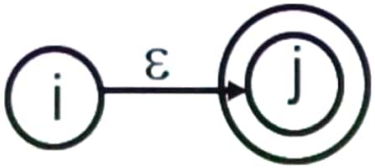
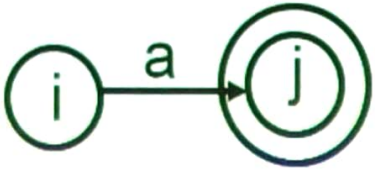
Construction of NFA from Regular Expression

Let the regular expression R over an alphabet Σ be given. We want to construct an NFA. The following rules are available.

1. $R = \varepsilon$



2. $R = a$



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;
```

Algorithm to find the DFA from an NFA

while there is an unmarked state $x=(s_1, s_2, \ldots s_n)$ do

begin

    mark x;

    for each input a do

    begin

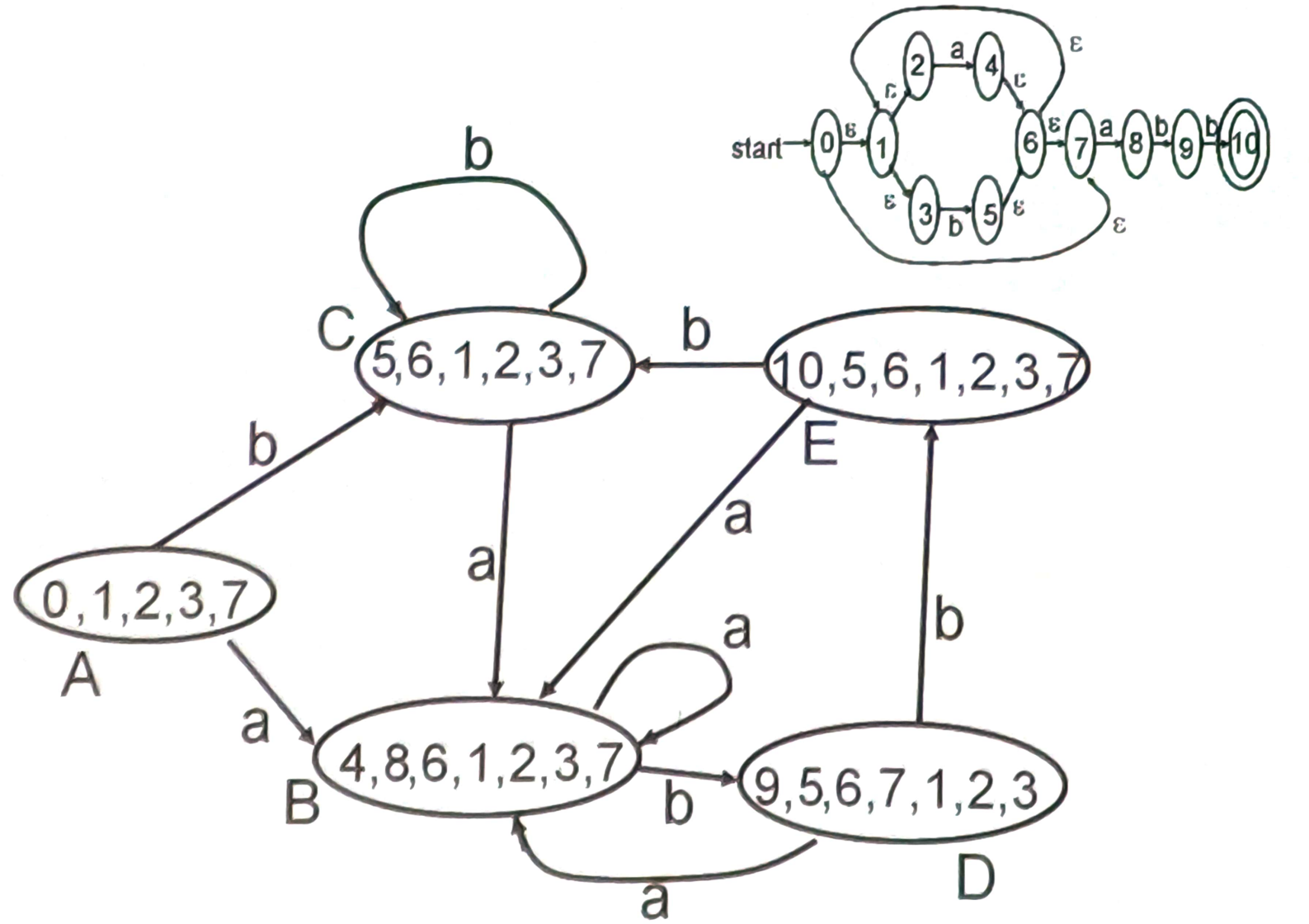
        let T be the set of states to which there is a transition on a from s_i in x;

        y:= ε-closure(T)

        if y has not yet been added to the set of states of D then mark y an unmarked state of D; add a transition from x to y labeled 'a' if not already present;

    end;

end;

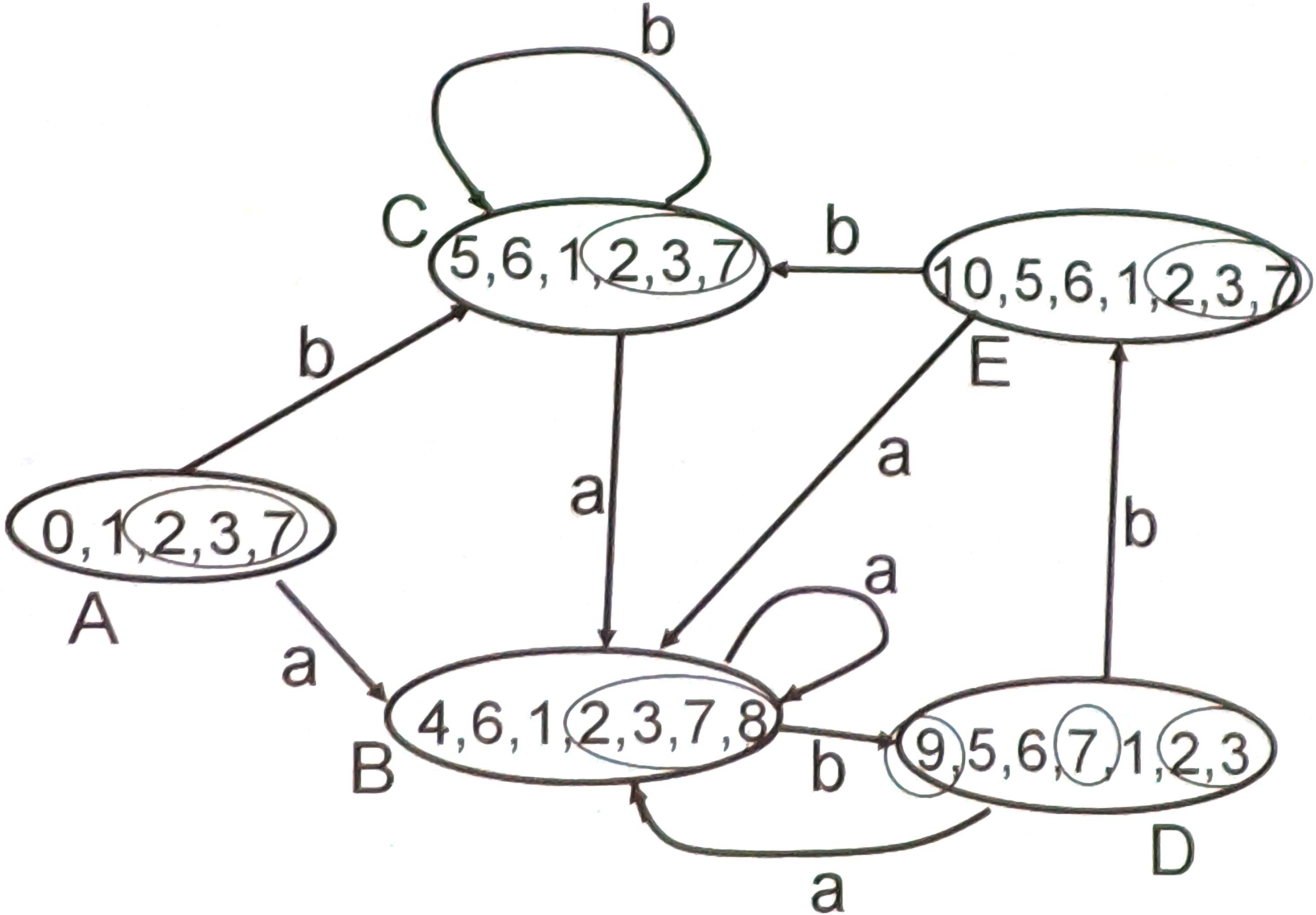

start
0
1
2
3
4
5
6
7
8
9
10
a
b
ε
A
0,1,2,3,7
B
4,8,6,1,2,3,7
C
5,6,1,2,3,7
D
9,5,6,7,1,2,3
E
10,5,6,1,2,3,7

Minimization of a DFA

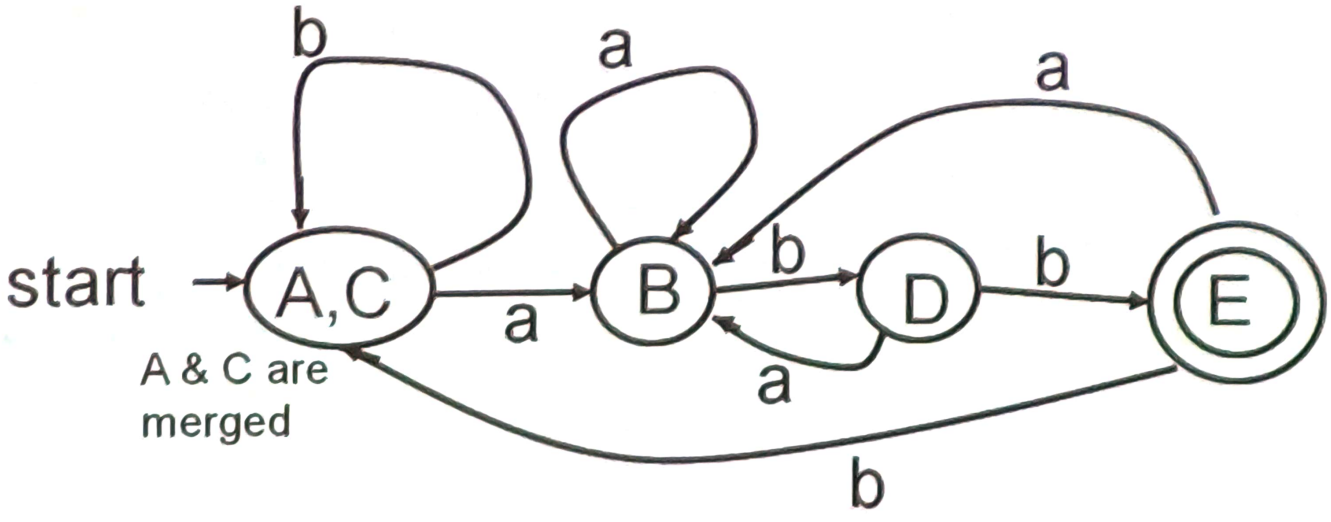
- Important states of an NFA is a state for which a non ε transition exists. Therefore two states of a DFA can be identified as a single state if

1. They have same important states and
2. they either both include or both exclude the accepting states of the corresponding NFA.

Important States : 2,3,7,8,9

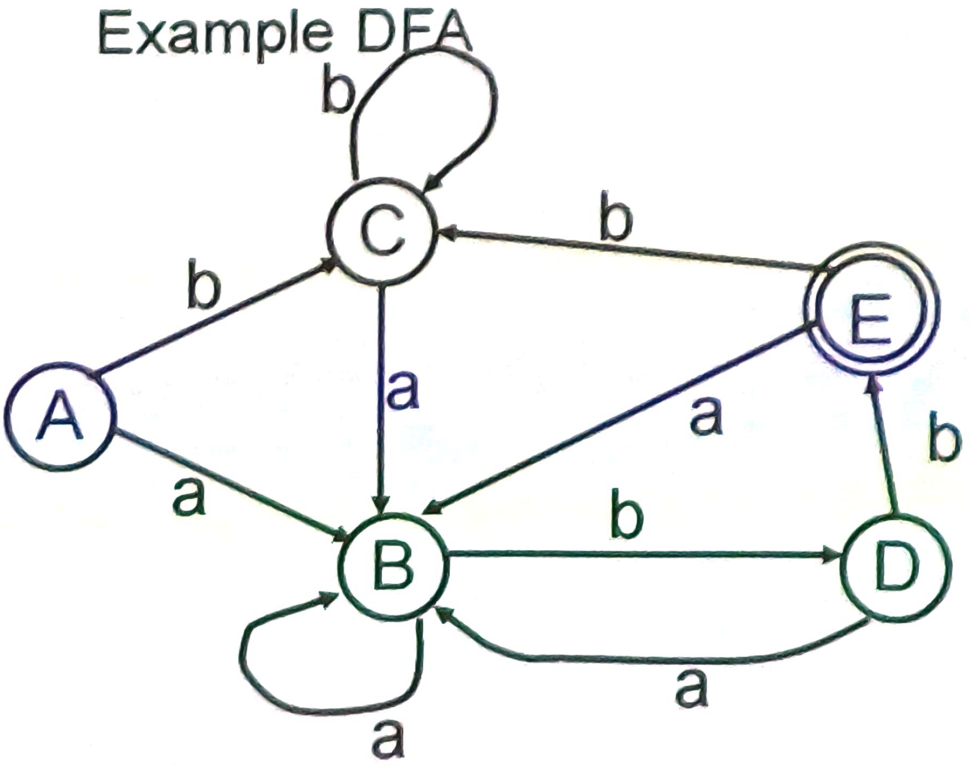


Minimized DFA



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

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

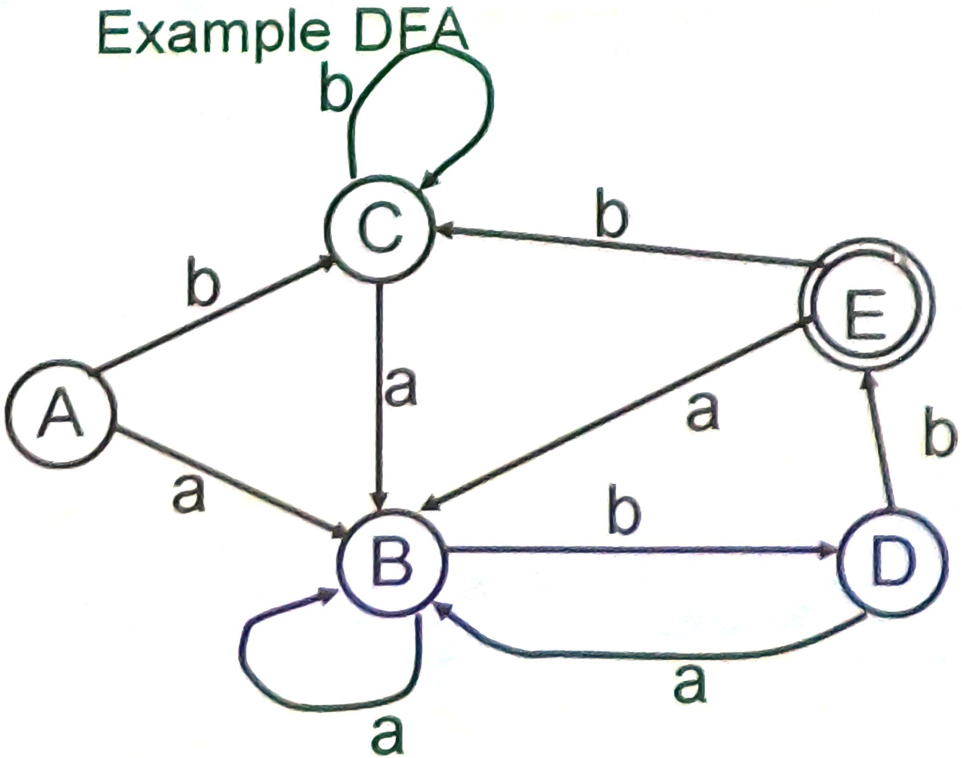


Π_1= (ABCD) (E)

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 the 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;

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



Π_1= (ABCD) (E)

Observations

Π_1= (ABCD) (E)

Transition on 'a' : A→B, B→B, C →B, D →B

Transition on 'b' : A →C, B →D, C →C, D →E

Π_2 =(ABC) (D) (E)

Transition on 'a' : A→B, B →B, C →B

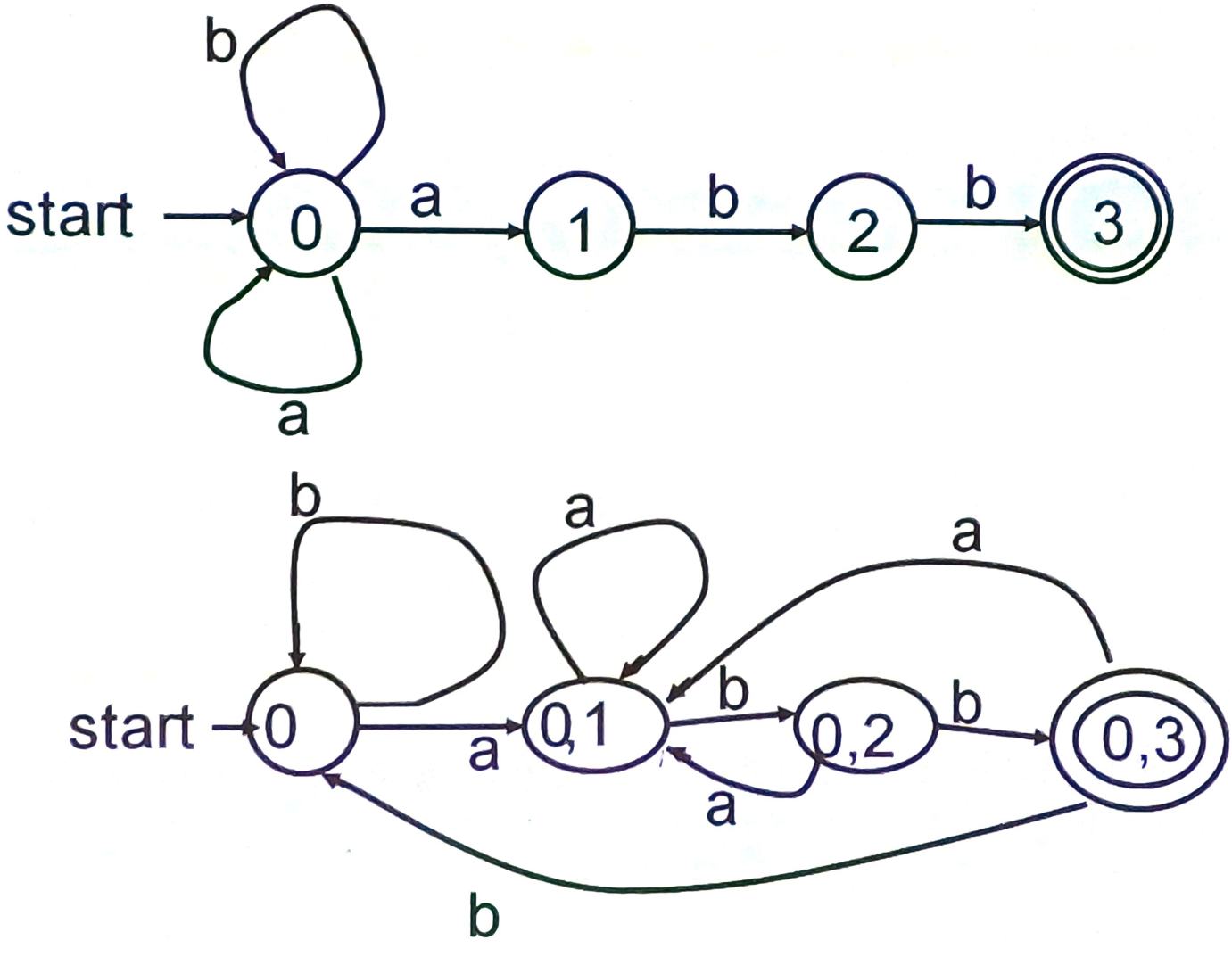
Transition on 'b' : A →C, B →D, C →C

Π_3 =(AC) (B) (D) (E)

Transition on 'a' : A→B, C →B

Transition on 'b' : A →C, C →C

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


b
start
0
a
1
b
2
b
3
a
b
a
a
start
0
a
0,1
b
0,2
b
0,3
a
b

Grammar

- A grammar gives a precise, yet easy to understand, syntactic specification for the programs of a particular language.
- An efficient parser can be constructed automatically from a properly designed grammar. Certain parser construction process can reveal syntactic ambiguities and other difficult to parse constructs which might otherwise go undetected in the initial design phase of a language and its compiler
- A grammar imparts a structure to a program that is useful for its translation into object codes and for detection of errors

Context Free Grammars

- 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 (1)
- If $S_1, S_2, \ldots, S_n$ are statements then
  "begin $S_1, S_2, \ldots, S_n$ end" is a statement (2)
- If E_1 and E_2 are expressions then $E_1 + E_2$ is an expression (3)
- (1) can be expressed by the rewriting rule
  Statement → **if** <expression> **then** <statement> **else** <statement> (4)
- Similarly for (3) we have
  <expression> → <expression> + <expression> (5)

Context Free Grammars

- For (2) we may write

  <statement> → **begin** <statement> ;<statement>;...;
  <statement> **end**
- We require the rewriting 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> (6)

Context Free Grammars

- 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 such as (4), (5) or (6) is an example of a grammar.
- In general a grammar involves four quantities ; ***terminals, nonterminals, a start symbol*** and ***productions.***

Context Free Grammars

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

  example : **begin, end, else, then**
- ***nonterminals*** are special symbols that denote sets of strings.

  example : <statement> , <statement-list> , <expression>
- One ***nonterminal*** is selected as the ***start symbol***, and it denotes the language in which we are truly interested.
- The ***productions (rewriting rules)*** define the way in which the syntactic categories may be built up from one another and from the ***terminals***

Context Free Grammars

- Each production consists of a ***nonterminal,*** followed by an arrow or "::=" followed by a string of ***nonterminals*** and ***terminals***

The rules in (6) represent three productions viz.,

<statement> → **begin** <statement-list> **end**
<statement-list> → <statement>
<statement-list> → <statement> ; <statement-list>

Context Free Grammar Example

nonterminals : <expression>, <operator>

start symbol : <expression>

Terminals : **id**, +, -, *, /, ↑, (,)

Example

The productions are

<expression> → <expression> <operator> <expression>

<expression> → (<expression>)

<expression> → - <expression>

<expression> → **id**

Example

The productions are

<expression> → <expression> <operator> <expression>

<expression> → (<expression>)

<expression> → - <expression>

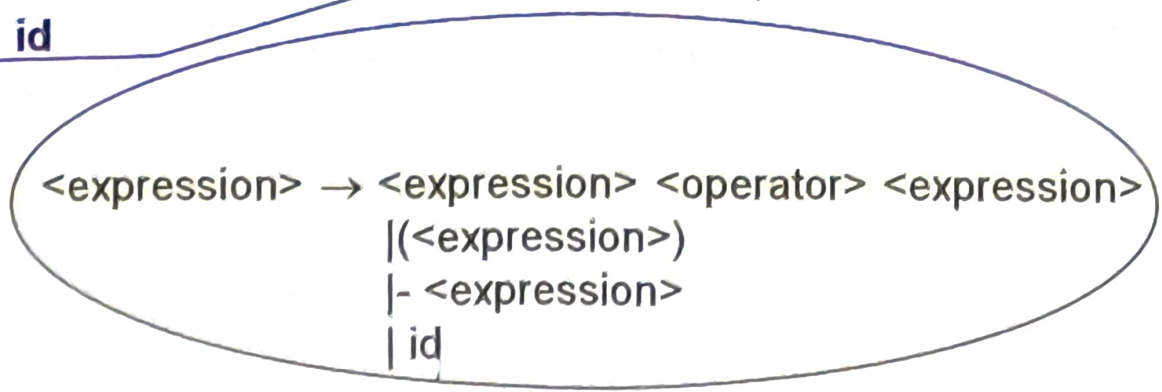
<expression> → **id**

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

Example

The productions are

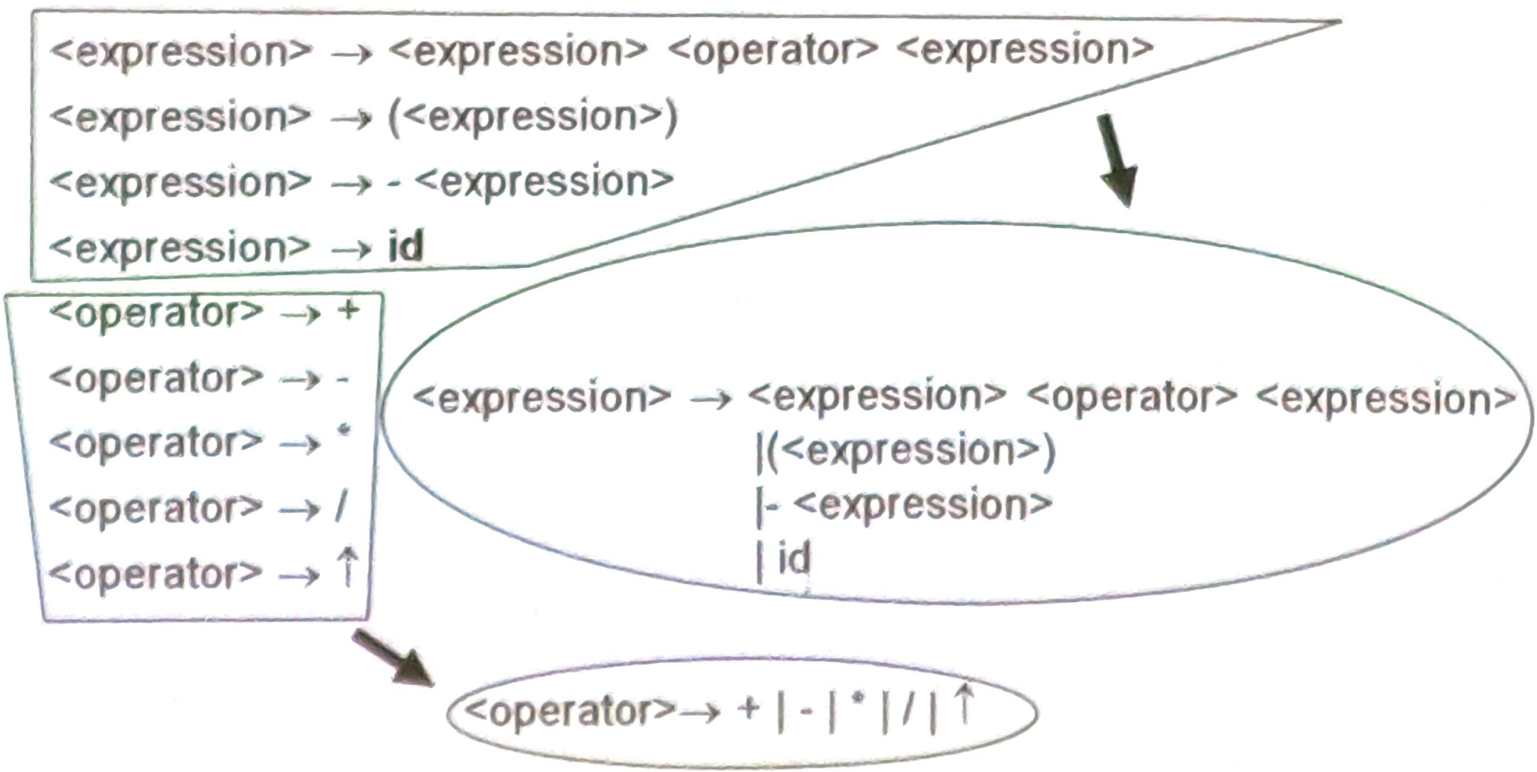
<expression> → <expression> <operator> <expression>
<expression> → (<expression>)
<expression> → - <expression>
<expression> → **id**



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

Example

The productions are



nonterminals

- lower-case 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.

terminals

- single lowercase letters a,b,c,...;
- operator symbols such as +,-,etc.;
- punctuation symbols such as parenthesis, comma, etc.;
- the digits 0, 1, 2,...,9;
- boldface strings such as **id** or **if**

- Capital symbol near the end of the alphabet, chiefly X,Y,Z represent ***grammar symbols***, that is, either ***nonterminals*** or ***terminals.***
- Small letters near the end of the alphabet, chiefly u,v,…,z, represent strings of terminals.
- Lowercase Greek letters, α, β, γ, for example, represent strings of grammar symbols. Thus a generic production could appear as $A \rightarrow \alpha$, indicating that there is a single nonterminal A on the left of the arrow (the ***left side*** of the production) and a string of grammar symbols α to the right of the arrow (***right side*** of the production).

- If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k$ are all productions with A on the left (we call them ***A-production***), we may write $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k$. We call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the ***alternates*** for A
- Unless otherwise stated the ***left side*** of the first production is the ***start symbol.***
- Using these short hands we could write the grammar of the previous example as

  $E \rightarrow E\ A\ E$

  $A \rightarrow + \mid - \mid * \mid / \mid \uparrow$
- E and A are ***nonterminals***, with E the ***start symbol***. The remaining symbols are ***terminals***.

- Consider the grammar

  $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid \mathbf{id}$

  $E \rightarrow -E$ is a production therefore we can replace a single E by –E. We write $E \Rightarrow -E$ that reads "E derives –E"

  We can take a single E and repeatedly apply productions in any order to obtain a sequence of replacements. For example,

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

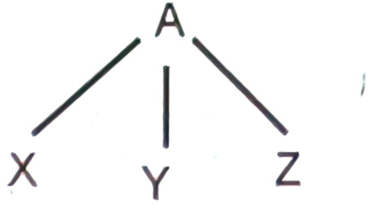
  This is the derivation of -(**id**) from E and provides the proof that one single instance of an expression is -(**id**).

- In a more abstract setting
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production and α and β are arbitrary strings of grammar symbols.
- If $\alpha_1 \Rightarrow \alpha_2, \Rightarrow \ldots \Rightarrow \alpha_n$ We say α_1 derives α_n .
- The symbol $\overset{*}{\Longrightarrow}$ means "derives in zero or more steps"
- The symbol $\overset{+}{\Longrightarrow}$ means "derives in one or more steps"

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production and α and β are arbitrary strings of grammar symbols.
- If $\alpha_1 \Rightarrow \alpha_2, \Rightarrow \ldots \Rightarrow \alpha_n$ We 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 the context free grammar G and the ***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 say a string of terminals *w* is in L(G) if and only if $S \overset{+}{\Rightarrow} w$. The string *w* is called a sentence of G. If $S \overset{*}{\Rightarrow} \alpha$, where α may contain *nonterminals*, then we say *α is a sentential form* of G.

Parse Tree

- A graphical representation of the derivations that filters out the choice regarding replacement order is called the parse tree.
- Each interior node of the parse tree is labeled by some nonterminal A and the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this A was replaced in the derivation. For example if $A \rightarrow XYZ$ is a production used at some step of a derivation then the parse tree for the derivation will have the subtree



- Operator precedence

  - (Unary minus)

  ↑

  * /

  + -

Associativity relation

a ↑b ↑c = a ↑(b ↑c)

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

a+-b↑c+d*e

=(a+((-b)↑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>

<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>

⇒ <element> + <term>

⇒ **id** + <term>

⇒ **id** + <term> * <factor>

⇒ **id** + <factor>* <factor>

⇒ **id** + <primary> * <factor>

⇒ **id** + <element> * <factor>

⇒ **id** + **id** * <factor>

⇒ **id** + **id** * <primary>

⇒ **id** + **id** * <element>

⇒ **id** + **id** * **id**

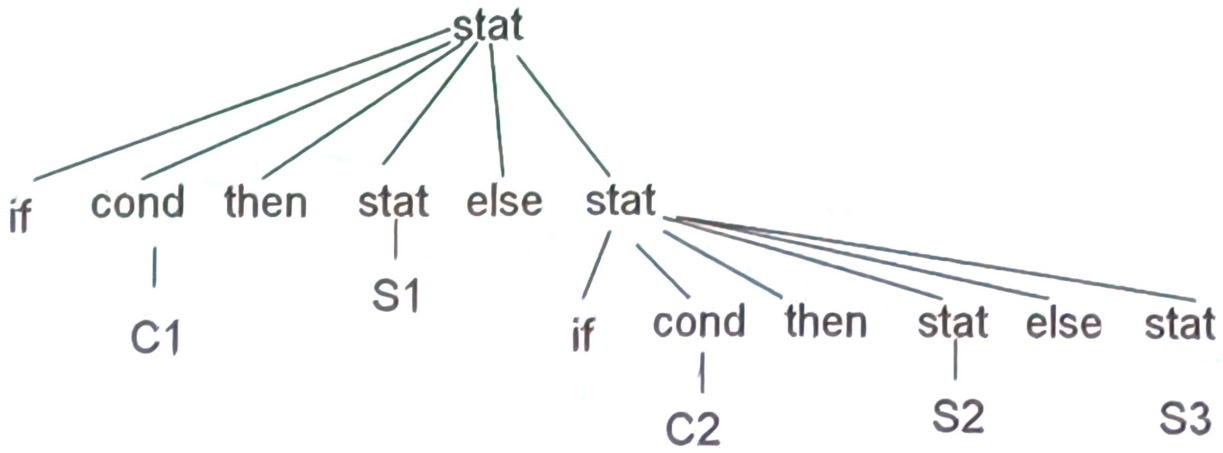
<stat> → If <cond > then <stat>

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

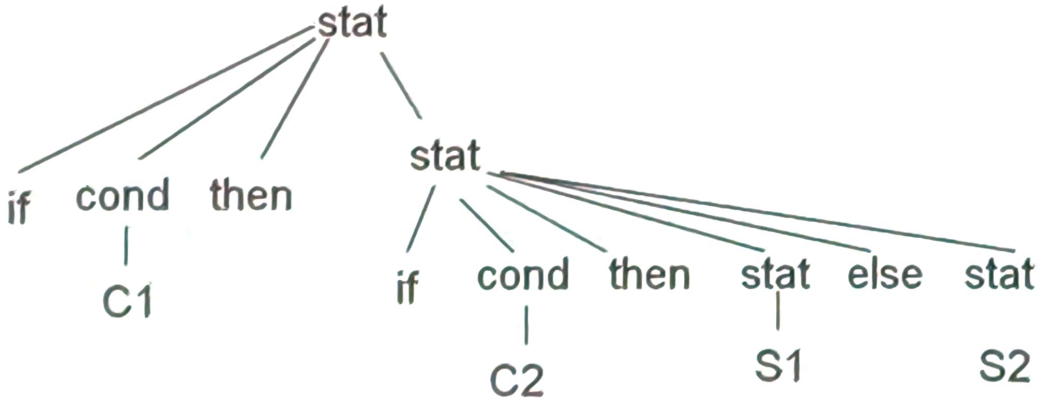
| <other stat>

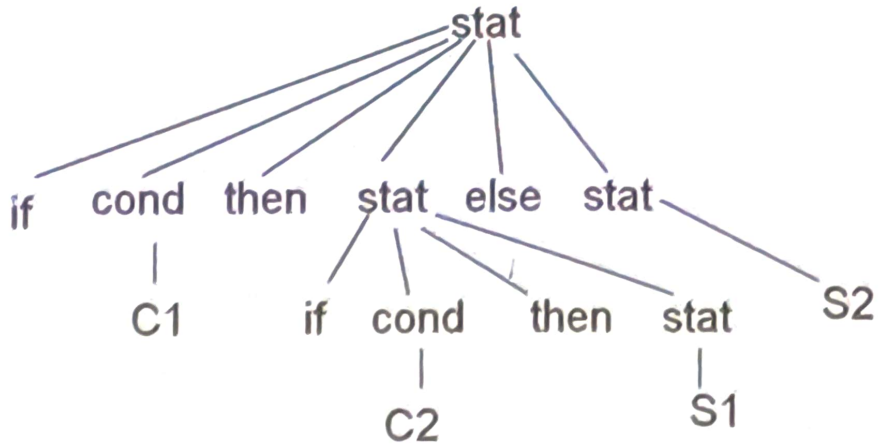
Example : if C1 then S1

else if C2 then S2 else S3



Example : if C1 then if C2 then S1 else S2





Unambiguous grammar

<stat> → <matched – stat>

<unmatched – stat>

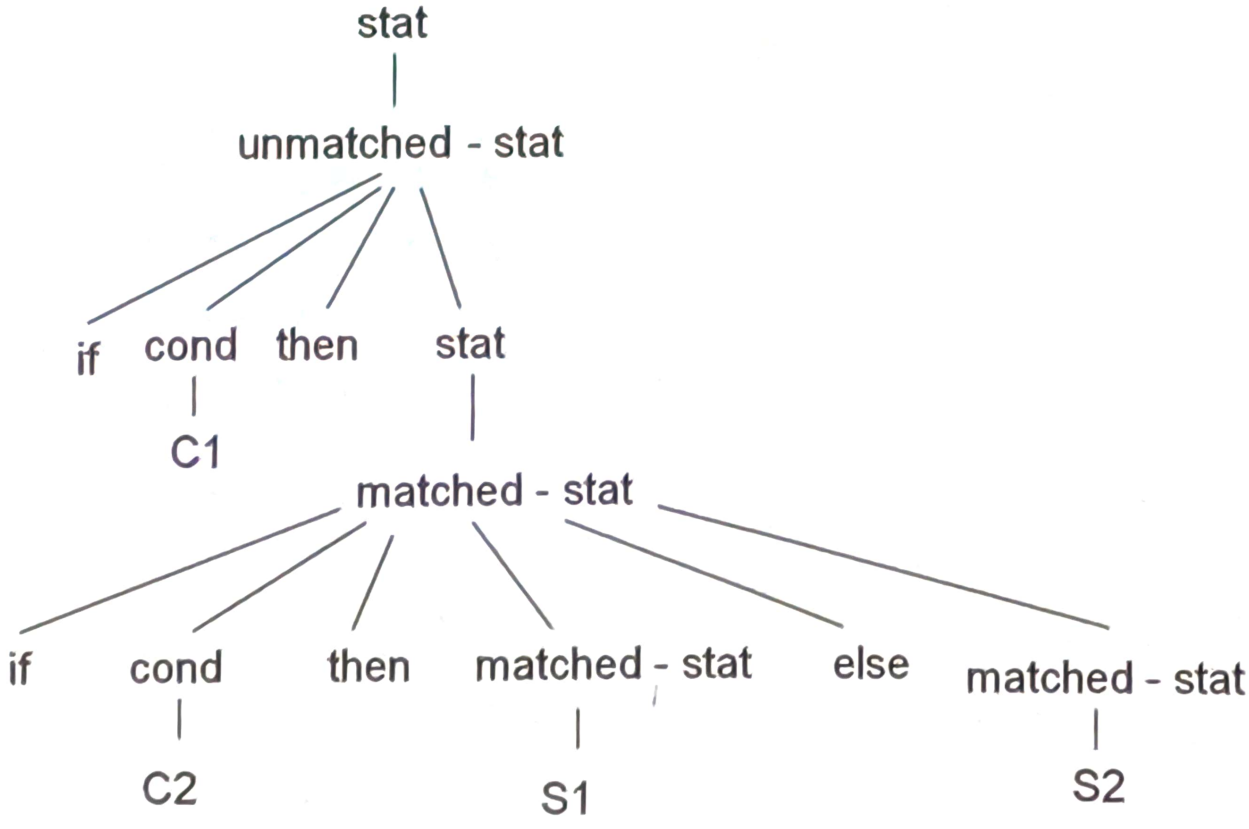
<matched – stat> → if <cond> then <matched - stat> else <matched -stat>

| <other – stat>

<unmatched – stat> → If <cond > then <stat>

| if <cond> then <matched - stat> else <unmatched - stat>

if C1 then if C2 then S1 else S2



Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"

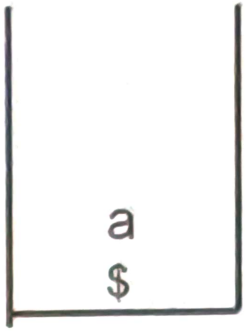
a b b c d e

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

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"



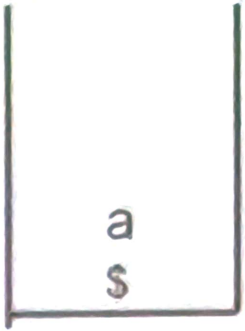
b b c d e

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

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"



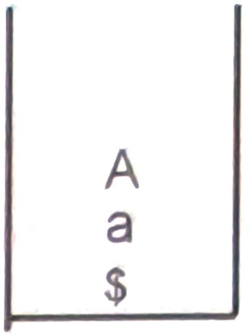
b c d e

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

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"



b c d e

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

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

Consider a string "abbcde"

c d e

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

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

Consider a string "abbcde"

c d e

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

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"

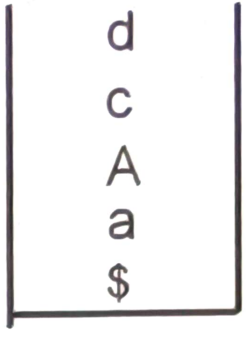
d e

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

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

Consider a string "abbcde"



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

e

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

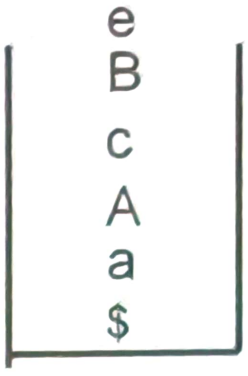
Consider a string "abbcde"

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

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"

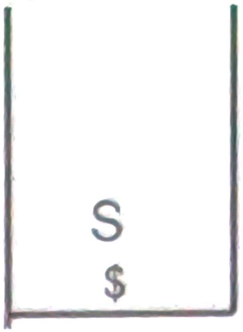


Stack	Input	Action
$	abbcde$	shift
$a	bbcde$	shift
$ab	bcde$	reduce A →b
$aA	bcde$	shift
$aAb	cde$	reduce A →Ab
$aA	cde$	shift
$aAc	de$	shift
$aAcd	e$	reduce B →d
$aAcB	e$	shift
$aAcBe	$	reduce S → aAcBe
$ S	$	accept

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"



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

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

Consider a string
"abbcde"

Right Most Derivation

$S \Rightarrow aAcBe$
$\Rightarrow aAcde$
$\Rightarrow aAbcde$
$\Rightarrow abbcde$

Left Most Derivation

$S \Rightarrow aAcBe$
$\Rightarrow aAbcBe$
$\Rightarrow abbcBe$
$\Rightarrow abbcde$

If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A \omega \underset{rm}{\Rightarrow} \alpha\beta\omega$ then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta\omega$

If the grammar is unambiguous then every right sentential form of the grammar has exactly one handle.

Example

Grammar

(1)E → E+E

(2)E → E*E

(3)E → (E)

(4)E → **id**

Consider the right most derivation

$E \Rightarrow \underline{E + E}$

$\Rightarrow E + \underline{E * E}$

$\Rightarrow E + E * \underline{id}$

$\Rightarrow E + \underline{id} * id$

$\Rightarrow \underline{id} + id * id$

$E \Rightarrow \underline{E * E}$

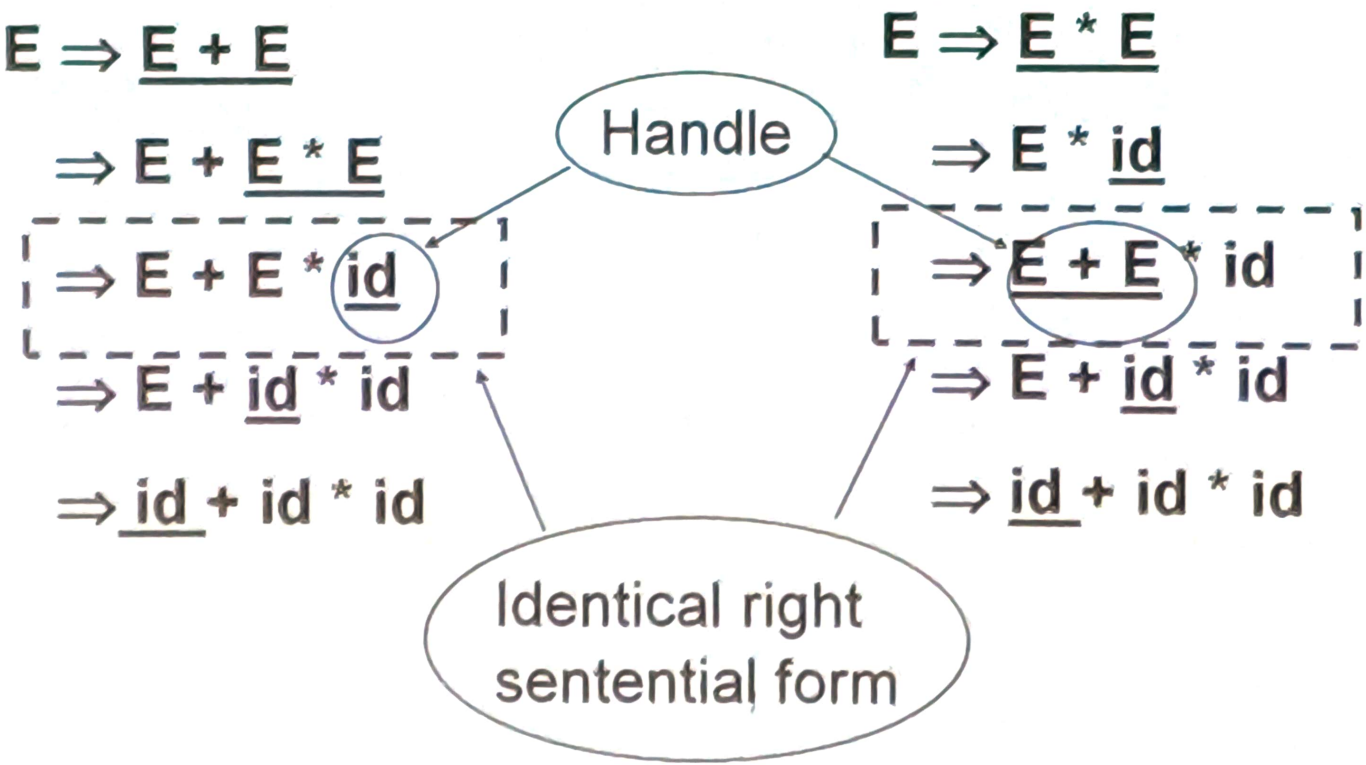
$\Rightarrow E * \underline{id}$

$\Rightarrow \underline{E + E} * id$

$\Rightarrow E + \underline{id} * id$

$\Rightarrow \underline{id} + id * id$

Consider the right most derivation



Right sentential form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

We will solve this ambiguity problem by using a grammar called operator grammar

Operator Grammar :

Definition : A grammar which has no production whose right side is ε or has two adjacent nonterminals

Original Grammar :

$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$

$\mathbf{A} \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Operator Grammar :

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid \mathbf{id}$

Representation of Precedence Relation

- We introduce the following three relational operators to represent the precedence relations
- $A \dot{>} B$ means A has higher precedence than B
- $A \dot{<} B$ means A has lower precedence than B
- $A \doteq B$ means A and B have same precedence.

Rules for Finding Operator Precedence Relation

1. If operator θ_1 has higher precedence than θ_2 then

   $\theta_1 \dot{>} \theta_2$ and $\theta_2 \dot{<} \theta_1$

2. If θ_1 and θ_2 have same precedence then if the operators are left associative then

   $\theta_1 \dot{>} \theta_2$ and $\theta_2 \dot{>} \theta_1$ and if they are right associative then $\theta_1 \dot{<} \theta_2$ and $\theta_2 \dot{<} \theta_1$

3. $\theta \lessdot \cdot \text{id} \;,\; \text{id} \gtrdot \theta \;,\; \theta \lessdot \cdot (\;,\; (\lessdot \cdot \theta \;,$
$) \gtrdot \theta,\; \theta \gtrdot),\; \theta \gtrdot \$$ and $\$ \lessdot \cdot \theta$

Also

$(\doteq),\; \$ \lessdot \cdot (,\; \$ \lessdot \cdot \text{id},\; (\lessdot \cdot (,\; \text{id} \gtrdot \$,$

$) \gtrdot \$,\; (\lessdot \cdot \text{id},\; \text{id} \gtrdot),\;) \gtrdot).$

1. If operator θ_1 has higher precedence than θ_2 then $\theta_1 \dot{>} \theta_2$ and $\theta_2 \dot{<} \theta_1$. For example $* \dot{>} +$ and $+ \dot{<} *$ These relations ensure 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		$\dot{>}$	$\dot{>}$	$\dot{>}$
+	$\dot{<}$	$\dot{>}$	$\dot{<}$	$\dot{>}$
*	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
$	$\dot{<}$	$\dot{<}$	$\dot{<}$	

$ $\dot{<}$ id $\dot{>}$ + $\dot{<}$ id $\dot{>}$ * $\dot{<}$ id $\dot{>}$ $

$ $\dot{<}$ E + $\dot{<}$ E * E $\dot{>}$ $

Parser Actions

- Shift
- Reduce
- Accept
- Report Error

1. In a shift action the next input symbol is shifted to the top of the stack.
2. 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.
3. In an accept action the parser announces successful completion of parsing.
4. In an error action, the parser discovers that a syntax error has occurred an calls an error recovery routine.

Parser Action Algorithm

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 $\lessdot$ b or a $\doteq$ b then shift b on the stack
    else if a $\gtrdot$ b then
    repeat
        pop the stack until top stack terminal is related by $\lessdot$ to the terminal most recently popped.;
        store each popped item in sequence and reduce else call the error correcting routine;
end.

Stack	Relation	Input	Action
$	$\lessdot$	Id+id*id $	shift
$id	$\gtrdot$	+id*id $	reduce E→id
$E	$\lessdot$	+id*id $	shift
$E+	$\lessdot$	id*id $	shift
$E+id	$\gtrdot$	*id $	reduce E→id
$E+E	$\lessdot$	*id $	shift
$E+E*	$\lessdot$	id $	shift
$E+E*id	$\gtrdot$	$	reduce E→id
$E+E*E	$\gtrdot$	$	reduce E →E*E
$E+E	$\gtrdot$	$	reduce E →E+E
$E		$	accept

Stack	Relation	Input	Action
$	$\lessdot$	Id+id*id $	shift
$id	$\gtrdot$	+id*id $	reduce E→id
$E	$\lessdot$	+id*id $	shift
$E+	$\lessdot$	id*id $	shift
$E+id	$\gtrdot$	*id $	reduce E→id
$E+E	$\lessdot$	*id $	shift
$E+E*	$\lessdot$	id $	shift
$E+E*id	$\gtrdot$	$	reduce E→id
$E+E*E	$\gtrdot$	$	reduce E →E*E
$E+E	$\gtrdot$	$	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 $	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
$	$\lessdot$	Id*id+id $	shift
$id	$\gtrdot$	*id+id $	reduce E→id
$E	$\lessdot$	*id+id $	shift
$E*	$\lessdot$	id+id $	shift
$E*id	$\gtrdot$	+id $	reduce E→id
$E*E	$\gtrdot$	+id $	reduce E →E*E
$E	$\lessdot$	+id $	shift
$E+	$\lessdot$	id $	shift
$E+id	$\gtrdot$	$	reduce E→id
$E+E	$\gtrdot$	$	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 the precedence relations $\lessdot$, $\doteq$ and $\gtrdot$ are disjoint

Rules for finding Operator Precedence Relation

1. If $S \rightarrow \alpha a \beta b \gamma$ where β is either a single nonterminal or ε then $a \doteq b$.
2. If $S \rightarrow \alpha a A \beta$ and $A \stackrel{+}{\Rightarrow} \gamma b \delta$ such that γ is either ε or a single nonterminal then $a \lessdot b$, also $\$ \lessdot b$ if $S \Rightarrow \gamma b \delta$ where γ is either ε or a single nonterminal
3. If $S \rightarrow \alpha A b \beta$ and $A \stackrel{+}{\Rightarrow} \gamma a \delta$ where δ is either ε or a single nonterminal then $a \gtrdot b$, also $a \gtrdot \$$ if $S \stackrel{+}{\Rightarrow} \gamma a \delta$ where δ is either ε or a single nonterminal

Rules for finding Operator Precedence Relation

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

$S \rightarrow iCtS$

$S \rightarrow iCtSeS$

$S \rightarrow a$

$C \rightarrow b$

$S \rightarrow iCtSeS$

$i \doteq t$ and $t \doteq e$

Rules for finding Operator Precedence Relation

2. If $S \rightarrow \alpha aA\beta$ and $A \stackrel{+}{\Rightarrow} \gamma b\delta$ such that γ is either ε or a single nonterminal then $a \lessdot b$, also $\$ \lessdot b$ if $S \stackrel{*}{\Rightarrow} \gamma b\delta$ where γ is either ε or a single nonterminal

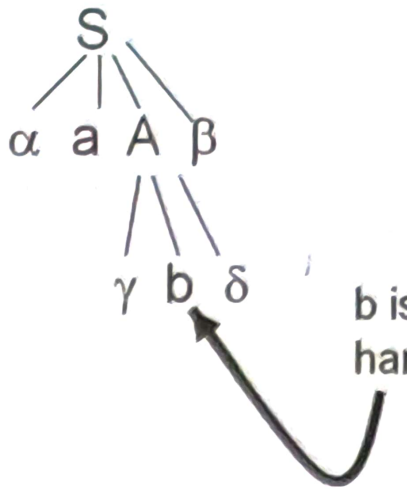
$S \rightarrow iCtS$

$S \rightarrow iCtSeS$

$S \rightarrow a$

$C \rightarrow b$

$S \rightarrow iCtS$ and $C \stackrel{+}{\Rightarrow} b$ so $i \lessdot b$



$\gamma b\delta$ should be reduced first to A and then $\alpha aA\beta$ should be reduced to S. Hence $a \lessdot b$

b is the leftmost terminal of a handle

Rules for finding Operator Precedence Relation

1. If $S \rightarrow \alpha A b \beta$ and $A \stackrel{+}{\Rightarrow} \gamma a \delta$ where δ is either ε or a single nonterminal then $a \doteq b$, also $a \doteq \$$ if $S \stackrel{*}{\Rightarrow} \gamma a \delta$ where δ is either ε or a single nonterminal

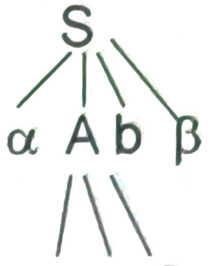
$S \rightarrow iCtS$

$S \rightarrow iCtSeS$

$S \rightarrow a$

$C \rightarrow b$

$S \rightarrow iCtS$ and $C \stackrel{+}{\Rightarrow} b$ so $b \doteq t$



$\gamma b \delta$ should be reduced first to A and then $\alpha a A \beta$ should be reduced to S. Hence $a \doteq b$

a is the rightmost 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 of the form

$A \rightarrow \alpha aA\beta,$

where, $\alpha = E$, $a = +$, $A = E$ and $\beta = \varepsilon$.

Again $E \overset{+}{\Rightarrow} E+E$ which is of the form $\gamma b\delta$, where, $\gamma = E$, $b = +$ and $\delta = E$. Then by rule (2)

$+ \lessdot + $

Similarly

$E \rightarrow E+E$

stands for $A \rightarrow \alpha Ab\beta$

where,

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

and $E \overset{+}{\Rightarrow} E+E$ stands for

$A \overset{+}{\Rightarrow} \gamma a\delta$

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

$\delta = E$, that is by rule (3)

$+ \doteq +$

Hence the above grammar is not an operator precedence grammar

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 (or id as the first symbol and the symbol) or id as the 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 the symbols *,(and id can be first and *,) and id can be last in a derivation from T.
- Similarly the symbols +,*,(or id can be first and +,*,) or id can be last in a derivation of E

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

We define the First terminal for a nonterminal A as LEADING(A)

We define the Last terminal for a nonterminal A as TRAILING(A)

- LEADING(A)
  = {a | A $\overset{+}{\rightarrow}\gamma a\delta$ where γ is ε or single nonterminal}
- Concept for computation of LEADING(A)
  1. **a** is in LEADING(A) if A $\rightarrow\gamma$**a**δ where γ is ε or a nonterminal
  2. **a** is in LEADING(A) if A $\rightarrow$Bα and **a** is in LEADING(B).

- TRAILING(A) =

  $= \{a \mid A \overset{+}{\Rightarrow} \gamma a \delta$ where δ is ε or single nonterminal$\}$
- Concept for computation of TRAILING (A)
  1. **a** is in TRAILING (A) if $A \rightarrow \gamma \mathbf{a} \delta$ where δ is ε or a nonterminal
  2. **a** is in TRAILING(A) if $A \rightarrow \alpha B$ and **a** is in TRAILING(B).

ALGORITHM

```
procedure Install(A,a)
If not L[A,a] then
begin
    L[A,a] := true;
    push(A,a) onto stack;
end;
```

Main Algorithm

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

- Grammar :

  $E \rightarrow E + T$

  $E \rightarrow T$

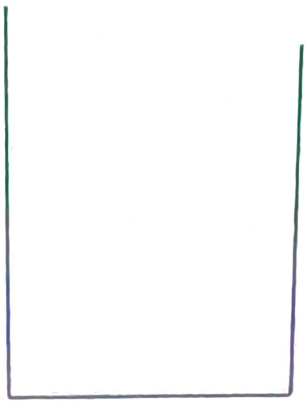
  $T \rightarrow T^*F$

  $T \rightarrow F$

  $F \rightarrow (E)$

  $F \rightarrow id$

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



LEADING(E) = {

LEADING(T) = {

LEADING(F) = {

- Grammar :

  $E \rightarrow E + T$

  $E \rightarrow T$

  $T \rightarrow T*F$

  $T \rightarrow F$

  $F \rightarrow (E)$

  $F \rightarrow id$

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

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

LEADING(E) = { +,

LEADING(T) = { *,

LEADING(F) = { (, id

- Grammar :

  $E \rightarrow E + T$

  $E \rightarrow T$

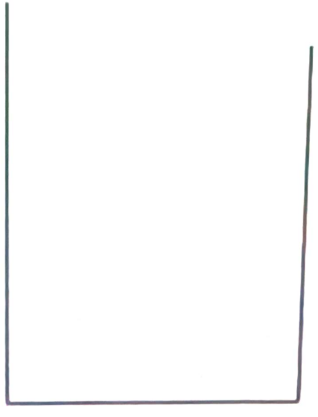
  $T \rightarrow T*F$

  $T \rightarrow F$

  $F \rightarrow (E)$

  $F \rightarrow id$

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



LEADING(E) = { +, id, (, * }

LEADING(T) = { *, id, (}

LEADING(F) = { (, id}

Main Algorithm

```
begin
   for ∀A,a  L[A,a] := false;
   for each production of the form
   A→ αa or A→αaB do Install(A,a);
   while stack is not empty do
   begin
        pop top pair (B,a) from stack;
        for each A→αB do Install (A,a)
   end;
end;
```

Grammar :

$E \rightarrow E + T$

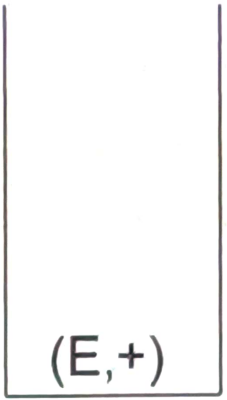
$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$




```
begin
  for ∀A,a  L[A,a] := false;
  for each production of the form
  A→ αa or A→αaB do Install(A,a);
  while stack is not empty do
  begin
    pop top pair (B,a) from stack;
    for each A→αB do Install (A,a)
  end;
end;
```

TRAILING(E) ={

TRAILING(T) ={

TRAILING(F) ={

Grammar :

$E \rightarrow E + T$

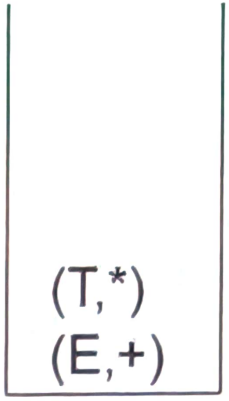
$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$




```
begin
  for ∀A,a  L[A,a] := false;
  for each production of the form
  A→ αa or A→αaB do Install(A,a);
  while stack is not empty do
  begin
    pop top pair (B,a) from stack;
    for each A→αB do Install (A,a)
  end;
end;
```

TRAILING(E) ={+,

TRAILING(T) ={*,

TRAILING(F) ={

Grammar :

$E \rightarrow E + T$

$E \rightarrow T$

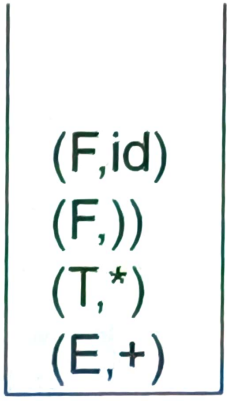
$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

```
begin
  for ∀A,a  L[A,a] := false;
  for each production of the form
  A→ αa or A→αaB do Install(A,a);
  while stack is not empty do
  begin
    pop top pair (B,a) from stack;
    for each A→αB do Install (A,a)
  end;
end;
```



TRAILING(E) ={+,

TRAILING(T) ={*,

TRAILING(F) ={),

Grammar :

$E \rightarrow E + T$

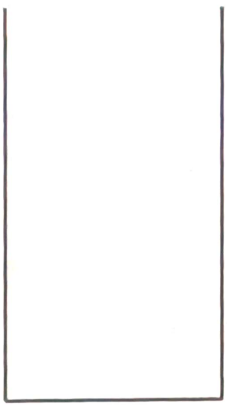
$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$



```
begin
  for ∀A,a  L[A,a] := false;
  for each production of the form
  A→ αa or A→αaB do Install(A,a);
  while stack is not empty do
  begin
    pop top pair (B,a) from stack;
    for each A→αB do Install (A,a)
  end;
end;
```

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 suggests 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 i.e., $a \doteq b$

Rules for finding Operator Precedence Relation

2. If $S \rightarrow \alpha aA\beta$ and $A \stackrel{+}{\Rightarrow} \gamma b\delta$ such that γ is either ε or a single nonterminal then $a \lessdot b$, also $\$ \lessdot b$ if $S \stackrel{+}{\Rightarrow} \gamma b\delta$ where γ is either ε or a single nonterminal

   b is in Leading A

From the above rule the string that includes a is "$\alpha a\gamma b\delta\beta$". And the rightmost derivation is

$S \Rightarrow \alpha aA\beta \stackrel{+}{\Rightarrow} \alpha a\gamma b\delta\beta$

Therefore $\gamma b\delta$ must be reduced to A first and then the string $\alpha aA\beta$ should be reduced to S that is $a \lessdot b$

Rules for finding Operator Precedence Relation

3. If $S \rightarrow \alpha Ab\beta$ and $A \stackrel{+}{\Rightarrow} \gamma a\delta$ where δ is either ε or a single nonterminal then $a \doteq b$, also $a \gtrdot \$$ if $S \stackrel{*}{\Rightarrow} \gamma a\delta$ where δ is either ε or a single nonterminal

a is in Trailing A

From the above rule following derivation is possible

$S \Rightarrow \alpha Ab\beta \stackrel{+}{\Rightarrow} \alpha\ \gamma a\delta\ b\beta$

That is $\gamma a\delta$ must be reduced first to A and then $\alpha Ab\beta$ should be reduced to A. Therefore $a \gtrdot b$

Concept for finding operator precedence

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

1. If x_i and x_{i+1} are terminals then set $x_i \doteq x_{i+1}$ (rule 1)
2. If x_i and x_{i+2} are terminals and x_{i+1} is a nonterminal then set $x_i \doteq x_{i+2}$ (rule1)
3. If x_i is a terminal and x_{i+1} is a nonterminals then for all a , a is in LEADING(x_{i+1}) set $x_i \lessdot a$ (rule 2)
4. If x_i is a nonterminal and x_{i+1} is a terminals then for all a , a is in TRAILING(x_i) set $a \gtrdot x_{i+1}$ (rule 3)
5. Set \$ $\lessdot$ a for all a in LEADING(S) and set b $\gtrdot$ \$ for all b in TRAILING(S) (rule 2 and 3)

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

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

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

- Example Grammar
- S → aAcBe
- A → Ab
- A → b
- B → d
- LEADING(S) = {a}
- LEADING(A) = {b}
- LEADING(B) = {d}
- TRAILING(S) = {e}
- TRAILING(A) = {b}
- TRAILING(B) = {d}

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

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

Consider a string "abbcde"

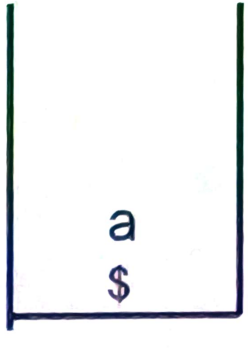
	a	b	c	d	e	$
a		$\lessdot$	$\doteq$			
b		$\gtrdot$	$\gtrdot$			
c				$\lessdot$	$\doteq$	
d					$\gtrdot$	
e						$\gtrdot$
$	$\lessdot$					

Stack		Input	Action
$	$\lessdot$	abbcde$	shift
$a	$\lessdot$	bbcde$	shift
$ab	$\gtrdot$	bcde$	reduce A $\rightarrow$b
$aA	$\lessdot$	bcde$	shift
$aAb	$\gtrdot$	cde$	reduce A $\rightarrow$Ab
$aA	$\doteq$	cde$	shift
$aAc	$\lessdot$	de$	shift
$aAcd	$\gtrdot$	e$	reduce B $\rightarrow$d
$aAcB	$\doteq$	e$	shift
$aAcBe	$\gtrdot$	$	reduce S $\rightarrow$ aAcBe
$ S		$	accept

Example Grammar

S → aAcBe
A → Ab|b
B → d

Consider a string "abbcde"



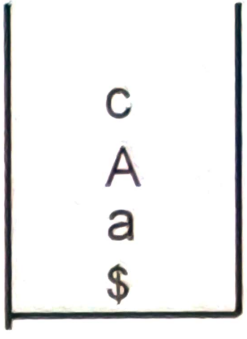
b b c d e

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

Example Grammar

$S \rightarrow aAcBe$
$A \rightarrow Ab|b$
$B \rightarrow d$

Consider a string "abbcde"



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

e

Grammar :

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

LEADING(E) = {+, *, id, (}

LEADING(T) = {*, id, (}

LEADING(F) = {id, (}

TRAILING(E) = {+, *, id,) }

TRAILING(T) = {*,id ,) }

TRAILING(F)= { id,) }

	+	*	(	)	id	$
+		$\lessdot$	$\lessdot$		$\lessdot$	
*						
(				$\doteq$		
)						
id						
$						

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

If x_i is a terminal and x_{i+1} is a nonterminals then for all a , a is in LEADING(x_{i+1}) set $x_i \lessdot a$ (rule 2)

Operator precedence relations

Grammar :

$E \rightarrow E + T$ ←

$E \rightarrow T$

$T \rightarrow T*F$ ←

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

	+	*	(	)	id	$
+	$\gtrdot$	$\lessdot$	$\lessdot$		$\lessdot$	
*	$\gtrdot$	$\gtrdot$	$\lessdot$		$\lessdot$	
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)	$\gtrdot$	$\gtrdot$				
id	$\gtrdot$	$\gtrdot$				
$						

LEADING(E) = {+, *, id, (}

LEADING(T) = {*, id, (}

LEADING(F) = {id, (}

TRAILING(E) = {+, *, id,) }

TRAILING(T) = {*, id,) }

TRAILING(F)= {id.) }

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

If x_i is a nonterminal and x_{i+1} is a terminals then

for all a , a is in TRAILING(x_i) set a $\gtrdot x_{i+1}$ (rule 3) ←

Rules for finding Operator Precedence Relation

1. If $S \rightarrow \alpha a \beta b \gamma$ where β is either a single nonterminal or ε then $a \doteq b$.
2. If $S \rightarrow \alpha a A \beta$ and $A \stackrel{+}{\Rightarrow} \gamma b \delta$ such that γ is either ε or a single nonterminal then $a \lessdot b$, also $\$ \lessdot b$ if $S \stackrel{+}{\Rightarrow} \gamma b \delta$ where γ is either ε or a single nonterminal
3. If $S \rightarrow \alpha A b \beta$ and $A \stackrel{+}{\Rightarrow} \gamma a \delta$ where δ is either ε or a single nonterminal then $a \gtrdot b$, also $a \gtrdot \$$ if $S \stackrel{+}{\Rightarrow} \gamma a \delta$ where δ is either ε or a single nonterminal

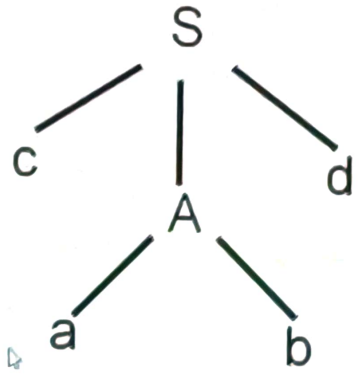
Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

Let the input string be cad



Example String :c a d

Input c a d

Failure – back track

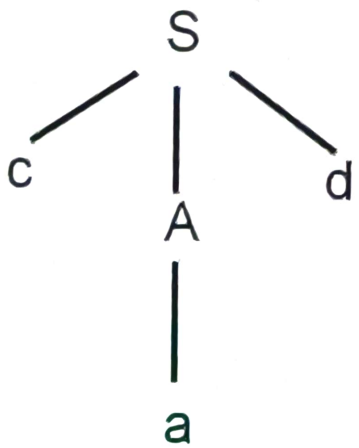
Consider the grammar

$S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

Let the input string be cad



Example String :c a d

Input c a d

Match is obtained

Back tracking was necessary because, we did not know whether we should expand A to ab or a.

Left Factoring

If $A \rightarrow \alpha\beta|\alpha\gamma$ 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

$$A \rightarrow \alpha A'$$

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

Example

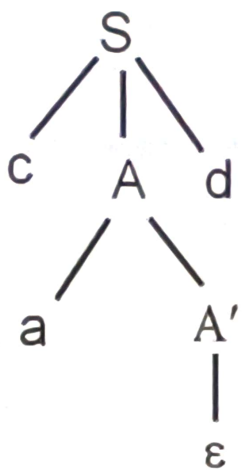
$S \rightarrow cAd$

$A \rightarrow ab$

$A \rightarrow a$

Input c a d

Parsing cad



Changed grammar

$S \rightarrow cAd$

$A \rightarrow aA'$

$A' \rightarrow b \mid \varepsilon$

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

$A \rightarrow \alpha A'$

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

Recursive – Descent Parsing

$S \rightarrow cAd$

$A \rightarrow aA'$

$A' \rightarrow b \mid \varepsilon$

```
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;
```

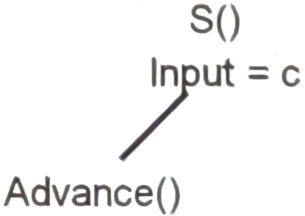
```
Procedure S()
begin
    if input symbol = 'c' then
    begin
        Advance();
        A();
        if input symbol = 'd' then
        Advance() else error();
    end;
end;
```

```
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

c a d



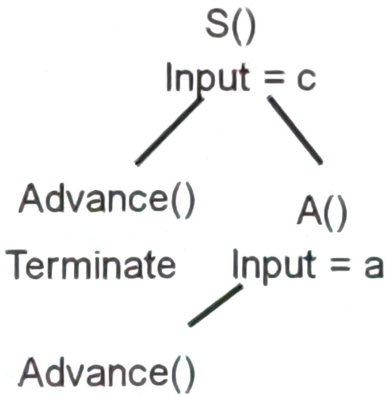
```
Procedure S()
begin
    if input symbol = 'c' then
    begin
        Advance();
        A();
        if input symbol = 'd' then
        Advance() else error();
    end;
end;
```

```
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

c a d



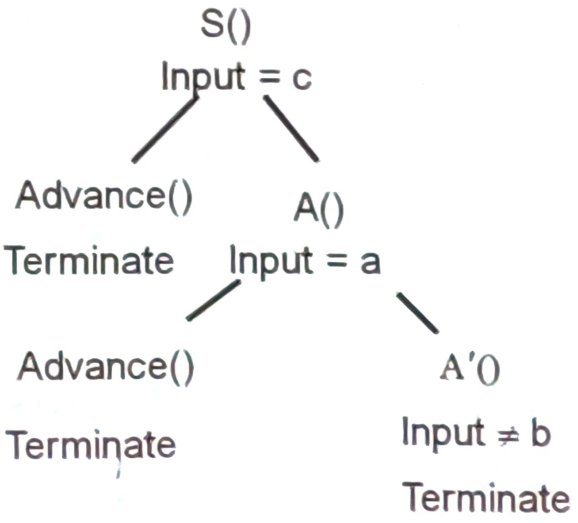
```
Procedure S()
begin
    if input symbol = 'c' then
    begin
        Advance();
        A();
        if input symbol = 'd' then
        Advance() else error();
    end;
end;
```

```
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

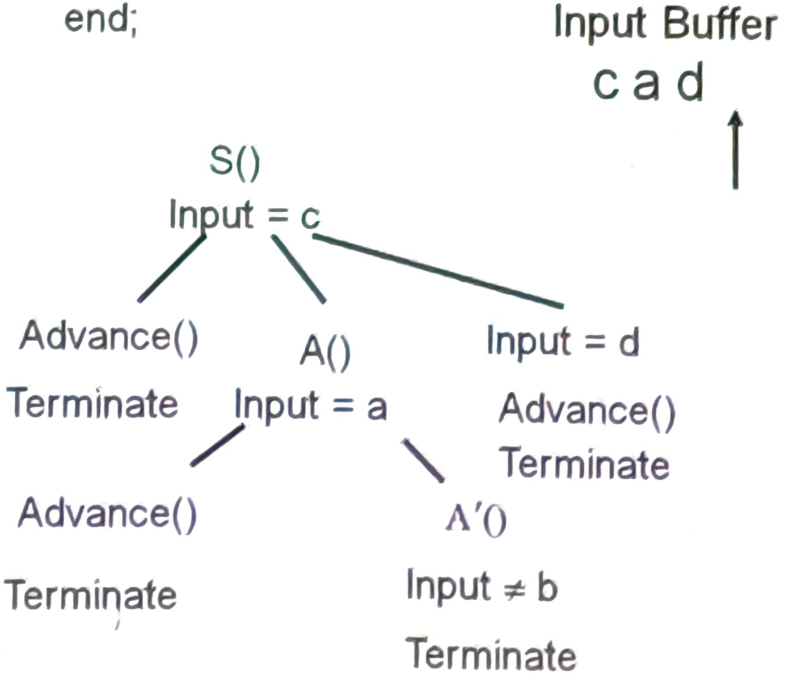
c a d



```
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;
```



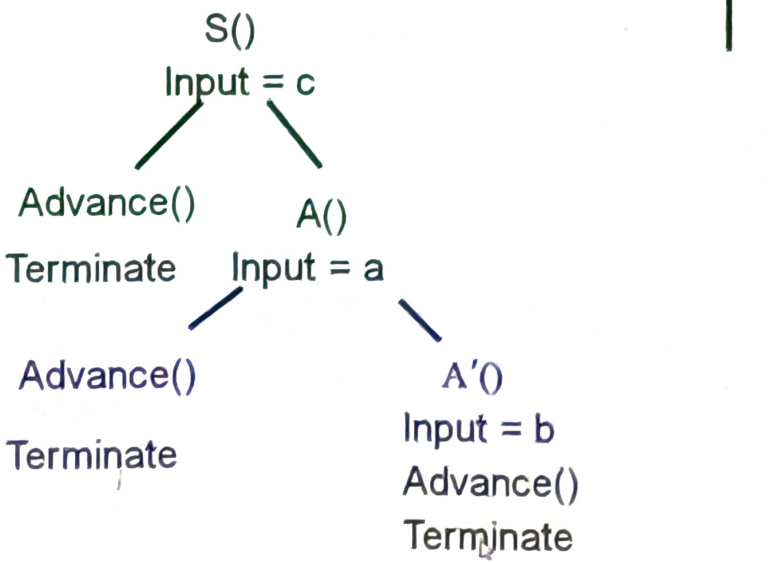
```
Procedure S()
begin
    if input symbol = 'c' then
    begin
        Advance();
        A();
        if input symbol = 'd' then
        Advance() else error();
    end;
end;

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

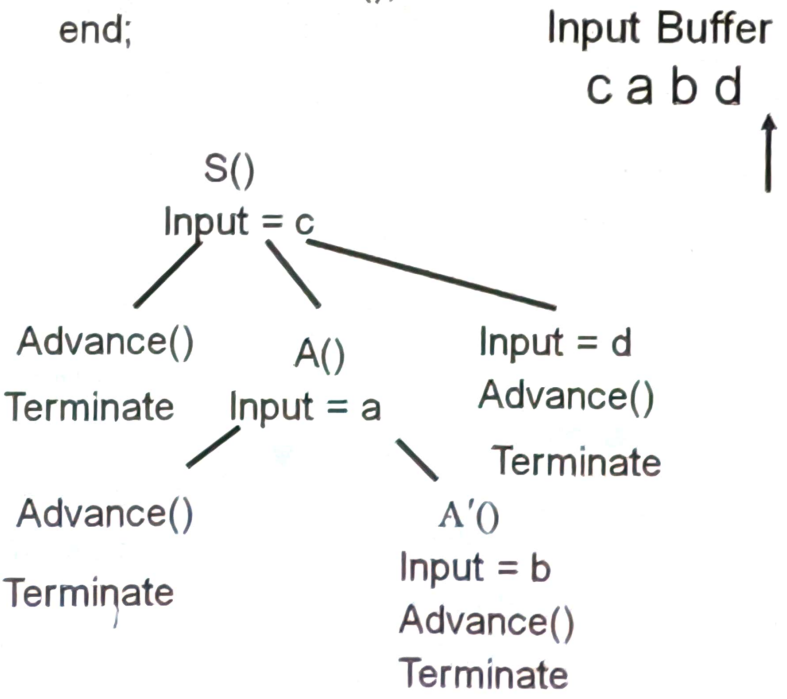
c a b d



```
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;
```



$S \rightarrow iCtS \mid iCtSeS \mid a$

$C \rightarrow b$

Upon left factoring the grammar

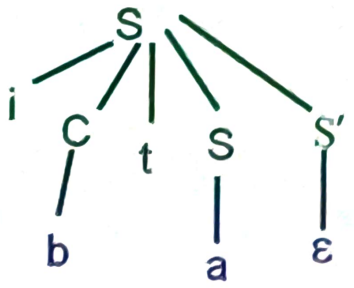
Becomes

$S \rightarrow iCtS\ S' \mid a$

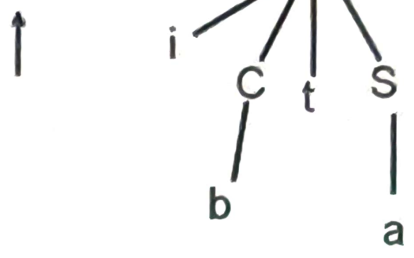
$S' \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

ibta

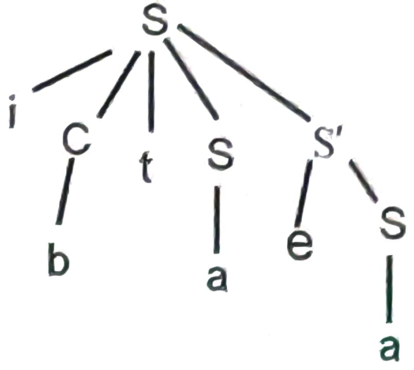


i b t a e a



Failure

ibtaea

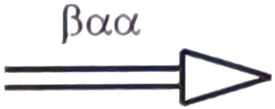


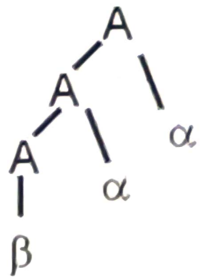
Left Recursion

Consider the grammar

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

This is a left recursive grammar.



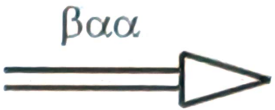


Left Recursion

Consider the grammar

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

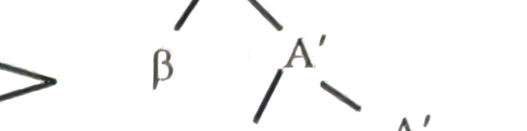
This is a left recursive grammar.



This can be modified to

$A \rightarrow \beta \quad A'$

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



Method for Eliminating Immediate Left Recursion

Consider the grammar

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

where no β_i begins with A

We can eliminate the left recursion as follows

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

Non Immediate Left Recursion

Consider the grammar

$S \rightarrow Aa \mid b$

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

The nonterminal S is left recursive because

$S \Rightarrow Aa \Rightarrow Sda$

The following algorithm will eliminate all left recursion if the grammar has no cycle, that is, it has a derivation of the form $A \overset{+}{\Rightarrow} A$ or has an ε production

Algorithm

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 productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

   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 as S, A, i.e., 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 the following A_1 productions available $S \rightarrow Aa$ and $S \rightarrow b$
- After replacing S in $A \rightarrow Sd$ we get $A \rightarrow Aad$ and $A \rightarrow bd$
- N ext we eliminate immediate left recursion among the current A productions 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

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bdA' \mid eA'$$

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

Example

Consider the grammar

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

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

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Eliminating the immediate left recursion we get

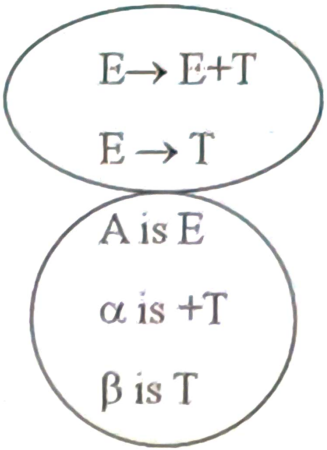
$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$



$E \Rightarrow TE'$

$\Rightarrow FT'E'$

$\Rightarrow F\varepsilon\ E'$

$\Rightarrow id\ E'$

$\Rightarrow id + TE'$

$\Rightarrow id + FT'E'$

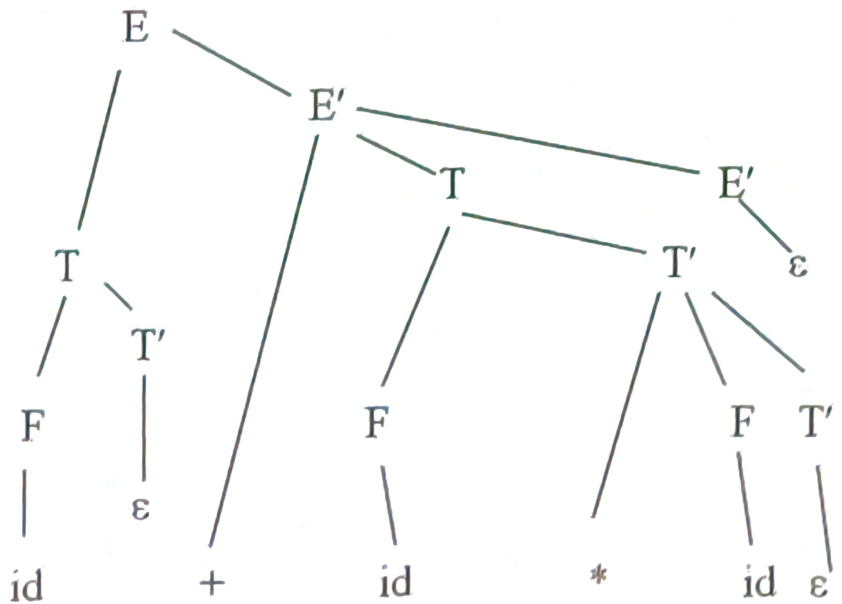
$\Rightarrow id + idT'E'$

$\Rightarrow id + id*FT'E'$

$\Rightarrow id + id*idT'E'$

$\Rightarrow id + id*idE'$

$\Rightarrow id + id*id$

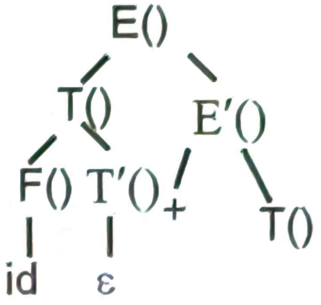


```
Procedure E()
begin
    T();
    E'();
end;
Procedure E'();
begin
    if input = '+' then
    begin
     Advance();
     T();
     E'();
    end;
end;
```

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

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

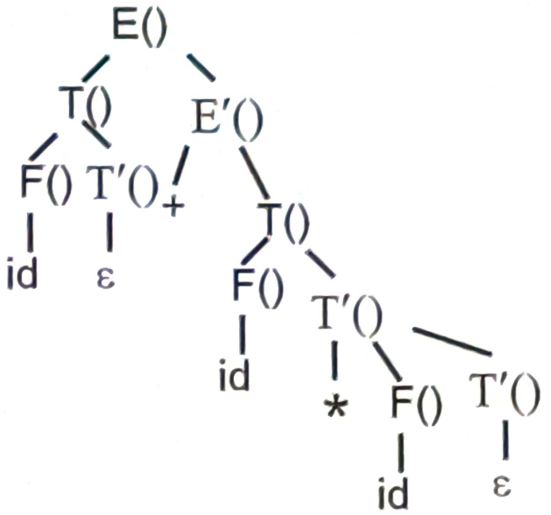
```
Procedure E()
begin
    T();
    E'();
end;
Procedure E'();
begin
    if input = '+' then
    begin
      Advance();
      T();
      E'();
    end;
end;
```



Input Buffer

id + id * id

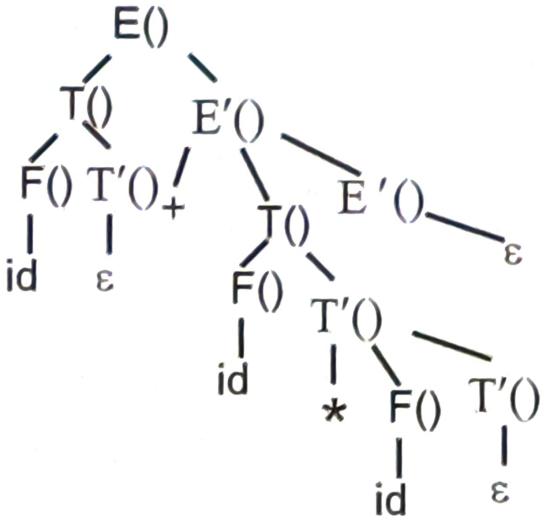
```
Procedure T()
begin
    F();
    T′();
end;
Procedure  T′()
begin
    if input =*  ' then
    begin
      Advance();
      F();
      T′()
    end;
end;
```



Input Buffer

id + id * id

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



Input Buffer

id + id * id