UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Compiler Design
Prof. Sankhadeep Chatterjee

- Construction of a parse tree is done by starting the root labeled by a start symbol

- Repeat following two steps

  - At a node labeled with non terminal **A** select one of the productions of **A** and construct children nodes

  - Find the next node at which subtree is Constructed

N = {S, A}

T = {a, b, c, d}

P =

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

S = {S}

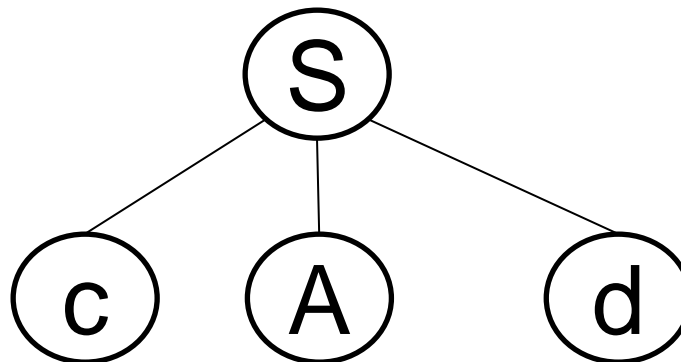Derive string (w) cad

$A \to ab \mid a$
$S \to cAd$

$\boxed{S}$

Derive string cad

Node $\to$ S

Input pointer $\to$ c

Selected Production
Rule: $S \to cAd$

$$A \rightarrow ab \mid a$$
$$S \rightarrow cAd$$

Derive string cad

Node → S

Input pointer → c

Selected Production
Rule: $S \rightarrow cAd$

$$A \to ab \mid a$$
$$S \to cAd$$

Derive string cad

Node → A

Input pointer → a

Selected Production
Rule: $A \to ab$

$A \rightarrow ab \mid a$
$S \rightarrow cAd$

Derive string cad

Node → A

Input pointer → a

Selected Production
Rule: $A \rightarrow ab$

$A \rightarrow ab \mid a$

$S \rightarrow cAd$

Derive string cad

Node → b

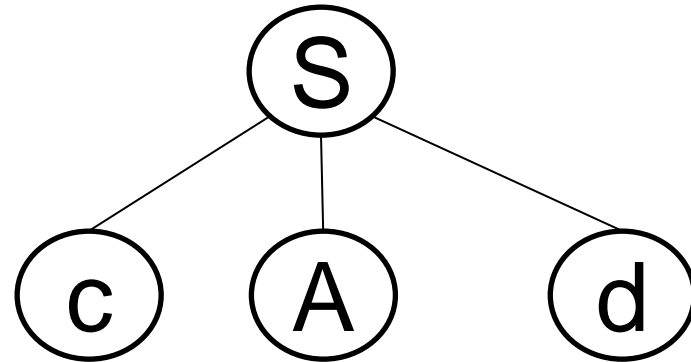Input pointer → d

Selected Production Rule: $Failure!$

Backtrack

$A \rightarrow ab \mid a$
$S \rightarrow cAd$

Derive string cad

Node $\rightarrow$ A

Input pointer $\rightarrow$ a

Selected Production
Rule: $A \rightarrow a$
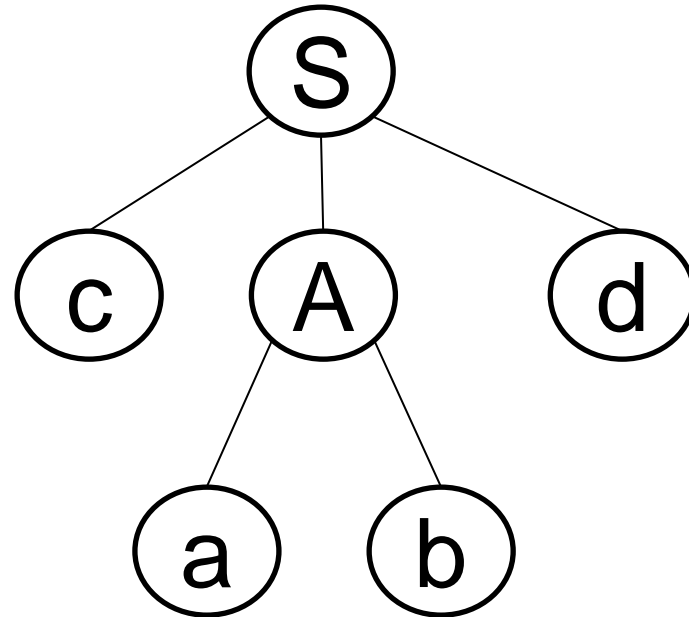
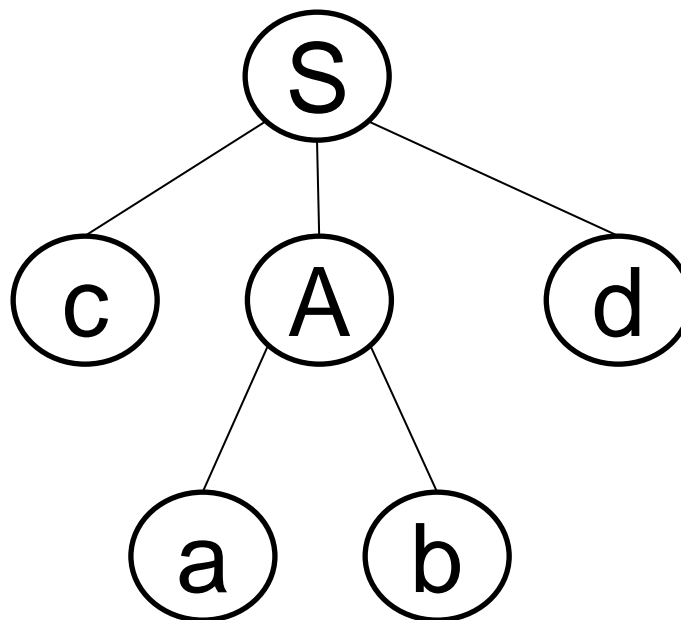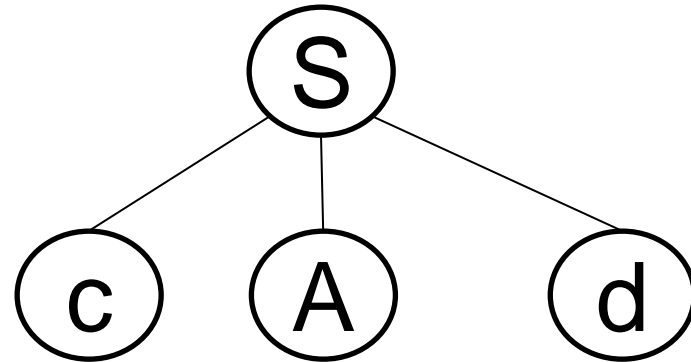$$A \rightarrow ab \mid a$$
$$S \rightarrow cAd$$

Derive string cad

Node → A

Input pointer → a

Selected Production
Rule: $A \rightarrow a$
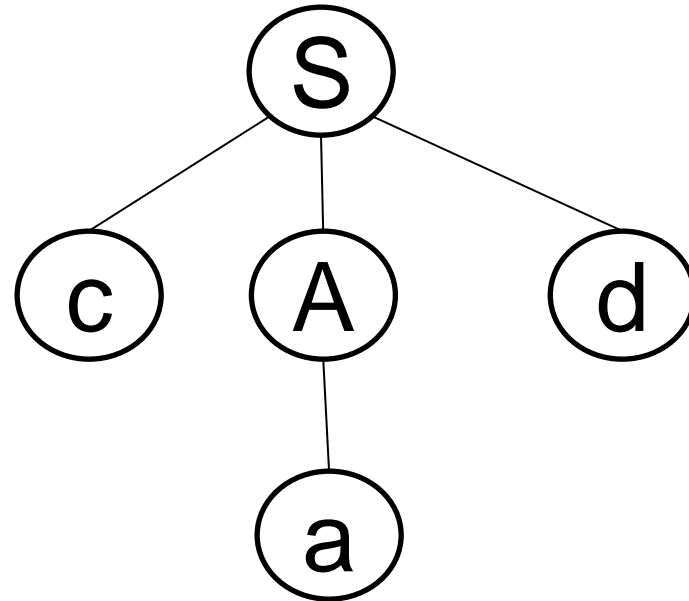
Recursive Descent Parsing

```
        void A() {
1)              Choose an A-production, A → X₁X₂ ⋯ Xₖ;
2)              for ( i = 1 to k ) {
3)                      if ( Xᵢ is a nonterminal )
4)                              call procedure Xᵢ();
5)                      else if ( Xᵢ equals the current input symbol a )
6)                              advance the input to the next symbol;
7)                      else /* an error has occurred */;
                }
        }
```

11

Recursive Descent Parsing

- Recursive Descent parsing is a set of procedures, one for each nonterminal

- Execution begins from Start symbol

- The pseudocode is nondeterministic

- It requires backtracking, which is not very efficient

- Tabular methods (Dynamic Programming) of parsing is used

Left Recursion

- Top down parser with production rule $A \rightarrow A\alpha \mid \beta$ may loop forever.
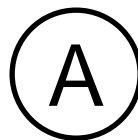
$$\boxed{A}$$

Left Recursion

- Top down parser with production rule $A \rightarrow A\alpha \mid \beta$ may loop forever.
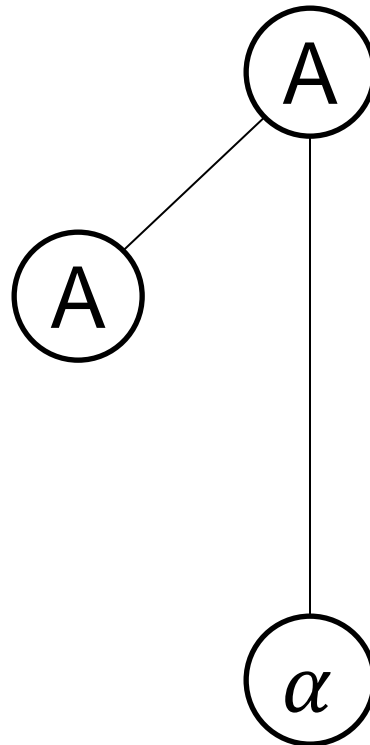
Left Recursion

- Top down parser with production rule $A \rightarrow A\alpha \mid \beta$ may loop forever.
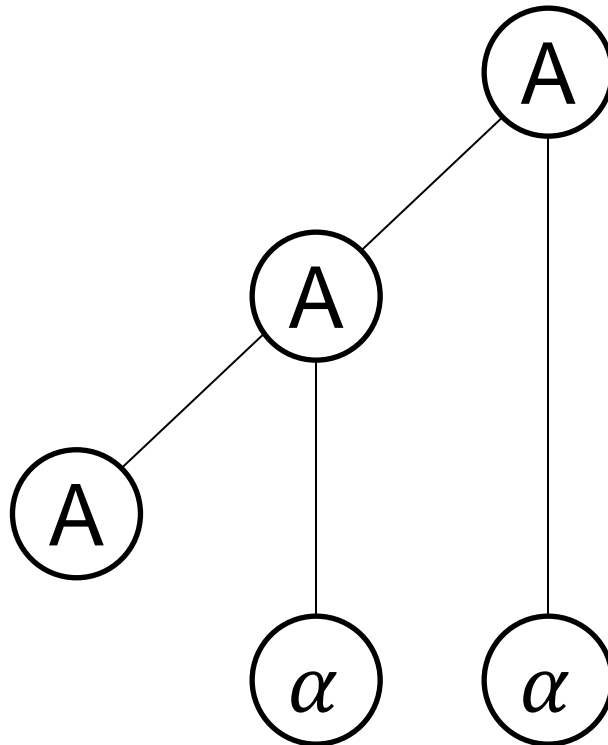
Left Recursion

- Top down parser with production rule $A \rightarrow A\alpha \mid \beta$ may loop forever.
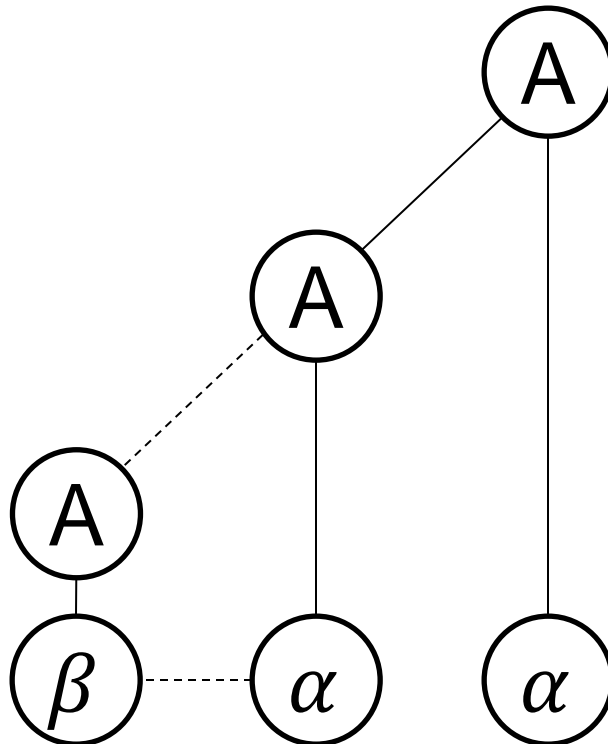
Left Recursion

- Top down parser with production rule $A \rightarrow A\alpha \mid \beta$ may loop forever.

- The left recursion can be removed by rewriting the grammar as follows;

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

17

- Both Parse tree generate $\beta\alpha^*$
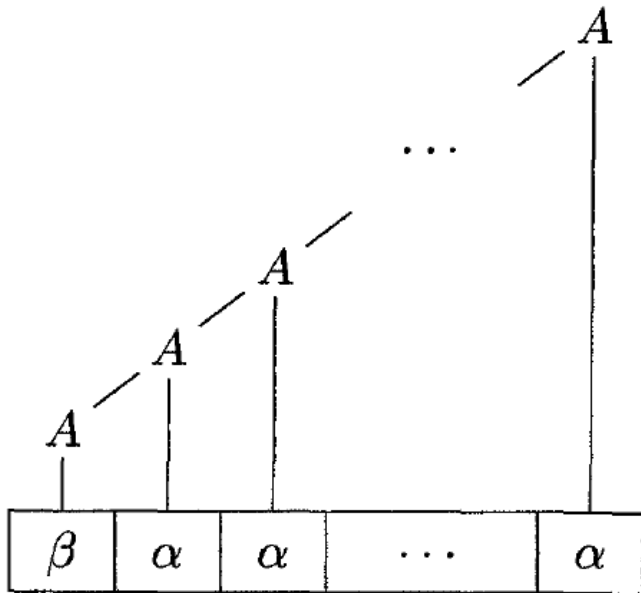
With Left Recursion   Without Left Recursion
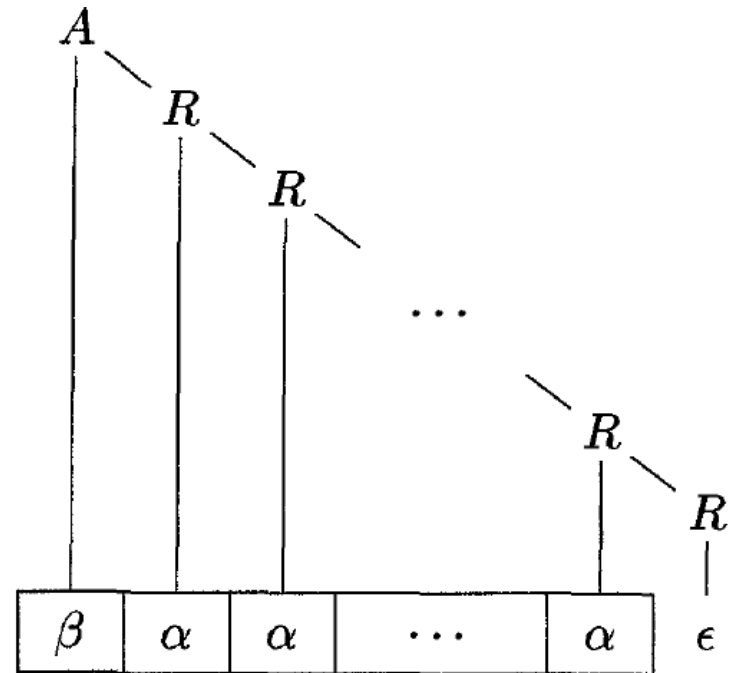
Left Recursion Removal

Remove left recursion from the following grammar:

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

19

Left Recursion Removal

Remove left recursion from the following grammar:

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

$$E \rightarrow TE'$$

Left Recursion Removal

Remove left recursion from the following grammar:

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

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

21

Left Recursion Removal

Remove left recursion from the following grammar:

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

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$

22

Left Recursion Removal

Remove left recursion from the following grammar:

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

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \varepsilon$$

Left Recursion Removal

Remove left recursion from the following grammar:

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

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

Left Recursion Removal

- In general

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

- Transforms to

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \ ... \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \epsilon$$

Left Recursion Removal

- Consider the following Grammar:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- There is left recursion as:

$$S \rightarrow Aa \rightarrow Sda$$

- How can we remove left recursion?
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol
  - Removing left recursion from the modified grammar

26

Left Recursion Removal

- After the first step (substitute S by its body in the rules) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- After the second step (removal of left recursion) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

27

Algorithm for Left Recursion Removal

INPUT: Grammar G with no cycles or ϵ-production

OUTPUT: Equivalent grammar without left recursion

1)   arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2)   **for** (each i from 1 to n) {
3)   **for** (each j from 1 to $i-1$) {
4)   replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \cdots \mid \delta_k \gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5)   }
6)   eliminate the immediate left recursion among the A_i-productions
7)   }

Left factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol
  - Defer the decision till we have seen enough input.

- In general if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
  - Defer decision by expanding A to $\alpha A'$
  - Later A' can be expanded to β_1 or β_2

- Therefore, $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ is transformed to
$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

FIRST & FOLLOW

- The Top down parser is generated with the help of two functions
  - FIRST
  - FOLLOW

- FIRST(α) is set of terminals (tokens) that begin the strings derived from string α

- FOLLOW(A) is set of terminals (tokens) that might follow the derivation of non terminal A

31

FIRST & FOLLOW

FIRST   FOLLOW
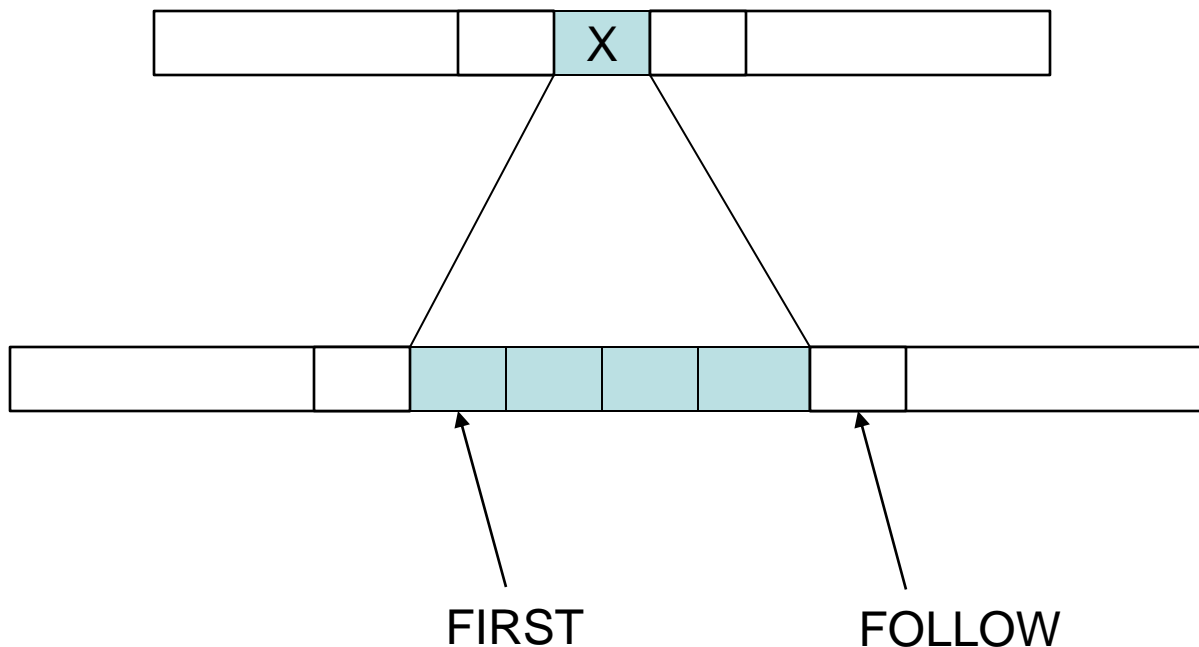
Algorithm for FIRST

- If X is a terminal symbol then First(X) = $\{X\}$

- If $X \to \epsilon$ is a production then ϵ is in First(X)

- If X is a non terminal & $X \to Y_1 Y_2 \dots Y_i Y_j \dots Y_k$ is a production then
  if $a \in$ First(Y_j) and ϵ is in all $Y_i \; \forall \; i < j$ then
  a is in First(X)

- If X is a non terminal and $X \to Y_1 Y_2 \dots Y_k$ is a production then
  If ϵ is in all $Y_i \; \forall \; 1 \leq i \leq k$ then
  ϵ is in First(X)

33

Example: FIRST

- For the following grammar:

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

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

FIRST(E') = { +, ϵ}

FIRST(T') = { *, ϵ}

Algorithm for FOLLOW

- Place $ in FOLLOW(S)

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

- If there is a production $A \rightarrow \alpha B$ then everything in FOLLOW(A) is in FOLLOW(B)

- If there is a production $A \rightarrow \alpha B \beta$ and FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B)

35

Example: FOLLOW

- For the following grammar:

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

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

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

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

Problem Solving

1. Find the FIRST and FOLLOW of every grammar symbol of the following grammar;

$$S \rightarrow aABb$$
$$A \rightarrow c \mid \epsilon$$
$$B \rightarrow d \mid \epsilon$$

2. Find the FIRST and FOLLOW of every grammar symbol of the following grammar;

$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \epsilon$$
$$B \rightarrow \epsilon$$

37

Predictive Parsing

- A non recursive top down parsing method

- Parser "predicts" which production to use

- It removes backtracking by fixing one production for every non-terminal and input token(s)

- Predictive parsers accept LL(k) languages
  - First L stands for left to right scan of input
  - Second L stands for leftmost derivation
  - k stands for number of lookahead token

- In practice LL(1) is used

Predictive Parsing

- Predictive parser can be implemented by maintaining an external stack

39
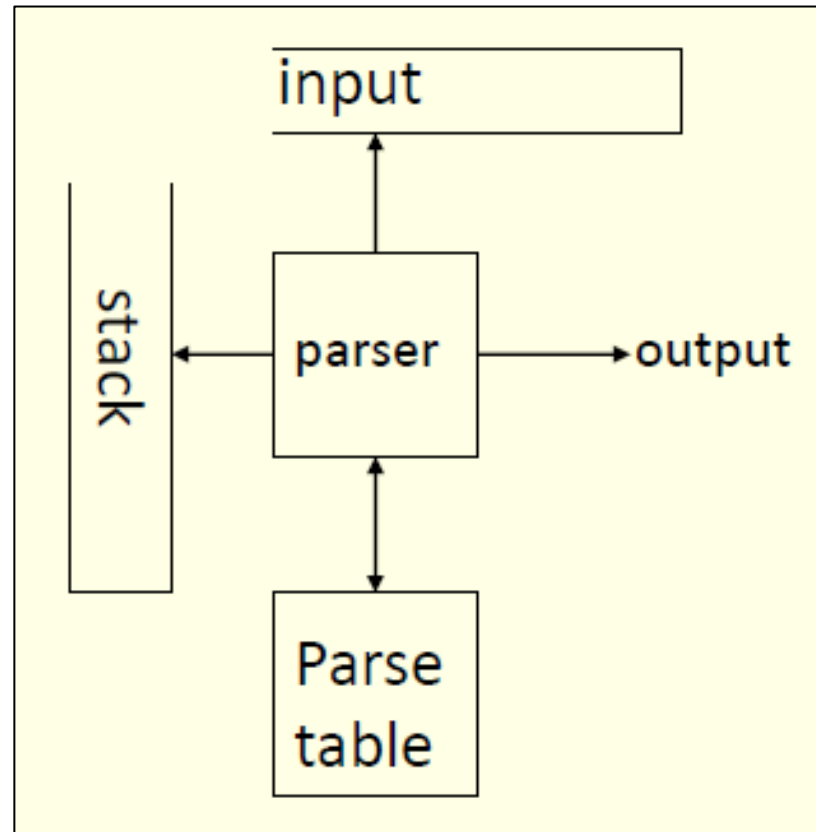
Predictive Parsing Table

- Parse table is a two dimensional array $M[X, a]$ where "X" is a non terminal and "a" is a terminal of the grammar

Terminals

	a_1	a_2	$\ldots$	a_m
X_1	$\ldots$	$\ldots$	$\ldots$	$\ldots$
X_2	$\ldots$	$X_2 \rightarrow \alpha$	$\ldots$	$\ldots$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
X_n	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Non terminals

40

Algorithm: Predictive Parsing Table

- for each production $A \rightarrow \alpha$ do

  for each terminal 'a' in FIRST(α)

  M[A,a] = $A \rightarrow \alpha$

- If ϵ is in FIRST(α)

  for each terminal b in FOLLOW(A)

  M[A,b] = $A \rightarrow \alpha$

  If $ is in FOLLOW(A)

  M[A,$] = $A \rightarrow \alpha$

Example: Predictive Parsing Table Generation

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

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

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

NON - TERMINAL	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

- The stack is initialized to contain $S, the $ is the "bottom" marker.

- The input has a $ added to the end.

- The parse table, M[X, a] contains what should be done when we see nonterminal X on the stack and current token "a"

- Parse Actions for
  - X = top of stack, and
  - a = current token

- If $X = a = \$$ then halt and announce success.

- If $X = a \neq \$$ then pop X off the stack and advance the input pointer to the next token.

- If X is nonterminal consult the table entry $M[X, a]$

Predictive Parsing

- If X is nonterminal then consult $M[X, a]$.
  - The entry will be either a production or an error entry.
  - If $M[X, a] = \{X \rightarrow UVW\}$
    - The parser replaces X on the top of the stack with W, V, U with the U on the top
    - As output print the name of the production used.

44

Predictive Parsing Algorithm

Set ip to the first token in $w\$$.

Repeat

    Let X be the top of the stack and a be the current token

    if X is a terminal or $\$$ then

        if $X = a$ then

            pop X from the stack and advance the ip

        else

            error()

    else /* X is a nonterminal */

        if $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ then

            Pop X from the stack

            Push $Y_1 Y_2 \dots Y_k$ onto the stack with Y_1 on top

            output the production $X \rightarrow Y_1 Y_2 \dots Y_k$

        else

            error()

Until $X = \$$

45

LL(1) Grammers

- A grammar is called LL(1) if its parsing table has no multiply defined entries.

- LL(1) grammars
  - Must not be ambiguous.
  - Must not be left-recursive.

- G is LL(1) if and only if whenever $A \rightarrow \alpha | \beta$
  - $FIRST(\alpha) \cap FIRST(\beta) = \phi$
  - At most one of α and β can derive ϵ
  - If $\beta \rightarrow \epsilon$ then $FIRST(\alpha) \cap FOLLOW(A) = \phi$

46

Example: LL(1)

$$S \rightarrow AS'$$
$$S' \rightarrow AS' \mid \epsilon$$
$$A \rightarrow a$$

	a	$\$$
S	$S \rightarrow AS'$	
S'	$S \rightarrow AS'$	$S' \rightarrow \epsilon$
A	$A \rightarrow a$	

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S $	aa $	$S \rightarrow AS'$

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S \$	aa \$	$S \rightarrow AS'$
	AS' \$	aa \$	$A \rightarrow a$

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S \$	aa \$	$S \rightarrow AS'$
	AS' \$	aa \$	$A \rightarrow a$
	aS' \$	aa \$	Matched

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S \$	aa \$	$S \rightarrow AS'$
	AS' \$	aa \$	$A \rightarrow a$
	aS' \$	aa \$	Matched
a	S' \$	a \$	$S \rightarrow AS'$

51

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S $	aa $	$S \rightarrow AS'$
	AS' $	aa $	$A \rightarrow a$
	aS' $	aa $	Matched
a	S' $	a $	$S \rightarrow AS'$
a	AS' $	a $	$A \rightarrow a$

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S $	aa $	$S \rightarrow AS'$
	AS' $	aa $	$A \rightarrow a$
	aS' $	aa $	Matched
a	S' $	a $	$S \rightarrow AS'$
a	AS' $	a $	$A \rightarrow a$
a	aS' $	a $	Matched

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S $	aa $	$S \rightarrow AS'$
	AS' $	aa $	$A \rightarrow a$
	aS' $	aa $	Matched
a	S' $	a $	$S \rightarrow AS'$
a	AS' $	a $	$A \rightarrow a$
a	aS' $	a $	Matched
aa	S' $	$	$S' \rightarrow \epsilon$

Example: Predictive Parsing

MATCHED	STACK	INPUT	ACTION
	S $	aa $	$S \rightarrow AS'$
	AS' $	aa $	$A \rightarrow a$
	aS' $	aa $	Matched
a	S' $	a $	$S \rightarrow AS'$
a	AS' $	a $	$A \rightarrow a$
a	aS' $	a $	Matched
aa	S' $	$	$S' \rightarrow \epsilon$
aa	$	$	

Bottom up Parsing

- Use explicit stack to perform a parse
- Simulate rightmost derivation (R) from left (L) to right, thus called LR parsing
- More powerful than top-down parsing
  - Left recursion does not cause problem
- Two actions
  - Shift: take next input token into the stack
  - Reduce: replace a string B on top of stack by a nonterminal A, given a production $A \rightarrow B$

Bottom up Parsing

57

Bottom up Parsing

- Informally, a "handle" of a string is a substring that matches the right side of the production

- Reduction of "handle" to non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T

62

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F

65

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T

66

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)

68

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F

69

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F
$ T	* id $	Shift

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce by F → id

72

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce by F → id
$ T * F	$	Reduce by T → T * F

73

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce by F → id
$ T * F	$	Reduce by T → T * F
$ T	$	Reduce by E → T

74

Shift-Reduce Parsing

STACK	INPUT	ACTION
$	(id + id) * id $	Shift
$ (	id + id) * id $	Shift
$ (id	+ id) * id $	Reduce by F → id
$ (F	+ id) * id $	Reduce by T → F
$ (T	+ id) * id $	Reduce by E → T
$ (E +	id) * id $	Shift
$ (E + id	) * id $	Reduce by F → id
$ (E + F	) * id $	Reduce by T → F
$ (E + T	) * id $	Reduce by E → E + T
$ (E	) * id $	Shift
$ (E)	* id $	Reduce by F → (E)
$ F	* id $	Reduce by T → F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce by F → id
$ T * F	$	Reduce by T → T * F
$ T	$	Reduce by E → T
$ E	$	accept

75