Parul® University

System Programming

Prof. Shaleen Shukla, Assistant Professor
Information Technology

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-3

Scanning and Parsing

Parul® University

DIGITAL LEARNING CONTENT

Outline

- Programming Language Grammars
- Classification of Grammar
- Ambiguity in Grammatic Specification,
- Scanning,
- Parsing
- Top Down Parsing
- Bottom up Parsing
- Language Processor Development Tools
- LEX, YACC

Parul®
University

DIGITAL LEARNING CONTENT

Programming Language Grammars

Some Basics.....

Symbol = 0, 1, a, b, A, B.....

String: Sequence of the symbols...

String = a, b, aa, ab, ba, bb, abb....

Some Basics.....

Language: Language is collection of strings.

Σ= {a,b}

L1=set of all string of length 2

={aa, ab, ba, bb}

L2=set of all string of length 3

={aaa, aab, aba, abb, baa, bab, bba, bbb}

L3=set of all string start with a

={a, aa, ab, aaa, abb, aba, aaaab.....}

Finite

Infinite

•ϵ is a special symbol of length 0

Regular Expression

- A regular expression is a sequence of characters that define a pattern.

One or More Occurrences

+

L1=One or More Occurrences of a=

a+

a

aa

aaa

aaaa

aaaaa.....

Infinite .....

Zero or More Occurrences

*

Regular Expression

L1 = Zero or More Occurrences of a= a*

ϵ

a

aa

aaa

aaaa

aaaaa.....

Infinite .....

Regular Expression Examples

1. 0 or 1
   0 | 1
2. 0 or 11 or 111
   0 | 11 | 111
3. String having zero or more *a*.

Regular Expression Examples

7. 0 or more occurrence of either a or b or both $(a|b)^*$
8. 1 or more occurrence of either a or b or both $(a|b)^+$

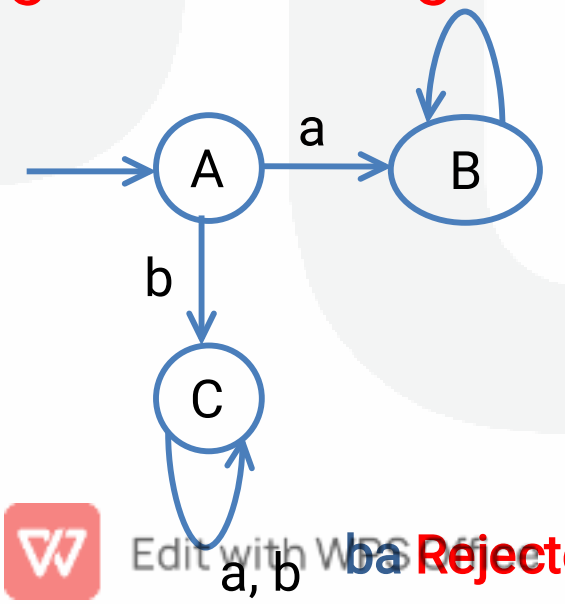
Regular Expression Examples

13. All string of a and b starting with a
    $a(a|b)^*$
14. String of 0 and 1 ends with 00
    $(0|1)^*00$
15. String ends with abb
    $(a|b)^*abb$

Finite Automata

- Finite Automata are recognizers.
  - FA simply say “Yes” or “No” about each possible input string.
  - Example: “language of all string starts with a”

1. String: aab
2. String: ba

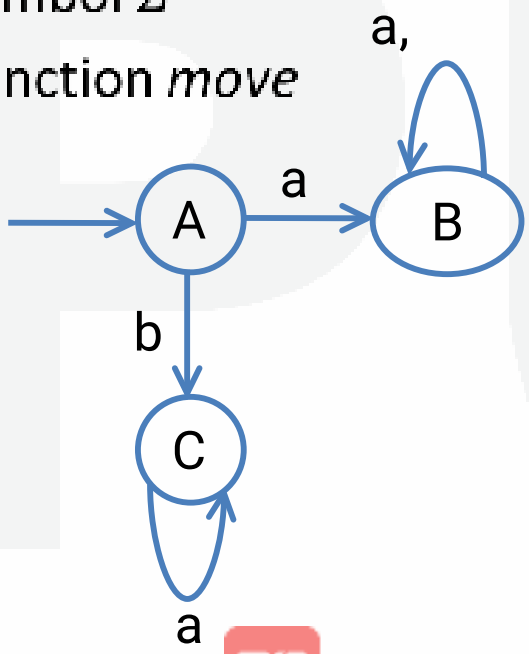


aab
Accepted

ba Rejected

Definition: Finite Automata

- Finite Automata is a mathematical model consist of:
  1. Set of states S.
  2. Set of input symbol Σ
  3. A transition function *move*
  4. Initial state S_0
  5. Final state F



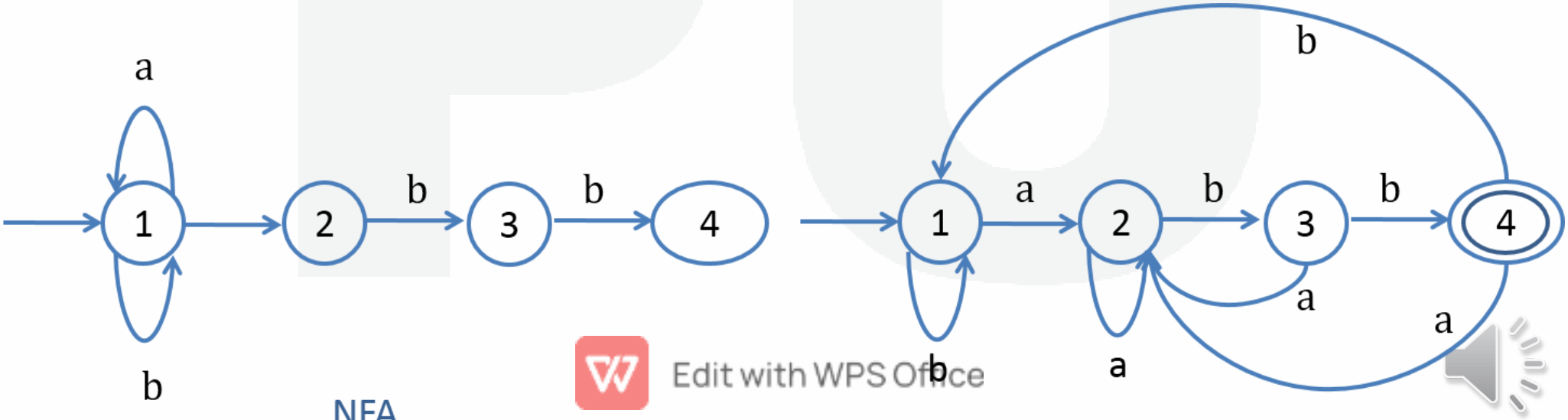
S	**A, B, C**
S_0	**A**
F	**B**
Σ	**a, b**
move	**Move(A,a)=B** **Move(A,b)=C** **....**

Types of Finite Automata

There are two types of finite automata:

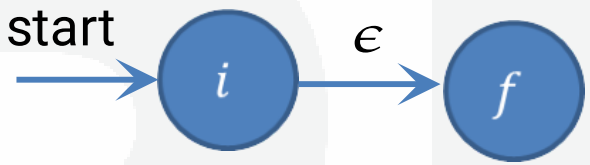
Nondeterministic finite automata (NFA): There are no restrictions on the edges leaving a state. There can be several with the same symbol as label and some edges can be labeled with ϵ.

Deterministic finite automata (DFA): have for each state exactly one edge leaving

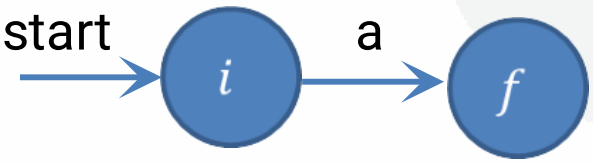


RE to NFA using Thompson's rule

1. For $\in$, construct the NFA

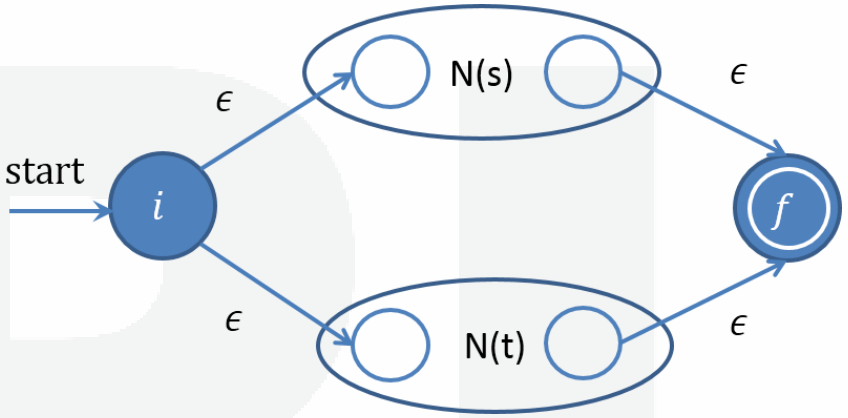


2. For a in Σ, construct the NFA

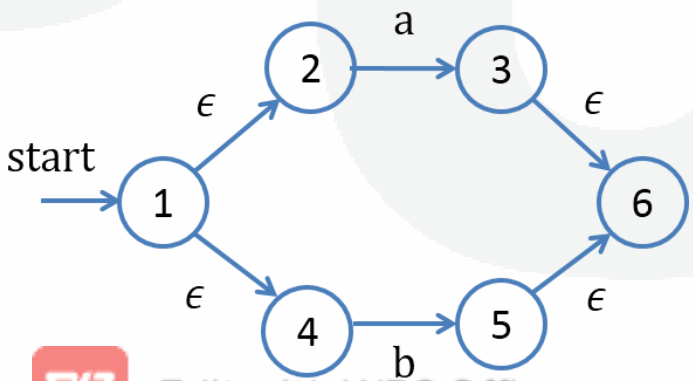


RE to NFA (Thompson's construction)

3. For regular expression $s|t$

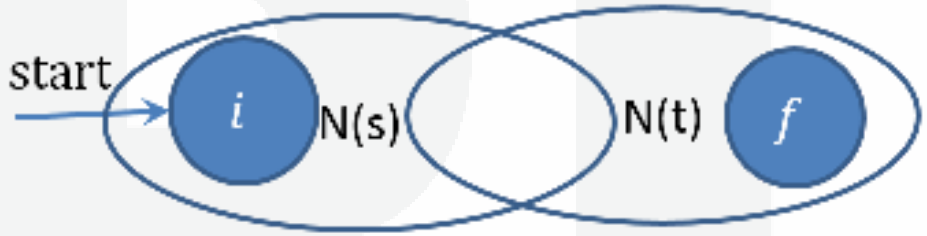


Ex: (a|b)



RE to NFA (Thompson's construction)

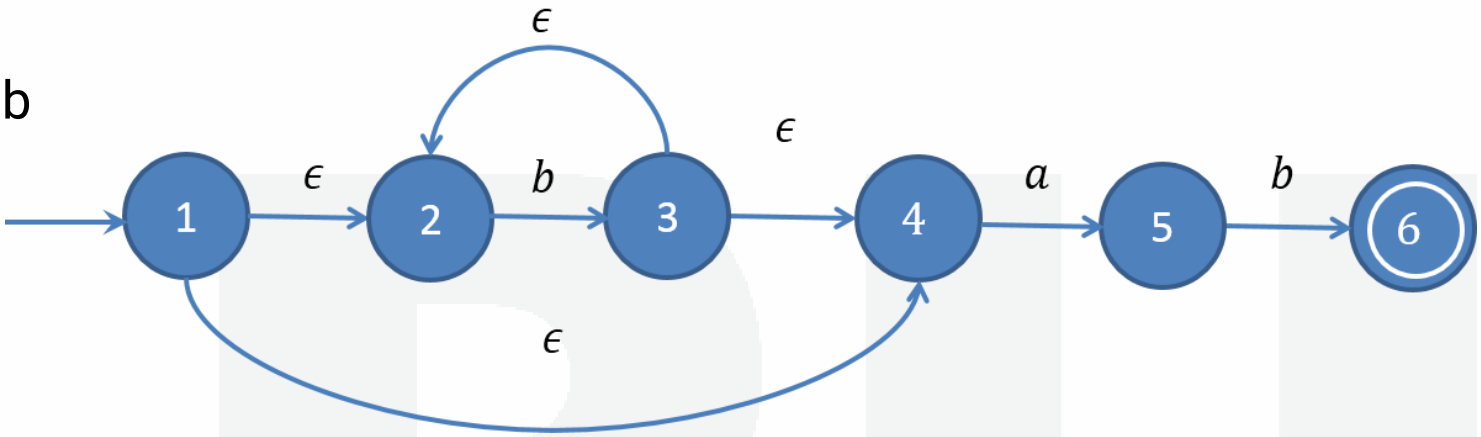
4. For regular expression st



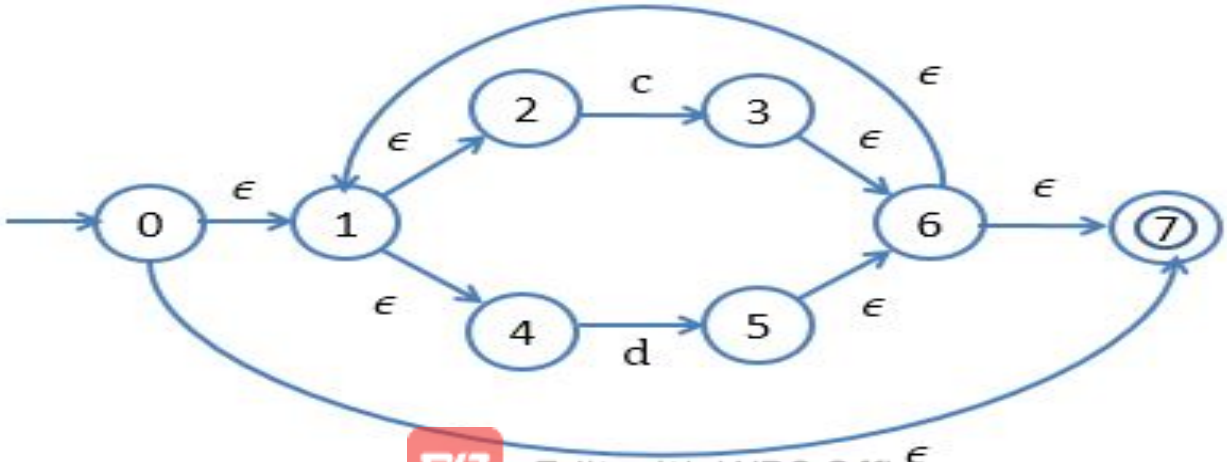
Ex: ab

RE to NFA (Thompson's construction)

b*ab



(c|d)*



Parul®
University

DIGITAL LEARNING CONTENT

Conversion from NFA to DFA

Subset construction algorithm

Input: An NFA N.

Output: A DFA D accepting the same language.

Method: Algorithm construct a transition table $Dtran$ for D. We use the following operation:

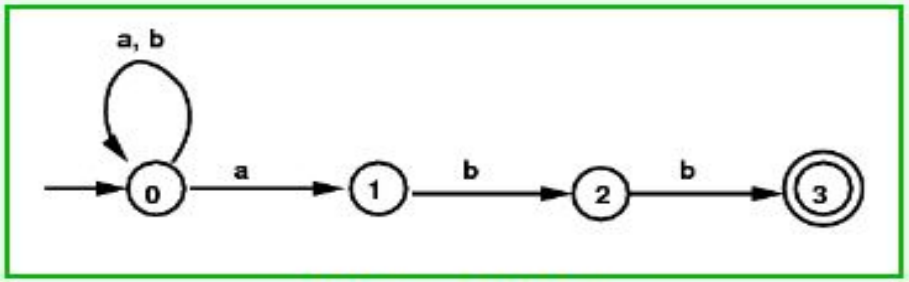
OPERATION	DESCRIPTION
$\in - closure(s)$	Set of NFA states reachable from NFA state s on $\in-$ transition alone.
$\in - closure(T)$	Set of NFA states reachable from some NFA state s in T on $\in-$ transition alone.
$\boldsymbol{Move}\ (T, a)$	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Subset construction algorithm

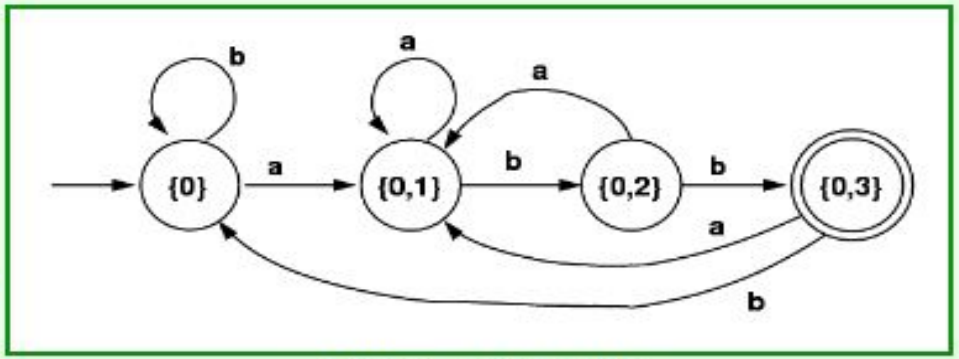
initially $\in -closure(s_0)$ be the only state in $Dstates$ and it is unmarked;
while there is unmarked states T in $Dstates$ ***do begin***
mark T;
for each input symbol a ***do begin***
$U = \epsilon - closure\big(move(T, a)\big)$;
if U is not in $Dstates$ ***then***
add U as unmarked state to $Dstates$;
$Dtran[\,T, a\,] = U$
end
end

Conversion from NFA to DFA

(a|b)*abb



NFA for (a|b)*abb



DFA for (a|b)*abb

Subset Algorithm (Tables)

NFA, (a|b)*abb		
State	a	b
0 (start)	0,1	0
1	–	2
2	–	3
3 (term)	–	–

DFA, (a|b)*abb		
States	a	b
{0} (start)	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3} (term)	{0,1}	{0}

Grammar

- A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where,

Σ= alphabet of L_G, the set of terminals

SNT= set of NTs

S= distinguished symbol (start symbol)

P= set of production

- Example:

Start Symbol

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

Nonterminals

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

Productions

Terminals

Types of Grammar

There are four types of grammar.
1.Type-3 grammar
2.Type-2 grammar
3.Type-1 grammar
4.Type-0 grammar

Type-3 grammar

- Type-3 grammars are characterized by productions of the form
  A → tB|t or
  A → Bt|t
- The specific form of the RHS alternatives—namely a single terminal symbol or a string containing a single terminal symbol and a single nonterminal symbol.
- Type-3 grammars are also known as linear grammars or regular grammars.

Type-2 grammar

- A typical Type-2 production is of the form

  $A \rightarrow \pi$

- which can be applied independent of its context. These grammars are therefore known as context free grammars (CFG). CFGs are ideally suited for programming language specification

Type-1 grammar

- A production of a Type-1 grammar has the form

$\alpha A \beta \rightarrow \alpha \pi \beta$

- Here, a string π can be replaced by 'A' (or vice versa) only when it is enclosed by the strings α and β in a sentential form. These grammars are also not relevant for programming language specification since recognition of programming language constructs is not context sensitive in nature.

Type-0 grammar

- These grammars are known as phrase structure grammars. Their productions are of the form

$\alpha \rightarrow \beta$

- where both α and β can be strings of terminal and nonterminal symbols. Such productions permit arbitrary substitution of strings during derivation or reduction, hence they are not relevant to specification of programming languages.

Parul®
University

DIGITAL LEARNING CONTENT

Derivation & Reduction

Derivation

- Let production P1 of grammar G be the form

$$A \rightarrow \alpha$$

- String: $\beta = \gamma A \theta$

$$\gamma \alpha \theta$$

- Replacement of A by α in string β constitutes a derivation according to production P1.
- The derivation operation helps to generate valid string.
- Types of derivations are:
  1. Leftmost derivation
  2. Rightmost derivation

Leftmost Derivation

- A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.
- Let us consider a Grammar

  S→S+S | S-S | S*S | S/S | (S) | a
- String: a*a-a

  S

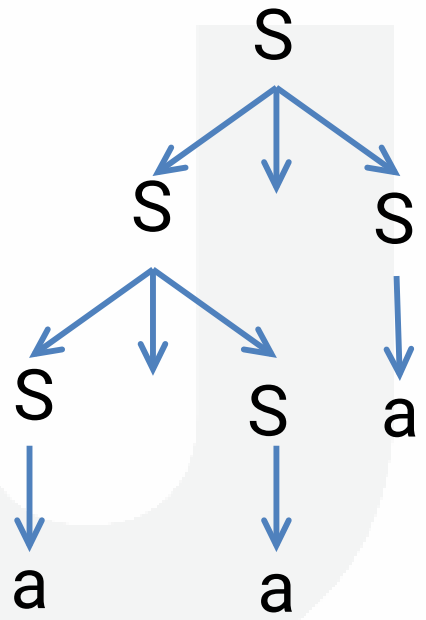
  S-S

  S*S-S

  a*S-S

  a*a-S

  a*a-a



Leftmost Derivation

Parse Tree

Rightmost Derivation

- A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
- Let us consider a Grammar

  S→S+S | S-S | S*S | S/S | (S) | a
- String: a*a-a

  S

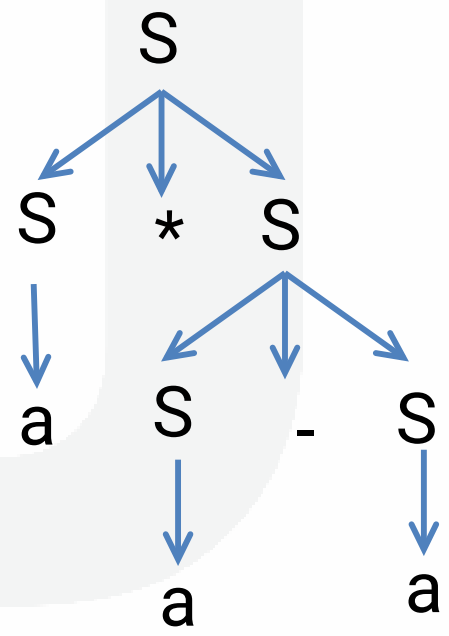
  S*S

  S*S-S

  S*S-a

  S*a-a

  a*a-a



Rightmost Derivation

Parse Tree

Exercise

1. S→A1B

A→0A | ϵ

B→0B | 1B | ϵ (String: 1001) Perform leftmost derivation.

2. E→E+E | E*E | id | (E) | -E

String : -(id+id) Perform rightmost derivation

Reduction

- Let production P1 of grammar G be the form P1: $A \rightarrow \alpha$ And let σ be a string such that $\sigma = \gamma\ \alpha\ \theta$, then replacement of α by A in string σ constitutes a reduction according to production P1.
- Reduction operation helps to recognize valid strings.
- Let us consider a Grammar: S→S+S | S-S | S*S | S/S | (S) | a
- String: a*a-a

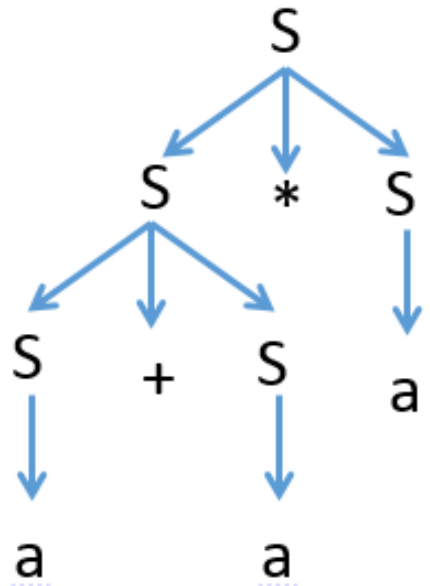
a*a-a
s*a-a
s*s-a
s-a
s-s
s

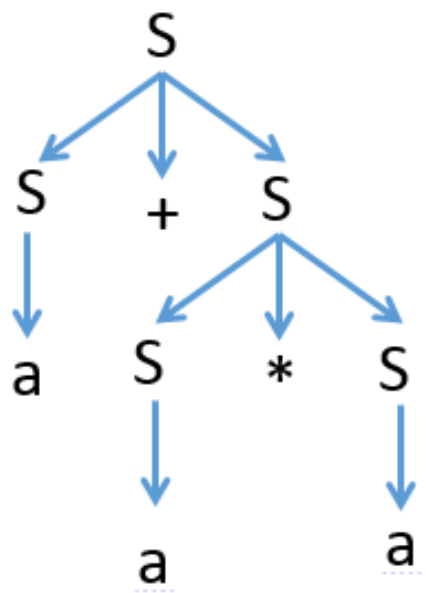
Ambiguous Grammar

- A context free grammar G is ambiguous if there is at least one string in L(G) having two or more distinct derivation tree.
- Example: S→S+S | S-S | S*S | S/S | (S) | a
- String: a+a*a

S
S*S
S+S*S
a+S*S
a+a*S
a+a*a

S
S+S
a
a
a
a





Here, we have two left most derivation for string *a+a*a* hence, above grammar is ambiguous.

Exercise

Check whether following grammars are ambiguous or not:

1.S→ aS | Sa | ϵ (string: aaaa)

2.S→ aSbS | bSaS | ϵ (string: abab)

3.S→SS+ | SS* | a (string: aa+a*)

Parsing

Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid sentence of given grammar.
Types of parsing are:

1.**Top down parsing**: In top down parsing parser build parse tree from top to bottom.
2.**Bottom up parsing**: Bottom up parser starts from leaves and work up to the root.

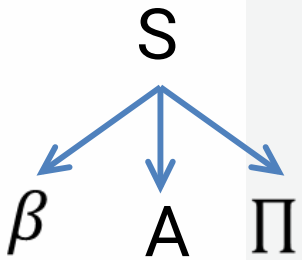
Top down parsing and Bottom up parsing

- Following are some of the important differences between Top Down Parsing and Bottom Up Parsing.

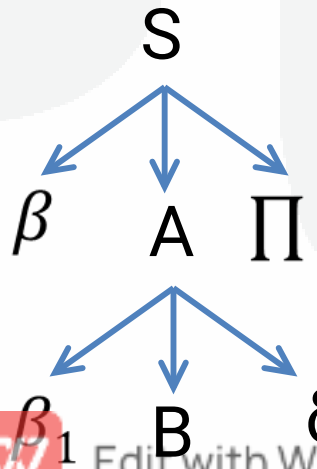
Sr. No.	Key	Top Down Parsing	Bottom Up Parsing
1	Strategy	Top down approach starts evaluating the parse tree from the top and move downwards for parsing other nodes.	Bottom up approach starts evaluating the parse tree from the lowest level of the tree and move upwards for parsing the node.
2	Attempt	Top down parsing attempts to find the left most derivation for a given string.	Bottom up parsing attempts to reduce the input string to first symbol of the grammer.
3	Derivation Type	Top down parsing uses leftmost derivation.	Bottom up parsing uses the rightmost derivation.
4	Objective	Top down parsing searches for a production rule to be used to construct a string.	Bottom up parsing searches for a production rule to be used to reduce a string to get a starting symbol of grammer.

Top-up Parsing Example

1. Current sentential form (CSF) = 'S'
2. Let CSF be of the form $\beta A \prod$, such that β is a string of terminals and A is the leftmost NT in CSF. Exit with success if CSF=α.
3. Make a derivation $A \rightarrow \beta_1 B \delta$ according to a production $A = \beta_1 B \delta$ of G such that β_1 is a string of Terminals. This makes $CSF = \beta\beta_1 B \delta \prod$.
4. Go to step 2.



CFS=$\beta A \prod$



CFS=$\beta\ \beta_1 B \delta \prod$

Bottom-up Parsing Example

Example:

Input string : a + b * c

Production rules:

S → E
E → E + T
E → E * T
E → T
T → id

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c
T + b * c
E + b * c
E + T * c
E * c
E * T
E
S

Backtracking

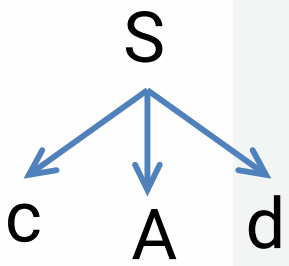
- In Backtracking expansion of non-terminal symbol we choose one alternative and if any mismatch occurs then we try another alternative.

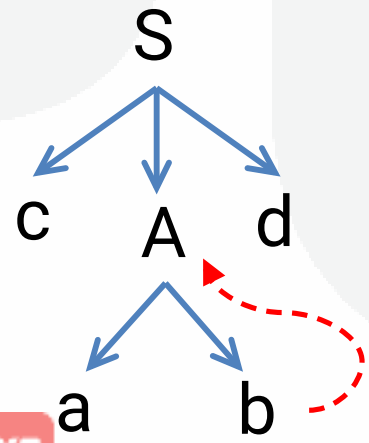
Example:

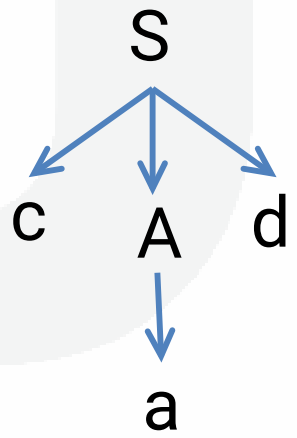
S→ cAd

A→ ab | a

Input string: cad







Exercise

1. E→ 5+T | 3-T
T→ V | V*V | V+V
V→ a | b
(String: 3-a+b)

Parul®
University

DIGITAL LEARNING CONTENT

Left Recursion & Left Factoring

Left Recursion Removal

Removal of Left recursion from production

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

$$A \rightarrow A'$$

$$A' \rightarrow A' \mid \epsilon$$

Example: Left Recursion Removal

E→E+T | T

E→TE'

E'→+TE' | ε

T→T*F | F

T→FT'

T'→*FT' | ε

X→X%Y | Z

X→ZX'

X'→%YX' | ε

Example: Left recursion removal

S→ Aa | b

A→ Ac | Sd | ε

Here, Non terminal S is left recursive because:

S→Aa→Sda

S→ Aa | b

A→ Ac

A→ Aad

A→ bd

A→ ε

A→Ac | Aad | bd | ε

Now, remove left recursion

S→ Aa | b

A→ bdA' | A'

A'→ cA' | adA' | ε

Exercise

1. A→Abd | Aa | a
   B→Be | b
2. A→AB | AC | a | b
3. S→A | B

A→ABC | Acd | a | aa

B→Bee | b

4. Exp→Exp+term | Exp-term | term

Left factoring Removal

Removal of Left factoring

$$A \rightarrow \alpha\beta \mid \alpha\delta \longrightarrow \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta \mid \delta \end{array}$$

Example: Left factoring removal

S→aAB | aCD

S→aS'

S'→AB | CD

A→ xByA | xByAzA | a

A→ xByAA' | a

A'→ Є | zA

A→ aAB | aA |a

A→aA'

A'→AB | A | ϵ

A'→AA" | ϵ

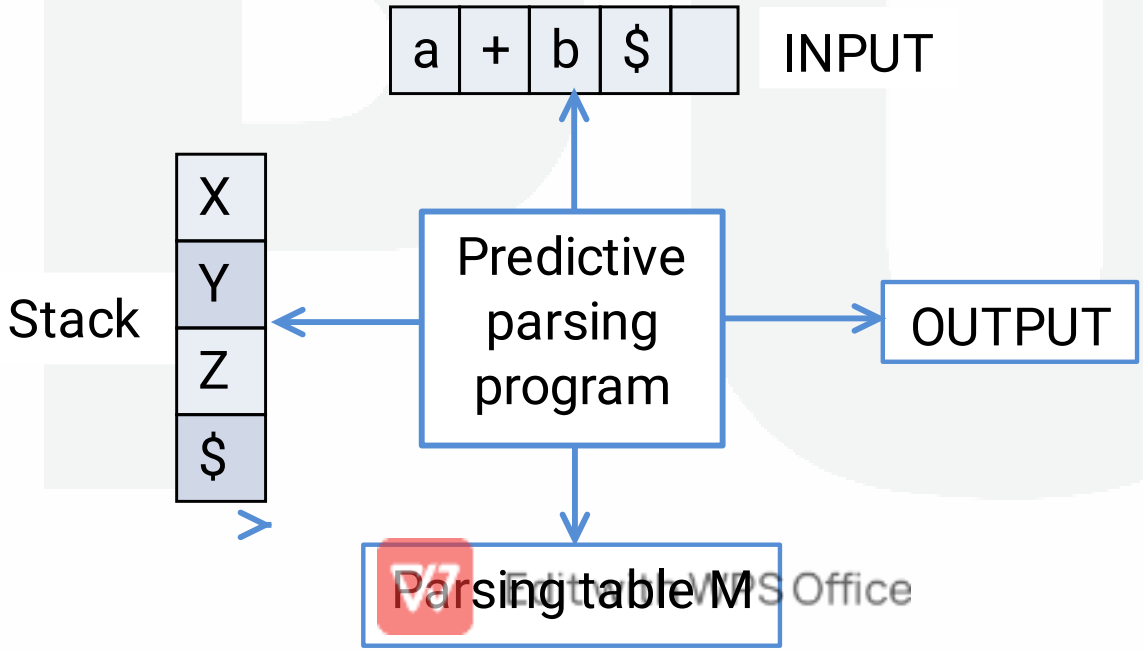
A"→B | ϵ

Exercise

1. S→iEtS | iEtSeS | a

LL(1) Parsing

- LL(1) is non recursive top down parser.
- LL (1) – the first L indicates input is scanned from left to right. The second L means it uses leftmost derivation for input string and 1 means it uses only input symbol to predict the parsing process.



LL(1) Parsing

Steps to construct LL(1) parser

1.Remove left recursion / Perform left factoring (if any).

2.Compute FIRST and FOLLOW of Non terminals.

3.Construct predictive parsing table.

4.Parse the input string using parsing table

Rules to compute First of NT

1. If $A \rightarrow \alpha$ and α is terminal, add α to $FIRST(A)$.
2. If $A \rightarrow \in$, add $\in$ to $FIRST(A)$.
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \ldots . Yk$ is a production, then place a in $FIRST(X)$ if for some i, a is in $FIRST(Yi)$, and ϵ is in all of $FIRST(Y_1), \ldots \ldots \ldots, FIRST(Yi\_1)$; that is $Y_1 \ldots Yi\_1 \Rightarrow \epsilon$. If ϵ is in $FIRST(Yj)$ for all $j = 1,2, \ldots . ., k$ then add ϵ to $FIRST(X)$.
   Everything in $FIRST(Y_1)$ is surely in $FIRST(X)$ If Y_1 does not derive ϵ, then we do nothing more to $FIRST(X)$, but if $Y1 \Rightarrow \epsilon$, then we add $FIRST(Y_2)$ and so on.

Rules to compute First of NT

If $A \rightarrow Y_1 Y_2 \ldots\ldots\ldots Y_K$,

If Y_1 does not derives $\in$ then, $FIRST(A) = FIRST(Y_1)$

If Y_1 derives $\in$ then,

$$FIRST(A) = FIRST(Y_1) - \epsilon \; U \; FIRST(Y_2)$$

If Y_1 & Y_2 derives $\in$ then,

$$FIRST(A) = FIRST(Y_1) - \epsilon \; U \; FIRST(Y_2) - \epsilon \; U \; FIRST(Y_3)$$

If Y1, Y2, Y3 derives $\in$ then,

$$FIRST(A) = FIRST(Y_1) - \epsilon \; U \; FIRST(Y_2) - \epsilon \; U \; FIRST(Y_3) - \epsilon \; U \; FIRST(Y_4)$$

If Y1, Y2, Y3.....Y_K all derives $\in$ then,

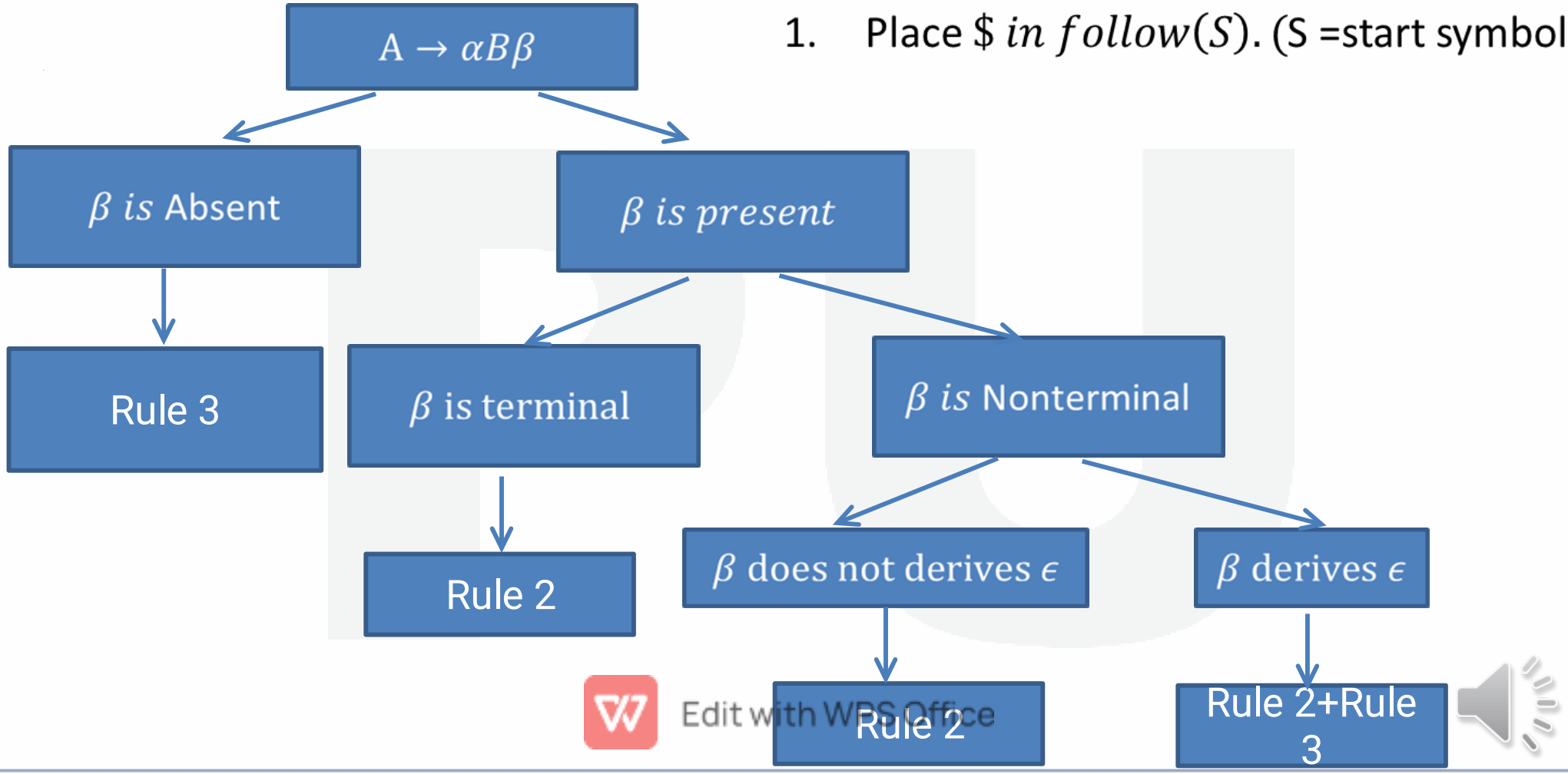
$$FIRST(A) = FIRST(Y_1) - \epsilon \; U \; FIRST(Y_2) - \epsilon \; U \; FIRST(Y_3) - \epsilon \; U \; FIRST(Y_4) - \epsilon \; U \ldots\ldots\ldots\ldots FIRST(Y_k)$$

Rules to compute FOLLOW of NT

1. Place \$ $in\ follow(S)$. (S =start symbol)
2. If $A \rightarrow \alpha B \beta$, then everything in $FIRST(\beta)$ except for ϵ is placed in $FOLLOW(B)$
3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where $FIRST(\beta)$ contains ϵ then everything in $FOLLOW(A) = FOLLOW(B)$

Rules to compute FOLLOW of NT

1. Place \$ $in\ follow(S)$. (S =start symbol)



Rules to construct Predictive Parsing Table

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in $first(\alpha)$, Add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in $first(\alpha)$, Add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $FOLLOW(B)$. If ϵ is in $first(\alpha)$, and \$ is in $FOLLOW(A)$, Add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

Example: LL(1) Parsing

E→E+T | T
T→T*F | F
F→(E) | id

Step1: Remove left recursion

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

Example: LL(1) Parsing

Step2: Compute FIRST

First(E)

E→TE'

E	→	T	E'
A	→	Y_1	Y_2

Rule 3
First(A)=First(Y1)

FIRST(E)=FIRST(T)

First(T)

T→FT'

T	→	F	T'
A	→	Y_1	Y_2

Rule 3
First(A)=First(Y1)

FIRST(T)=FIRST(F)

First(F)

F→(E)

F→id

F	→	(	E	)
A	→	α		

F	→	id
A	→	α

FIRST(F)={ (, id }

Edit with WPS Office

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

NT	First
E	
E'	
T	
T'	
F	

Rule 1
add α to $FIRST(A)$

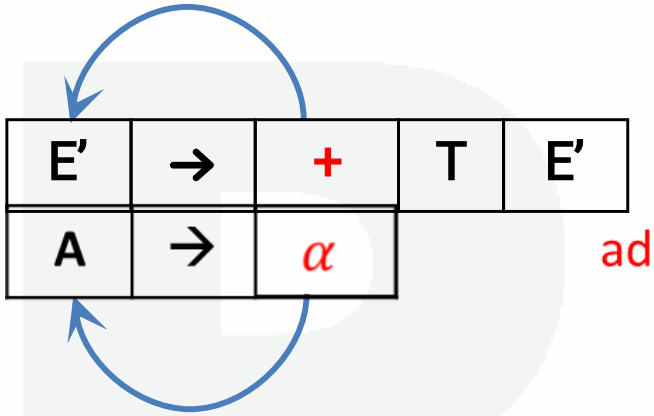
Example: LL(1) Parsing

Step2: Compute FIRST

First(E')

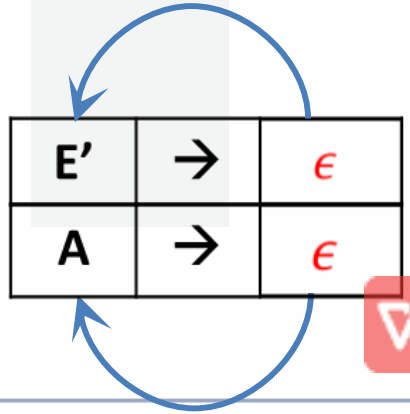
E'→+TE'

E'→ϵ



Rule 1

add α to $FIRST(A)$



Rule 2

add ϵ to $FIRST(A)$

FIRST(E')={+, ϵ}

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

NT	First
E	{ (,id }
E'	
T	{ (,id }
T'	{*, ϵ}
F	{ (,id }

Example: LL(1) Parsing

Step2: Compute FOLLOW

FOLLOW(E)

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

F→(E)

Rule 1: Place $ in FOLLOW(E)

F	→	(	E	)
A	→	α	B	β

Rule 2

FOLLOW(E)={) }

$

NT	First	Follow
E	{ (,id }	
E'	{ +, ϵ }	
T	{ (,id }	
T'	{ *, ϵ }	
F	{ (,id }	

Example: LL(1) Parsing

Step2: Compute FOLLOW

FOLLOW(E')

E→TE'

E	→	T	E'
A	→	α	B

Rule 3

E'→+TE'

E'	→	+T	E'
A	→	α	B

Rule 3

FOLLOW(E')={ $,) }

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

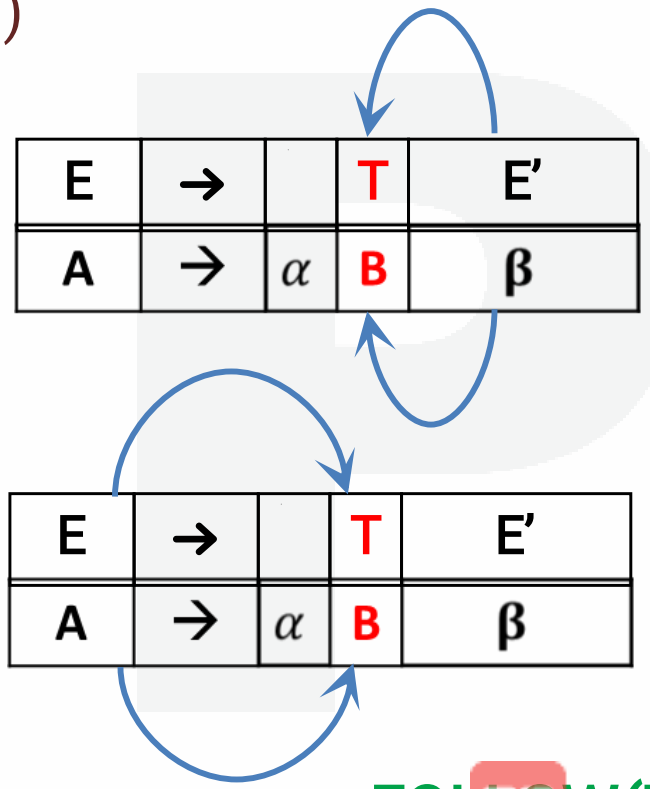
NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	
T	{ (,id }	
T'	{ *, ϵ }	
F	{ (,id }	

Example: LL(1) Parsing

Step2: Compute FOLLOW

FOLLOW(T)

E→TE’



E→TE’

E’→+TE’ | ϵ

T→FT’

T’→*FT’ | ϵ

F→(E) | id

NT	First	Follow
E	{ (,id }	{ $,) }
E’	{ +, ϵ }	{ $,) }
T	{ (,id }	
T’	{ *, ϵ }	
F	{ (,id }	

FOLLOW(T)={ +,

Example: LL(1) Parsing

Step2: Compute FOLLOW

FOLLOW(T)

E'→+TE'

E'	→	+	T	E'
A	→	α	B	β

Rule 2

E'	→	+	T	E'
A	→	α	B	β

Rule 3

FOLLOW(T)={ +$,) }

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	
T'	{ *, ϵ }	
F	{ (,id }	

Example: LL(1) Parsing

Step2: Compute FOLLOW

FOLLOW(T')

T→FT'

T	→	F	T'
A	→	α	B

Rule 3

T'→*FT'

T'	→	*F	T'
A	→	α	B

Rule 3

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

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	
F	{ (,id }	

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

Example: LL(1) Parsing

Step 2: Compute FOLLOW

FOLLOW(F)

T→FT'

T	→		F	T'
A	→	α	B	β

Rule 2

T	→		F	T'
A	→	α	B	β

Rule 3

E→TE'

E'→+TE' | ϵ

T→FT'

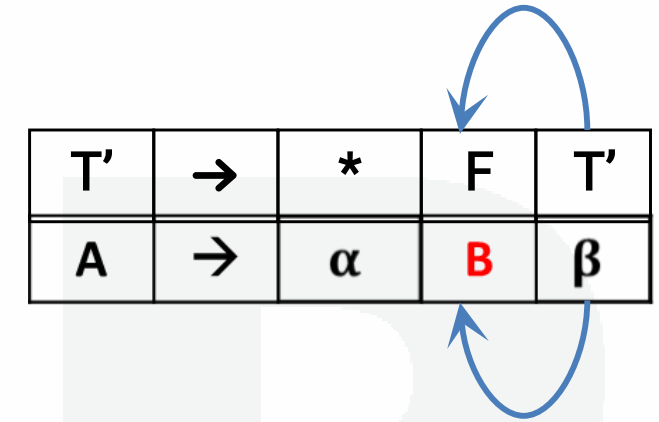
T'→*FT' | ϵ

F→(E) | id

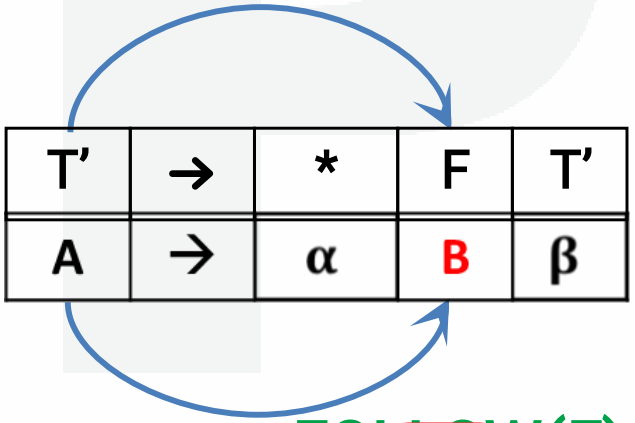
NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	

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

Example: LL(1) Parsing



Rule 2



Rule 3

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

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

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	

Example: LL(1) Parsing

Step3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE' a=FIRST(TE')={ (,id }

M[E,(]=E→TE'

M[E,id]=E→TE'

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) Parsing

Step3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'				
T						
T'						
F						

E'→+TE'

a=FIRST(+TE')={ + }

M[E',+]=E'→+TE'

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) Parsing

Step3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T						
T'						
F						

E'→ϵ

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

M[E',$]=E'→$\epsilon$

M[E',)]=E'→ϵ

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

Rule: 3

A→ α

b = follow(A)

M[A,b] = A→α

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) Parsing

Step3: Construct predictive parsing table

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

T→FT'

a=FIRST(FT')={ (,id }

M[T,(]=T→FT'

M[T,id]=T→FT'

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Rule: 2

A→ α

a = first(α)

M[A,a]=A→ α

Example: LL(1) Parsing

Step3: Construct predictive parsing table

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

T'→*FT'

a=FIRST(*FT')={ * }

M[T',*]=T'→*FT'

Rule: 2

A→ α

a = first(α)

M[A,a] = A→α

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Example: LL(1) Parsing

Step3: Construct predictive parsing table

NT	Input Symbol					
	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					Rule: 3 A→ α b = follow(A) M[A,b] = A→ α	

T'→ϵ
b=FOLLOW(T')={ +,$,) }
M[T',+]=T'→ϵ
M[T',$]=T'→$\epsilon$
M[T',)]=T'→ϵ

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

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

Example: LL(1) Parsing

Step3: Construct predictive parsing table

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

a=FIRST((E))={ (}

M[F,(]=F→(E)

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

F→(E) | id

Rule: 2

A→ α

a = first(α)

M[A,a] = A→ α

Example: LL(1) Parsing

Step3: Construct predictive parsing table

NT	Input Symbol					
	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)		

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

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

a=FIRST(id)={ id }

M[F,id]=F→id

Rule: 2
A→ α
a = first(α)
M[A,a] = A→ α

Example: LL(1) Parsing

Step3: Construct predictive parsing table

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'	Error	Error	E→TE'	Error	Error
E'	Error	E'→+TE'	Error	Error	E'→ϵ	E'→ϵ
T	T→FT'	Error	Error	T→FT'	Error	Error
T'	Error	T'→ϵ	T→*FT'	Error	T'→ϵ	T'→ϵ
F	F→id	Error	Error	F→(E)	Error	Error

Rule: 4 Make each undefined entry of M table be ERROR

NT	First	Follow
E	{ (,id }	{ $,) }
E'	{ +, ϵ }	{ $,) }
T	{ (,id }	{ +,$,) }
T'	{ *, ϵ }	{ +,$,) }
F	{ (,id }	{*,+,$,)}

E→TE'

E'→+TE' | ϵ

T→FT'

T'→*FT' | ϵ

Example: LL(1) Parsing

Step4: Parse the string : id + id * id $

STACK	INPUT	OUTPUT
$E	id+id*id$	
		T'→ϵ

NT	Input Symbol					
	id	+	*	(	)	$
E	E→TE'	Error	Error	E→TE'	Error	Error
E'	Error	E'→+TE'	Error	Error	E'→ϵ	E'→ϵ
T	T→FT'	Error	Error	T→FT'	Error	Error
T'	Error	T'→ϵ	T→*FT'	Error	T'→ϵ	T'→ϵ
F	F→id	Error	Error	F→(E)	Error	Error

Exercise

S → AaAb | BbBa A→ϵ B→ϵ	S→ aAB | bA | ϵ A→ aAb | ϵ B→ bB | ϵ	S→iCtSA | a A→ eS | ϵ C→ b
S→ (L) | a L→ L,S | S	E→ TA A→ +TA | ϵ T→ VB B→ *VB |ϵ V→ id | (E)	S→ a | ^ | (R) T→ S, T | S R→ T

Recursive Descent Parsing

- A top down parsing that executes a set of recursive procedure to process the input without backtracking is called recursive descent parser.
- There is a procedure for each non terminal in the grammar.
- Consider RHS of any production rule as definition of the procedure.
- As it reads expected input symbol, it advances input pointer to next position.

Recursive Descent Parsing

```
Procedure E
{
   If lookahead=num
   {
      Match(num);
      T();
   }
   Else
      Error();
   If lookahead=$
   {
      Declare success;
   }
   Else
      Error();
}
```

```
Procedure T
{
   If lookahead='*'
   {
      Match('*');
      If lookahead=num
      {
         Match(num);
         T();
      }
      Else
         Error();

   }
   Else
      NULL
}
```

```
Proceduce Match(token t)
{
   If lookahead=t

Lookahead=next_token;
   Else
      Error();
}
```

```
Procedure Error
{
   Print("Error");
}
```


Example:
E→ numT

3	*	4	$

Parul®
University

DIGITAL LEARNING CONTENT

Bottom Up Parsing

Naïve bottom up parsing Algorithm

1. SSM := 1; n := 0
2. r := n
3. Compare the string of r symbols to the left of SSM with all RHS alternatives in G which have length of r symbols.
4. If a match is found with a production A ::=α, then;

reduce the string of r symbols to NT A;
n := n – r + 1; Goto step 2

5. r:=r-1;

if (r > 0), then goto step 3;

6. If no more symbols exist to the right of SSM then

If current string form = 'S' then
exit with success;
else report error and exit with failure;

7. SSM := SSM + 1;

n := n + 1; goto step 2;

Operator Precedence Parser

- **Operator Grammar**: A Grammar in which there is no Є in RHS of any production or no adjacent non terminals is called operator grammar.

Relation	Meaning
a<·b	a “yields precedence to” b
a=·b	a “has the same precedence as” b
a·>b	a “takes precedence over” b

Steps of Operator Precedence Parsing

1. Find Leading and trailing of NT
2. Establish relation
3. Creation of table
4. Parse the string

Leading & Trailing

Leading:- Leading of a nonterminal is the first terminal or operator in production of that nonterminal.

Trailing:- Trailing of a nonterminal is the last terminal or operator in production of that nonterminal.

Example: E→E+T | T

T→T*F | F

F→id

Nonterminal	Leading	Trailing
E		
T		
F		

Rules to Establish a Relation

1. For a = b, aAb, where A is ϵ or a single NT
2. a <· b $\Rightarrow Op\ .\ NT\ \ then\ \ Op\ <.\ Leading(NT)$
3. a ·> b $\Rightarrow NT\ .\ Op\ then\ (Trailing(NT))\ \cdot > Op$
4. $ <· leading(start symbol)
5. Trailing(start symbol) ·> $

Example: Operator precedence parsing

Step1: Find Leading & Trailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

E→ E+T | T
T→ T *F| F
F→ id

Step2: Establish Relation

a <· b

$Op \cdot NT$	$Op <\cdot Leading(NT)$
$+T$	$+ <\cdot \{*, id\}$
$* F$	$* <\cdot \{id\}$

Step3: Creation of Table

	+	*	id	$
+				
*	⋗	⋗	⋗	⋗
id	⋗	⋗		⋗
$	<·	<·	<·	

Example: Operator precedence parsing

Step1: Find Leading & Tailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

E→ E + T| T
T→ T * F| F
F→ id

Step2: Establish Relation

a ·>b

$NT \cdot Op$	$(Trailing(NT)) \cdot > Op$
$E +$	$\{+,*,id\} \cdot > +$
$T *$	$\{*,id\} \cdot > *$

Step3: Creation of Table

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

Example: Operator precedence parsing

E→ E+T| T
T→ T* F| F
F→ id

Step1: Find Leading & Tailing of NT

Nonterminal	Leading	Trailing
E	{+,*,id}	{+,*,id}
T	{*,id}	{*,id}
F	{id}	{id}

Step2: Establish Relation

$<· leading(start symbol)

$\$ <\cdot \{+,*,id\}$

Trailing(start symbol) ·> $

$\{+,*,id\} \cdot > \$$

Step3: Creation of Table

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

Example: Operator precedence parsing

Step4: Parse the string using Precedence Table

1.Scan the input string until first ·> is encountered.
2.Scan backward until <· is encountered.
3.The handle is string between <· and ·>

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

String: id+id*id

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

Example: Operator precedence parsing

$ <· Id ·> + <· Id ·> * <· Id ·> $	
$ F + <· Id ·> * <· Id ·> $	
$ F + F * <· Id ·> $	
$ F + F * F $	
$ E + T * F $	
$ + * $	
$ <· + <· * >$	
$ <· + >$	
$ $	

Operator Precedence Parsing Algorithm

Algorithm

1.TOS:= SB-1; SSM=0;

2.Push ' ⊢' on the stack;

3.ssm=ssm+1;

If current source symbol is an operator, goto step 5;

4.x:=newnode(source symbol, null, null)

TOS.operand_pointer:=x; Go to step 3;

5.while TOS operator .> current operator

x:=newnode(TOS operator, TOSM.operand_pointer,TOS.operand_pointer);

pop an entry of the stack;

TOS.operand_pointer:=x;

	+	*	id	-|
+	·>	<·	<·	·>
*	·>	·>	<·	·>
id	·>	·>		·>
⊢	<·	<·	<·	=

Operator Precedence Parsing Algorithm

6.If TOS operator <. current operator, then

Push the current operator on the stack. Go to step 3;

7.while TOS operator .= current operator, then

if TOS operator = ├ then exit successfully

if TOS operator =’(‘, then

temp:=TOS.operand_pointer;

pop an entry off the stack

TOS.operand_pointer:=temp; Go to step 3;

8.if no precedence define between TOS operator and current operator the report error and exit unsuccessfully

Language Processor Development Tools

- The two widely used language processor development tools are the lexical analyzer generator LEX and the parser generator YACC.
- The input to these tools are specifications of the lexical and syntactic constructs of a programming language L, and the semantic actions that should be performed on recognizing the constructs

LEX

- The input to LEX consists of two components.
- The first component is a specification of strings that represents the lexical units in L. This specification is in the form of regular expressions.
- The second component is a specification of semantic actions that are aimed at building the intermediate representation.
- The intermediated representation produced by a scanner would consist of a set of tables of lexical units and a sequence of tokens for the lexical units occurring in a source statement.
- The scanner generated by LEX would be invoked by a parser whenever the parser needs the next token.
- Accordingly, each semantic action would perform some table building actions and return a single token.

YACC

- Each translation rule input to YACC has a string specification that resembles a production of a grammar: It has a nonterminal on the LHS and a few alternatives on the RHS.
- For simplicity, we will refer to a string specification as a production. YACC generates an LALR(1) parser for language L from the productions, which is a bottom-up parser.
- The parser would operate as follows:

1. For a shift action, it would invoke the scanner to obtain the next token and continue the parse by using that token.
2. While performing a reduce action in accordance with a production, it would perform the semantic action associated with that production.

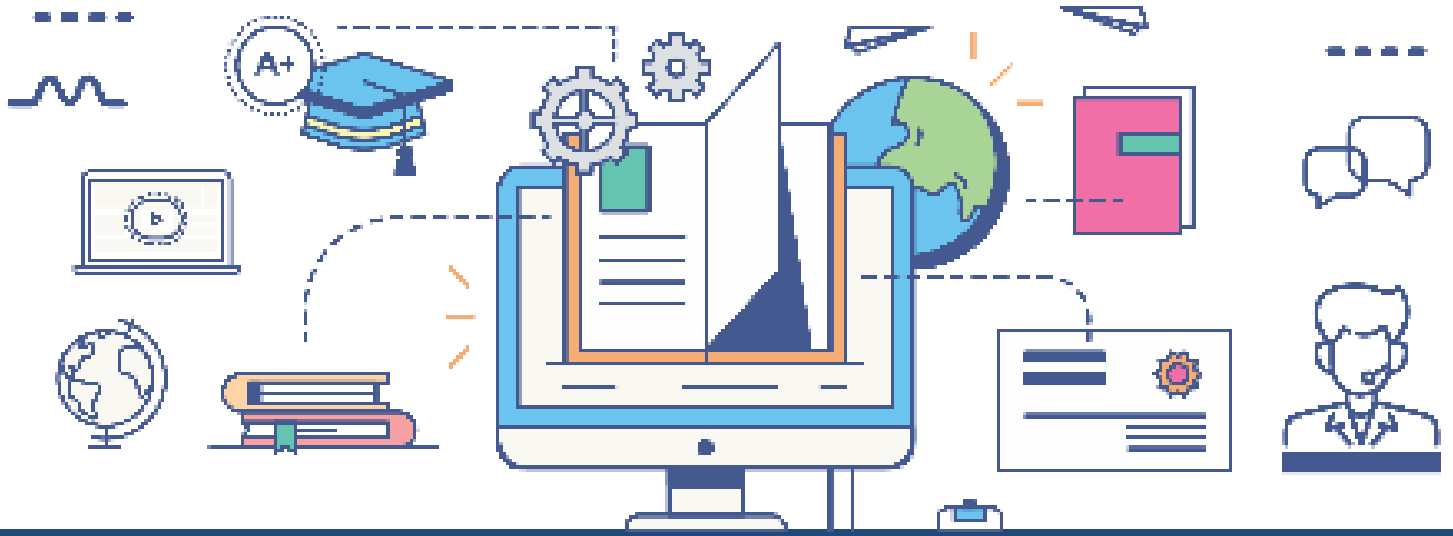
YACC

- The semantic actions associated with productions achieve building of an intermediate representation or target code as follows:
- Every nonterminal symbol in the parser has an attribute. The semantic action associated with a production can access attributes of nonterminal symbols used in that production.
- A symbol '$n' in the semantic action, where n is an integer, designates the attribute of the n^{th} nonterminal symbol in the RHS of the production
- The symbol '$$' designates the attribute of the LHS nonterminal symbol of the production.
- The semantic action uses the values of these attributes for building the intermediate representation or target code.
- The attribute type can be declared in the specification input to YACC.

References

1. System Programming by D M Dhamdhere McGraw Hill Publication
2. System Programming by Srimanta Pal OXFORD Publication
3. System Programming and Compiler Construction by R.K. Maurya& A. Godbole.
4. System Software – An Introduction to Systems Programming by Leland L. Beck, 3rd Edition, Pearson Education Asia, 2000
5. System Software by SantanuChattopadhyay, Prentice-Hall India,2007

DIGITAL LEARNING CONTENT



Parul® University

www.paruluniversity.ac.