'$ and $E \to TE'$ add the contents of FIRST(F) to FIRST(T) and FIRST(T) to FIRST(E).

⇒ Since there is no other production of the form

$X \to Y_1 Y_2 Y_3 \ldots Y_k$ that satisfies condition 3 hence no other symbol will be added to FIRST(E), FIRST(T) or FIRST(F)

⇒ It can be trivially shown by 2 that

FIRST(E') = {+, ε}, FIRST(T') = {*, ε}

FOLLOW

1. $ is in FOLLOW(S) where S is the start symbol
2. If $A \rightarrow \alpha B \beta$, $\beta \neq \varepsilon$, then every element in FIRST(β) but not ε is in FOLLOW(B)
3. If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε then every element in FOLLOW(A) is in FOLLOW(B)

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

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

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

⇒ $ must be in FOLLOW(E)

⇒ $F \rightarrow (E)$ is of the form $A \rightarrow \alpha B \beta$ where $\alpha = (, B = E, \beta =)$

⇒ FIRST(β) by definition is {) }

⇒ Therefore FOLLOW(E) should include)

⇒ There is no production containing E

Construction of predictive parsing table

1. For each production $A \to \alpha$ do step 2 and 3.
2. For each a in $FIRST(\alpha)$ add $A \to \alpha$ to $M[A, a]$
3. If ε is in $FIRST(\alpha)$ add $A \to \alpha$ to $M[A, b]$ for each terminal b in $FOLLOW(A)$. If ε is in $FIRST(\alpha)$ add $A \to \alpha$ to $M[A, \$]$
4. Make each undefined entry of M "error"

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

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

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

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

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

Stack	Input	Output
E	$id+id*id$	
$E'T$	$id+id*id$	$E \to TE'$
$E'Tid$	$id+id*id$	$T \to FT'$
$E'T'F$	$id+id*id$	$F \to id$
$E'T'id$	$id+id*id$	
$E'T'$	$+id*id$	$T' \to \varepsilon$
E'	$+id*id$	
$E'T+$	$+id*id$	$E' \to +TE'$
$E'T$	$id*id$	
$E'T'F$	$id*id$	$T \to FT'$
$E'T'id$	$id*id$	$F \to id$
$E'T'$	$*id$	
$E'T'F*$	$*id$	$T' \to *FT'$
$E'T'F$	id	
$E'T'id$	id	$F \to id$
$E'T'$	$\$$	$T' \to \varepsilon$
E'	$\$$	$E' \to \varepsilon$
$\$$	$\$$	Accept

$E \to TE'$	$M[E, id]$
$E \to +TE'$	$M[E', +]$
$E' \to \varepsilon$	$M[E', \$]$
$T \to FT'$	$M[T, id]$
$T' \to *FT'$	$M[T', *]$
$T' \to \varepsilon$	$M[T', +]$
$T' \to \varepsilon$	$M[T', \$]$
$F \to id$	$M[F, id]$
$F \to (E)$	$M[F, (]$

(pi+pi) * pi

Stack	Input	Output
$E	id*(id+id)$	
$E'T	id*(id+id)$	E → TE'
$E'T'F	id*(id+id)$	T → FT'
$E'T'id	id*(id+id)$	F → id
$E'T'	*(id+id)$	
$E'T'F*	*(id+id)$	T' → *FT'
$E'T'F	(id+id)$	
$E'T')E(	(id+id)$	F → (E)
$E'T')E	id+id)$	
$E'T')E'T	id+id)$	E → TE'
$E'T')E'T'F	id+id)$	T → FT'
$E'T')E'T'id	id+id)$	F → id
$E'T')E'T'	+id)$	
$E'T')E'	+id)$	T' → ε
$E'T')E'T+	+id)$	E' → +TE'
$E'T')E'T	id)$	
$E'T')E'T'F	id)$	T → FT'
$E'T')E'T'id	id)$	F → id
$E'T')E'T'	)$	T' → ε
$E'T')E'	)$	E' → ε
$E'T')	)$	
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	Accept

It not accepted then error

(pi+pi) * (pi * pi)

Stack	Input	Output
$	(pi+pi) * (pi*pi)	
$ (	pi+pi) * (pi*pi)	
$ (pi	+pi) * (pi*pi)	
$ (F	+pi) * (pi*pi)	F → pi
$ (T	+pi) * (pi*pi)	T → F
$ (E	+pi) * (pi*pi)	E → T
$ (E+	pi) * (pi*pi)	
$ (E+pi	) * (pi*pi)	F → pi
$ (E+F	) * (pi*pi)	T → F
$ (E+T	) * (pi*pi)	E → E+T
$ (E	) * (pi*pi)	F → (E)
$ F	* (pi*pi)	T → F
$ T	* (pi*pi)	
$ T*	(pi*pi)	
$ T*(	pi*pi)	
$ T*(F	*pi)	
$ T*(T	*pi)	
$ T*(T*	pi)	
$ T*(T*pi	)	F → pi
$ T*(T*F	)	T → F
$ T*(T	)	T → T*F
$ T*(E	)	E → T
$ T*(E)		F → (E)
$ F		T → F
$ T		E → T
$ E		

Algorithm

repeat
begin
let X be the top stack symbol and 'a' the next input symbol

if X is a terminal or $ then
if X = a then
pop X from the stack and remove 'a' from the input
else error();
else
if M[X, a] = X → Y1 Y2 Y3 ... Yn then
begin
pop X from the stack;
push Yn ... Y3 Y2 Y1 onto the stack
end
else
error()
end
Until X = $

$S \rightarrow iCtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

$\text{FIRST}(S) = \{i, a\}$

$\text{FIRST}(S') = \{e, \varepsilon\}$

$\text{FIRST}(C) = \{b\}$

$\text{FOLLOW}(S) = \{e, \$\}$

$\text{FOLLOW}(S') = \{e, \$\}$

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

$S \rightarrow iCtSS' \quad [iCt = \alpha, S = B, S' = \beta]$

∴ FOLLOW(S) should include $ and FIRST(S') other than ε should be included in FOLLOW(S)

	i	t	a	b	e	$
S	$S \rightarrow iCtSS'$		$S \rightarrow a$			
S'					$S' \rightarrow eS$ $S' \rightarrow \varepsilon$	$S' \rightarrow \varepsilon$
C				$C \rightarrow b$		

इसको निकाल दिया तो ये LL(1) बन जाएगा

LL(1) grammar

For $A \to \alpha | \beta$

1. $\neg \exists a$: FIRST(α) $\wedge$ FOLLOW(β)
2. Both FIRST(α) and FIRST(β) do not contain ε
3. If FIRST(β) contains ε then α does not derive any string beginning with a terminal in FOLLOW(A)

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$

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

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

FOLLOW(E) = {), $}

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

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

FOLLOW(E') = {), $}

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

LL(K) Parser

Meaning of LL(K)

⇒ First L of LL means — read input string left to right

⇒ Second L of LL means — produces leftmost derivation

⇒ K — number of lookahead symbols.

$$
\left\{
\begin{array}{l}
S \longrightarrow aAd \\
S \longrightarrow bAc \\
S \longrightarrow bBd \\
S \longrightarrow aBc \\
A \longrightarrow c \\
B \longrightarrow c
\end{array}
\right.
\Rightarrow
\left\{
\begin{array}{l}
1.\ S \longrightarrow aS' \\
2.\ S' \longrightarrow Ad \\
3.\ S' \longrightarrow Bc \\
4.\ S \longrightarrow bS'' \\
5.\ S'' \longrightarrow Ac \\
6.\ S'' \longrightarrow Bd \\
7.\ A \longrightarrow c \\
8.\ B \longrightarrow c
\end{array}
\right.
$$

FIRST(S) = {a, b}

FIRST(A) = {c}

FIRST(B) = {c}

FIRST(S') = {c}

FIRST(S'') = {c}

Possible strings

acd, bce, bcd, ace

LL(1) Parsing table

	a	b	c	d	e	$
S	1	4				
S'			2, 3			
S''			5, 6			
A			7			
B			8			

Stack	Input	Output
$ S	acd $	
$ S' a	acd $	S → a S'
$ S'	cd $	

↓

we can't even proceed further due to ambiguity

For $A \to \alpha | \beta$

1. $\neg \exists a : FIRST(\alpha) \wedge FIRST(\beta)$
2. Both FIRST(α) and FIRST(β) do not contain ε
3. If FIRST(β) contains ε then α does not derive any string beginning with a terminal in FOLLOW(A)

Consider

$S' \to Ad$ and $S' \to Bc$

FIRST(Ad) = {c} & FIRST(Bc) = {c}

Hence the grammar is not an LL(1) grammar

LL(2)

1. $S \rightarrow aS'$
2. $S' \rightarrow Ad$
3. $S' \rightarrow Be$
4. $S \rightarrow bS''$
5. $S'' \rightarrow Ae$
6. $S'' \rightarrow Bd$
7. $A \rightarrow c$
8. $B \rightarrow c$

FIRST(S) = {ac, bc}
FIRST(S') = {cd, ce}
FIRST(S'') = {ce, cd}
FIRST(A) = {c}
FIRST(B) = {c}

FIRST(aS') = {ac}
FIRST(bS'') = {bc}
FIRST(Ad) = {cd}
FIRST(Be) = {ce}
FIRST(Ae) = {ce}
FIRST(Bd) = {cd}
FIRST(c) = {c}

LL(2) parsing table

	ac	bc	bd	cd	ce	c
S	1	4				
S'				2	3	
S''				6	5	
A						7
B						8

LL(1) parsing

Stack	Input	Rule
$S	acd$	
$Sa	acd$	1
$S	cd$	
$dA	cd$	2
$dA	cd$	7
$dc	cd$	
$	$	

LL(1) parsing table

$S \rightarrow ACB$
$A \rightarrow aAb \mid ab$
$B \rightarrow aBb \mid acb$

FIRST(S) = {a}
FIRST(A) = {a}
FIRST(B) = {a}

FOLLOW(S) = {$}
FOLLOW(A) = {b, c}
FOLLOW(B) = {$, b}

	a	b	c	$
S	$S \rightarrow ACB$			
A	$A \rightarrow aAb$ $A \rightarrow ab$			
B	$B \rightarrow aBb$ $B \rightarrow acb$			

Left factoring

If $A \to \alpha\beta \mid \alpha\gamma$ are two A productions and the input begins with a non empty string derived from α, we do not know whether to expand A to $\alpha\beta$ or to $\alpha\gamma$. We may defer the decision by expanding A to $\alpha A'$. Then after seeing the input derived from α we expand A' to β or to γ. That is, the original grammar is left factored. After left factoring the original grammar we get

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

$S \to ACB$
$A \to aAb \mid ab$
$B \to aBB \mid a \mid \epsilon$

$S \to ACB$
$A \to aA'$
$A' \to As \mid b$
$B \to aB'$
$B' \to Bs \mid \epsilon$

FIRST (S) = {a}
FIRST (A) = {a}
FIRST (A') = {a, b}
FIRST (B) = {a}
FIRST (B') = {a, c}

FIRST (ACB) = {a}
FIRST (aA') = {a}
FIRST (As) = {a}
FIRST (b) = {b}
FIRST (aB') = {a}
FIRST (Bs) = {a}
FIRST (cs) = {c}

	a	b	c
S	$S \to .ACB$		
A	$A \to aA'$		
A'	$A' \to As$	$A' \to b$	
B	$B \to aB'$		
B'	$B' \to Bs$		$B' \to cs$

(Q) abcacb

Stack	Input	Output
$S	abcacb$	
$BCA	abcacb$	S → AcB
$BCA'a	abcacb$	A → aA'
$BCA'	bcacb$	
$BC	bcacb$	A' → b
$Bc	cacb$	
$B	acb$	
$B'a	acb$	B → aB'
$B'	cb$	
$B'c	cb$	B' → c
$	$	Accept

LL(1) parsing table

FIRST(S) = {aa, ab}
FIRST(A) = {aa, ab}
FIRST(B) = {aa, ac}
FIRST(AcB) = {aa, ab}
FIRST(aAb) = {aa}
FIRST(ab) = {ab}
FIRST(aBb) = {aa}
FIRST(acb) = {ac}

	aa	ab	ac
S	S → AcB	S → AcB	
A	A → aAb	A → ab	
B	B → aBb		B → acb

S → AcB
A → aAb | ab
B → aBb | acb

Stack	Input	Action
$ B	aaabbcacb$	
$ BcA	aaabbcacb$	S → AcB
$ BcbAa	aabbcacb$	A → aAb
$ BcbA	abbcacb$	
$ Bcbb	abbcacb$	A → ab
$ B	acb$	
$ bca	acb$	B → acb
$		

	aa	ab	ac
S	S → AcB	S → AcB	
A	A → aAb	A → ab	
B	B → aBb		B → acb

Ignore

Renewed Parser table for string id+id*id

	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)		

Matched	Stack	Input	Action
	$E	id+id*id$	
	$E'T	id+id*id$	E→TE'
	$E'T'F	id+id*id$	T→FT'
	$E'T'id	id+id*id$	F→id
	$E'T'	+id*id$	pop
	$E'	+id*id$	T'→ε
	$E'T+	+id*id$	E'→+TE'
	$E'T	id*id$	pop
	$E'T'F	id*id$	T→FT'
	$E'T'id	id*id$	F→id
	$E'T'	*id$	pop
	$E'T'F*	*id$	T'→*FT'
	$E'T'F	id$	pop
	$E'T'id	id$	F→id
	$E'T'	$	pop
	$E'	$	T'→ε
	$	$	E'→ε

Accept

Ignore

Not checked

ignore

Match

Stack	Input	Action
$E	$ id + id * id $	
$E' T	$ id + id * id $	E → TE'
$E' T' F	$ id + id * id $	T → FT'
$E' T' id	$ id + id * id $	F → id
$E' T'	$ + id * id $	match id
$E'	$ + id * id $	T' → ε
$E' T +	$ + id * id $	E' → +TE'
$E' T	$ id * id $	match +
$E' T' F	$ id * id $	T → FT'
$E' T' id	$ id * id $	F → id
$E' T'	$ * id $	match id
$E' F *	$ * id $	T' → *FT'
$E' F	$ id $	match *
$E' id	$ id $	F → id
$E'	$	match id
$	$	E' → ε

id + id * id

id + id * id

+ id * id

id * id

* id

id

ε

(id+id) * (id+id) * id

(z+q) * w + x * y

ignore

Example of top down Parsing

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

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

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

Parse table for expression & grammar

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

Example of FOLLOW() Calculation

ignore

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

N	FOLLOW(N)
E	{), $}
E'	{), $}
T	{+, −,), $}
T'	{+, −,), $}
F	{*, /, +, −,), $}

① Add $ to FOLLOW(E)

② $E \rightarrow TE'$
⇒ Add FIRST(E') to FOLLOW(T)

③ $E' \rightarrow +TE'$ (similarly $E' \rightarrow -TE'$)
⇒ Add FIRST(E') to FOLLOW(T)
⇒ $E' \rightarrow \varepsilon$ so FOLLOW(E') is added to FOLLOW(T)

④ $T \rightarrow FT'$
⇒ Add FIRST(T') to FOLLOW(F)
⇒ $T' \rightarrow \varepsilon$ so FOLLOW(T') is added to FOLLOW(F)

⑤ $F \rightarrow (E)$
⇒ Add FIRST(')') to FOLLOW(E)

Parse Trace of (z+9)*m + w*y

Stack	Input	Output
$E	(id+id)*id+id*id$	
$E'T	(id+id)*id+id*id$	E → TE'
$E'T'F	(id+id)*id+id*id$	T → FT'
$E'T'E(	(id+id)*id+id*id$	F → (E)
$E'T'E	id+id)*id+id*id$	
$E'T'E'T	id+id)*id+id*id$	E → TE'
$E'T'E'F	id+id)*id+id*id$	T → F
$E'T'E'id	id+id)*id+id*id$	F → id
$E'T'E'	+id)*id+id*id$	
$E'T'	+id)*id+id*id$	E' → +TE'

LR Parser

(Left to Right, Right most derivation)

Grammar

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

Parsing table (LR0 or Simple SLR parsing table)

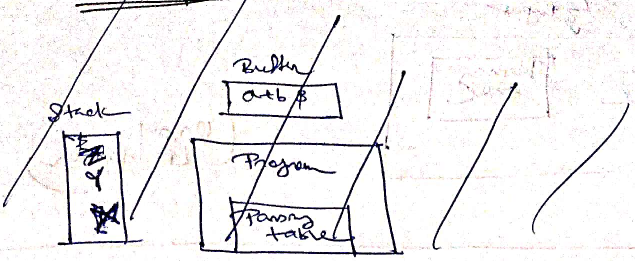
State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Action for parser

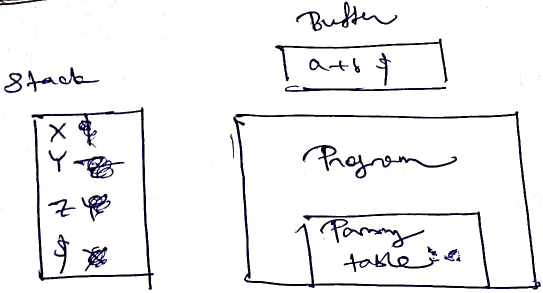
(i) Shift (S)
(ii) Reduce (R)
(iii) Accept (acc)
(iv) Error

Stack	Input	Action
0	id*id+id$	S5
0id5	*id+id$	R6
0E3	*id+id$	R4
0T2	*id+id$	S7
0T2*7	id+id$	S5
0T2*7id5	+id$	R6
0T2*7F10	+id$	R3
0E1	+id$	R2
0E1+6	id$	S5
0E1+6id5	$	R6
0E1+6E3	$	R4
0E1+6T9	$	R1
0E1	$	acc

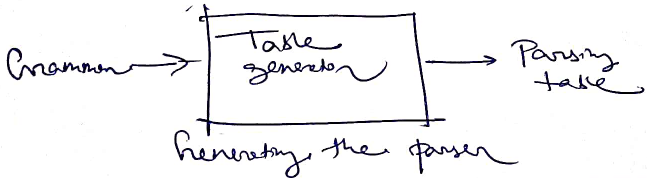
Predictive Parser



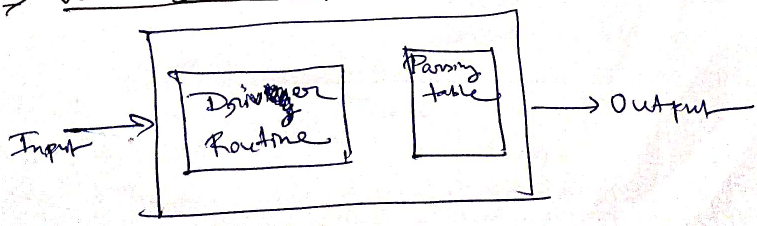
⇒ Predictive Parser

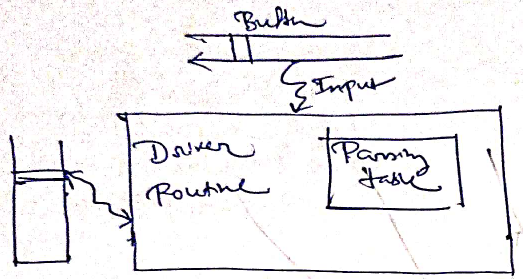


⇒ LR Parser



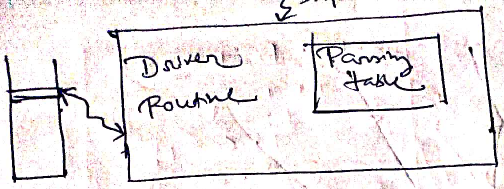
⇒ Generating LR parser





LR parser

(Using LR parser)



(Original LR parser)

⇒ Automatic efficient parser is LR parser.

⊛ Automatic construction parser:—

LR parser method is more general than operator precedence or any other common shift reduce parsing techniques and it can be implemented with some degree of efficiency than other method. LR parser dominates common form of top-down parsing without backtracking. LR parser can detect syntactic error as soon as possible so as to do left to right of the input.

Closure (I):

If I is the set of items in G then

1. Every item in I is in closure (I)
2. If $A \to \alpha \cdot B\beta$ is in closure (I) then exist a Product $B \to \gamma$ then $B \to \cdot\gamma$ is in closure (I)

Goto (I, X):

If $A \to \alpha \cdot X\beta$ is in I then closure of all items $[A \to \alpha X \cdot \beta]$ is Goto (I, X)

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

Augmented Grammar (I.) item

0. E' → E
1. E → E+T
2. E → T
3. T → T*F
4. T → F
5. F → (E)
6. F → id

Closure of I is I_0 = closure (I)

Canonical LR(0) collection

Closure(I) = I_0

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

(.E আগে থাকে E এর prodⁿ e dot থাকবে)

I_1 = Goto (I_0, E)   (new)   (col)

E' → E.
E → E.+T

I_2 = Goto (I_0, T)

E → T.
T → T.*F

I_3 = Goto (I_0, F)

T → F.

(Candidate state)

(I_0 থেকে E করলে হবে I_1 আসবে)

I_4 = Goto (I_0, ( )

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

I_5 = Goto (I_0, id)

F → id.

$I_8 = Goto(I_4, E)$

$F \to (E \cdot)$

$E \to E \cdot + T$

$I_5 = Goto(I_4, T)$

$E \to T \cdot$

$T \to T \cdot * F$

$I_{10} = Goto(I_4, F)$

$T \to F \cdot$

$I_{11} = Goto(I_4, ()$

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_6 = \cdot(I_7, +)$

$E \to E + \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_7 = Goto(I_2, *)$

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$I_8 = Goto(I_4, E)$

$F \to (E \cdot)$

$E \to E \cdot + T$

$I_4, T) = I_2$ (now, goto)

$(I_4, F) = I_3$

$(I_4, () = I_4$

$(I_4, id) = I_5$

$I_9 = (I_6, T)$

$E \to E + T \cdot$

$T \to T \cdot * F$

$(I_6, () = I_4$

$(I_6, id) = I_5$

$(I_7, () = I_4$

$(I_7, id) = I_5$

$I_0 = (E' \to \cdot E)$

$T \to T * F$

$(I_7, () = I_9$

$(I_7, id) = I_5$

$I_1 = (E \to E \cdot)$

$F \to (E)$ → Candidate rules

$(I_8, +) = I_6$

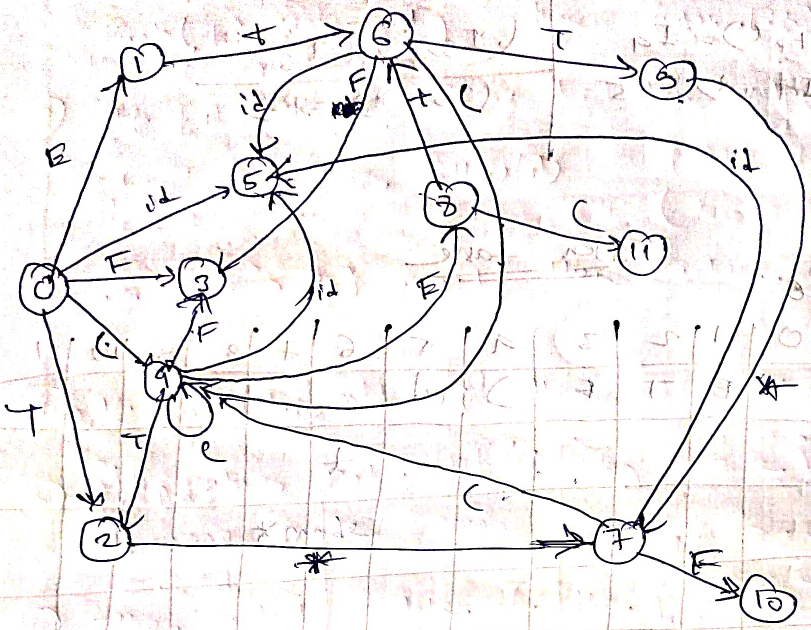
$(I_5, *) = I_7$

Action table

State	0	1	2	3	4	5	6	7	8	9	10	11
0		E	T	F	(	id						
1							+					
2								*				
3												
4			T	F	(	id			E			
5												
6				F	(	id				T		
7					(	id					F	
8							+					)
9								*				
10												
11												

अंदर row
बाहर column

State transition diagram



The D.F.A

Candidate rules for reduction

State	1	2	3	5	8	10	11
Production rule	0	2	4	6	1	3	5

FIRST

0. $E' \rightarrow E$
1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

FIRST (id) = {id}
FIRST (() = {(}
FIRST ()) = {)}
FIRST (*) = {*}
FIRST (+) = {+}

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

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

Construction of SLR parser

1. If $[A \to \alpha . a\beta]$ is in I_i and $GOTO(I_i, a) = I_j$ then action$[i, a] :=$ shift j
2. If $[A \to \alpha .]$ is in I_i, then action$[i, a] :=$ reduce $A \to \alpha$ for all a in FOLLOW(A)
3. If $[S' \to S.]$ in I_i then action$[i, \$] :=$ accept
4. If GOTO$(I_i, A) = I_j$ then GOTO$[i, A] := j$
5. All entries not defined by rule 1 through 4 are made "error"
6. The initial state of the parser is the one constructed from the set of items containing $S' \to S$

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

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

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

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

Candidate rules for Reduction

State	1	2	3	5	8	10	11
Prodn rule	0	2	4	6	1	3	5

I no. → state no.

Prodn rule → Reduced Production

R → in follow of prodn no.

Item table

	0	1	2	3	4	5	6	7	8	9	10	11
0		E	T	F	(	id						
1							+					
2								*				
3												
4			T	F	(	id			E			
5												
6				F	(	id				T		
7					(	id					F	
8							+					)
9								*				
10												
11												

SLR Parsing table

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		R_2	S_7		R_2	R_2			
3		R_4	R_4		R_4	R_4			
4	S_5			S_4			8	2	3
5		R_6	R_6		R_6	R_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		R_1	S_7		R_1	R_1			
10		R_3	R_3		R_3	R_3			
11		R_5	R_5		R_5	R_5			

1. (E) → Follow(E)

Grammar

0. $S' \to S$
1. $S \to aAd$
2. $S \to bAe$
3. $S \to bBd$
4. $S \to aBe$
5. $A \to c$
6. $B \to c$

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

FOLLOW(S) = {$}
FOLLOW(A) = {d, e}
FOLLOW(B) = {d, e}

I0

$S' \to .S$
$S \to .aAd$
$S \to .bAe$
$S \to .bBd$
$S \to .aBe$

I1 = (0, S)

$S' \to S.$

I2 = (0, a)

$S \to a.Ad$
$S \to a.Be$
$A \to .c$
$B \to .c$

I3 = (0, b)

$S \to b.Ae$
$S \to b.Bd$
$A \to .c$
$B \to .c$

I4 = (2, A)

$S \to aA.d$

I5 = (2, B)

$S \to aB.e$

I6 = (2, c)

$A \to c.$
$B \to c.$

I7 = (3, A)

$S \to bA.e$

I8 = (3, B)

$S \to bB.d$

(3, c) = I6

I9 = (4, d)

$S \to aAd.$

I10 = (5, e)

$S \to aBe.$

$I_{11} = \{S \to bA.\}$

$S \to bA.$

$I_{12} = \{S \to bBd.\}$

$S \to bBd.$

State transition matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		S	a	b									
1													
2					A	B	C						
3							C	A	B				
4										d			
5											e		
6													
7												e	
8													d
9													
10													
11													
12													

Candidate rules for reduction

State	1	6	9	10	5	11	11
Production rule	0	5	6	1	4	1	2	3

	a	b	c	d	e	$	S	A	B
0	s2	s3					1		
1						acc			
2			s6					4	5
3			s6					7	8
4				s3					
5					s10				
6				s5 r6	r6 s6				
7					s11				
8				s12					
9									
10									
11									
12									

Follow(S) = {$}

Follow(A) = {d, e}

Follow(B) = {d, e}

0	acd $	shift s2
0a2	cA $	shift s6
0	acef	shift s2
0a2	ceB	shift s6
0a2c6	e$	

(fails) ambiguity → LR(1) needed

ambiguity (fails)

$A \rightarrow c.$

$B \rightarrow c.$

$A \rightarrow c.,\ d$

$B \rightarrow c.,\ e$

$B \rightarrow c.,\ d$

$B \rightarrow c.,\ e$

LR(1) Item Construction

Closure (I)

If $[A \to \alpha \cdot B\beta, a]$ is an item in I and there exists a production $B \to \gamma$ such that $FIRST(\beta a)$ includes a terminal b then the item $[B \to \cdot\gamma, b]$ must be included in closure (I)

The closure of the set of all items of the form $[A \to \alpha X \cdot \beta, a]$ such that $[A \to \alpha \cdot X\beta, a]$ is in I is defined as Goto (X)

Grammar

$S' \to S$

$S \to A$

$S \to aAb$

$A \to aAb$

$A \to B$

$B \to \varepsilon$

Item

$[A \to a \cdot Ab, \$]$ is in I

Production

$A \to B$

Item

$[A \to \cdot B, b]$

should be included in closure (I)

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

FIRST(c) = {c}
FIRST(d) = {d}
FIRST(S') = {c, d}
FIRST(S) = {c, d}
FIRST(C) = {c, d}
FOLLOW(S) = {$}
FOLLOW(C) = {$, c, d}

I_0
$S' \rightarrow .S$
$S \rightarrow .CC$
$C \rightarrow .cC$
$C \rightarrow .d$

I_1 = GOTO(I_0, S)
$S' \rightarrow S.$

I_2 = GOTO(I_0, C)
$S \rightarrow C.C$
$C \rightarrow .cC$
$C \rightarrow .d$

I_3 = GOTO(I_0, c)
$C \rightarrow c.C$
$C \rightarrow .cC$
$C \rightarrow .d$

I_4 = GOTO(I_0, d)
$C \rightarrow d.$

I_5 = GOTO(I_2, C)
$S \rightarrow CC.$

GOTO(I_2, c)
$C \rightarrow c.C$
$C \rightarrow .cC$
$C \rightarrow .d$
} I_3

GOTO(I_2, d)
$C \rightarrow d.$
} I_4

I_6 = GOTO(I_3, C)
$C \rightarrow cC.$

GOTO(I_3, d)
$C \rightarrow d.$
} I_4

I value	State	1	4	5	6
Reduced rule no.	Prod. rule	0	3	1	2

(row, col)

Item table

	0	1	2	3	4	5	6
0		S	C	C	d		
1							
2				c	d	C	
3					d		C
4							
5							
6							

SLR parsing table

State	Action c	Action d	Action $	Goto S	Goto C
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3		S_4			6
4	R_3	R_3	R_3		
5			R_1		
6	R_2	R_2	R_2		

LR(1) / CLR

Augmented Grammar

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

Look Ahead ⇒
$A \rightarrow \cdot B\beta, \$$
$B \rightarrow \cdot b$, FIRST($\beta\$$)

I_0 = closure(I_1)

$S' \rightarrow \cdot S, \$$ ← look ahead
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

First(C)

$I_1 = (0, S)$

$S' \rightarrow S \cdot, \$$

$I_2 = (0, C)$

$S \rightarrow C \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

$I_3 = (0, c)$

$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$I_4 = (0, d)$

$C \rightarrow d \cdot, c/d$

$I_5 = (2, C)$

$S \rightarrow CC \cdot, \$$

$I_6 = (2, c)$

$C \rightarrow c \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

$I_7 = (2, d)$

$C \rightarrow d \cdot, \$$

$I_8 = (3, C)$

$C \rightarrow cC \cdot, c/d$

$(3, c) = I_3$

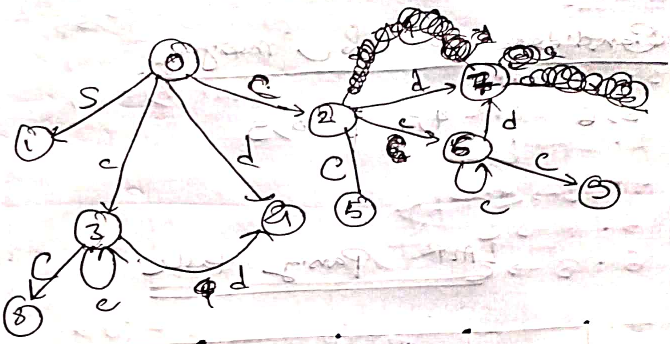
$(3, d) = I_4$

$I_9 = (6, C)$

$C \rightarrow cC \cdot, \$$

$(6, c) = I_6$

$(6, d) = I_7$



CLR parsing table

	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

State	4	5	7	8	9
Rule No.	3	1	3	2	2
Input	c/d	$	$	c/d	$

(R) — row

col. — look aheads

FIRST(C) = {c, d}

FIRST(d) = {d}

FIRST(S) = {c, d}

FIRST(S) = {c, d}

FIRST(C) = {c, d}

FOLLOW(S) = {$}

FOLLOW(C) = {$, c, d}

Candidate sets for merging

3,6 | 4,7 | 8,9

LALR Parsing table

	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Comparison of SLR and LALR Parsing table

SLR Parsing Table

	c	d	$	S	C
0	s_3	s_4		1	2
1			acc		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

LALR Parsing table

	c	d	$	S	C
0	s_{36}	s_{47}		1	2
1			acc		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Since the grammar is also an SLR grammar so the parsing tables are identical.

Grammer

0. $S' \to S$
1. $S \to aAd$
2. $S \to bAe$
3. $S \to bBd$
4. $S \to aBe$
5. $A \to c$
6. $B \to c$

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

I_0
$S' \to .S, \$$
$S \to .aAd, \$$
$S \to .bAe, \$$
$S \to .bBd, \$$
$S \to .aBe, \$$

$I_1 = (0, S)$
$S' \to S., \$$ → 0

$I_2 = (0, a)$
$S \to a.Ad, \$$
$A \to .c, d$
$S \to a.Be, \$$
$B \to .c, e$

$I_3 = (0, b)$
$S \to b.Ae, \$$
$A \to .c, e$
$S \to b.Bd, \$$
$B \to .c, d$

$I_4 = (2, A)$
$S \to aA.d, \$$

$I_5 = (2, c)$
$A \to c., d$ → 5
$B \to c., e$ → 6

$I_6 = (3, B)$
$S \to bB.d, \$$

$I_7 = (3, A)$
$S \to bA.e, \$$

$I_8 = (a, B)$
$S \to aB.e, \$$

$I_9 = (3, c)$
$A \to c., e$ → 5
$B \to c., d$ → 6

$I_{10} = (4, d)$
$S \to aAd., \$$ → 1

$I_{11} = (5, e)$
$S \to aBe., \$$ → 4

$I_{12} = (7, e)$
$S \to bAe., \$$ → 2

$I_{13} = (3, d)$
$S \to bBd., \$$ → 3

LR(1)

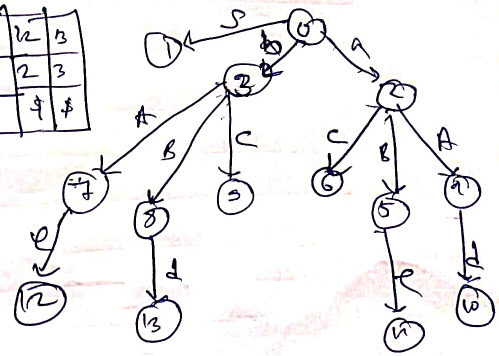
	a	b
0	S_2	S_3
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		

State
Rule No
Input

LR(1) Parsing table

	a	b	c	d	e	$	B	A	B
0	S_2	S_3					1		
1						acc			
2			S_6					4	5
3			S_7					7	8
4				S_{10}					
5					S_{11}				
6				S_5	S_6				
7					S_{12}				
8				S_{13}					
9				r_6	r_6				
10						r_1			
11						r_4			
12						r_2			
13						r_3			

State	6	6	9	9	10	11	12	13
Rule No	5	6	5	6	1	4	2	3
Input	d	e	e	d	$	$	$	$



There exist no pair of sets that have the same configuration.

Hence, the grammar is not an LALR grammar.