UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Compiler Design

- 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

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

S

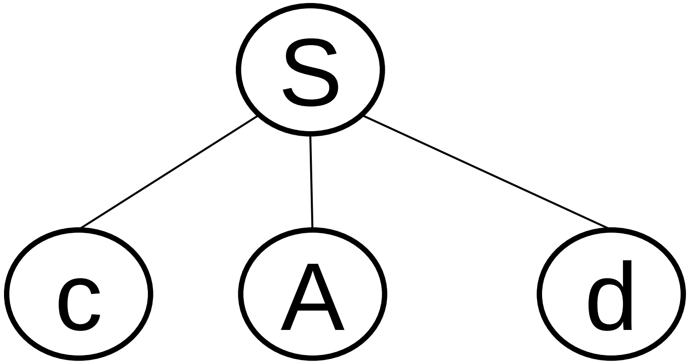
Derive string cad

Node → S

Input pointer → c

Selected Production Rule: $S \rightarrow cAd$

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



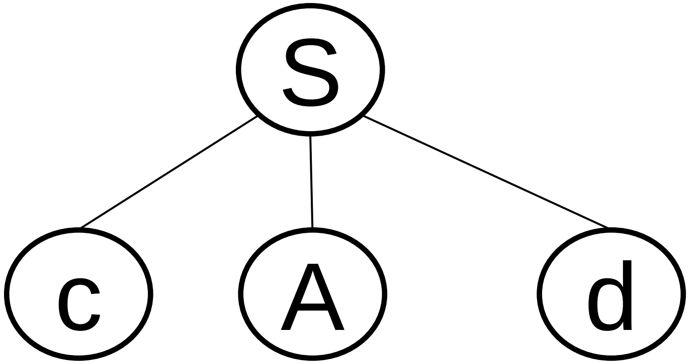
Derive string cad

Node → S

Input pointer → c

Selected Production Rule: $S \rightarrow cAd$

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



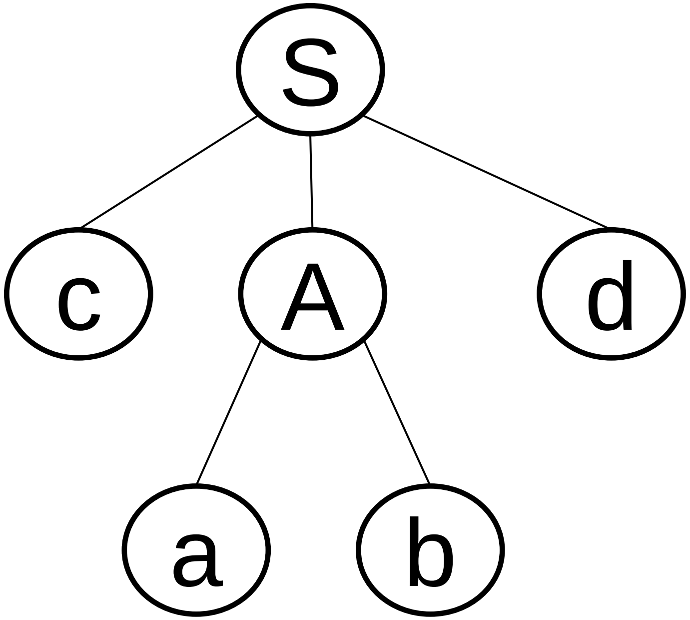
Derive string cad

Node → A

Input pointer → a

Selected Production Rule: $A \rightarrow ab$

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



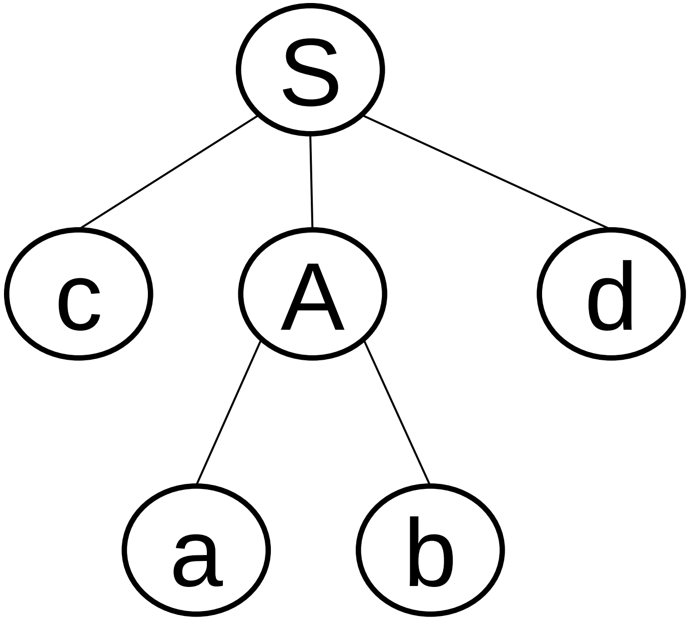
Derive string cad

Node → A

Input pointer → a

Selected Production Rule: $A \rightarrow ab$

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



Derive string cad

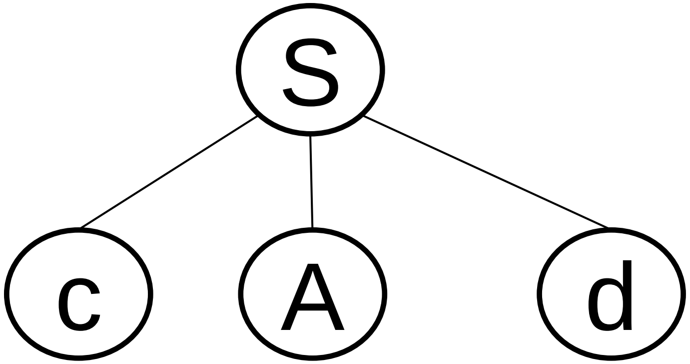
Node → b

Input pointer → d

Selected Production Rule: $Failure$!

Backtrack

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



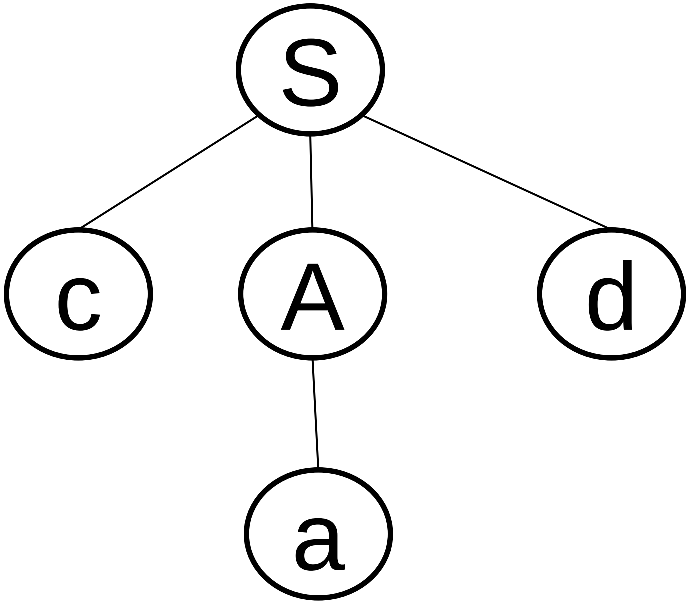
Derive string cad

Node → A

Input pointer → a

Selected Production Rule: $A \rightarrow a$

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



Derive string cad

Node → A

Input pointer → a

Selected Production Rule: $A \to a$

Recursive Descent Parsing

void $A()$ {
    Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

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

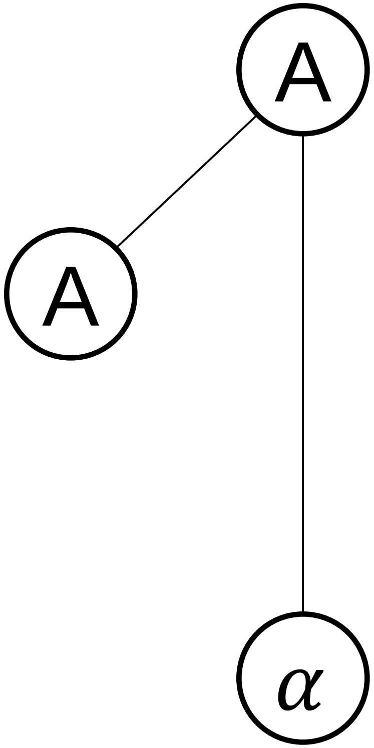
Left Recursion

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

A

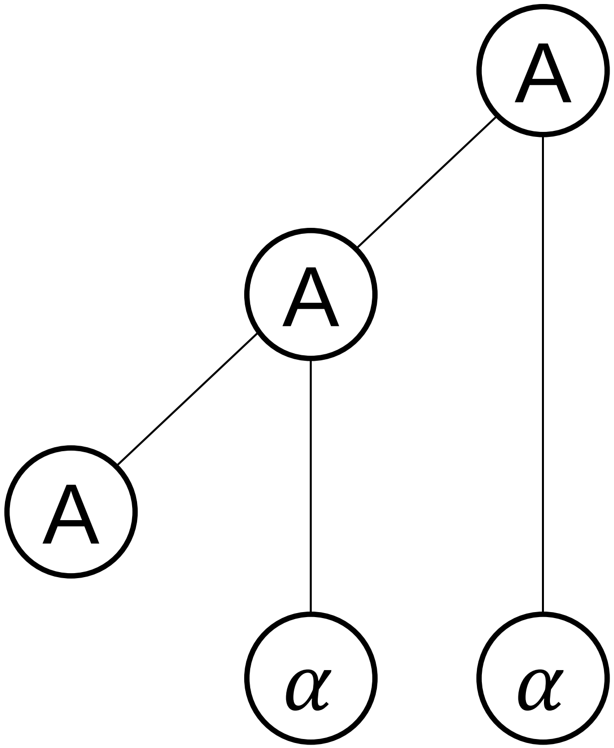
Left Recursion

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



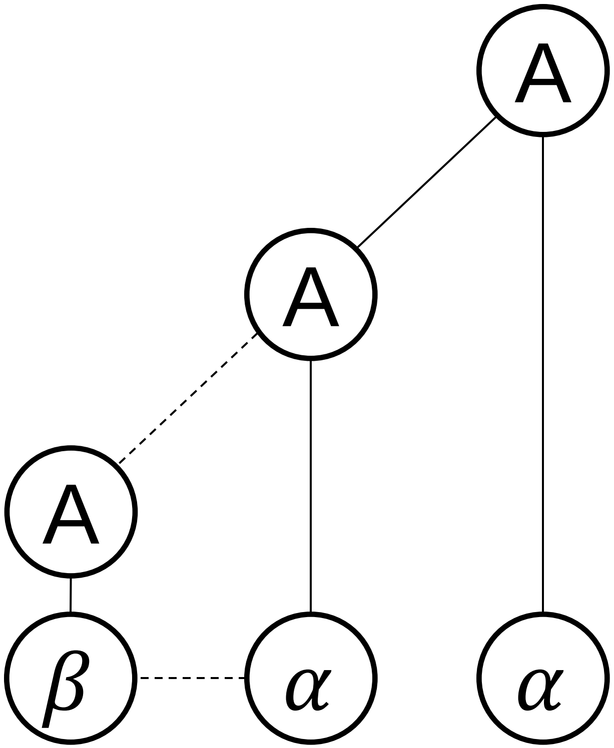
Left Recursion

- 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.



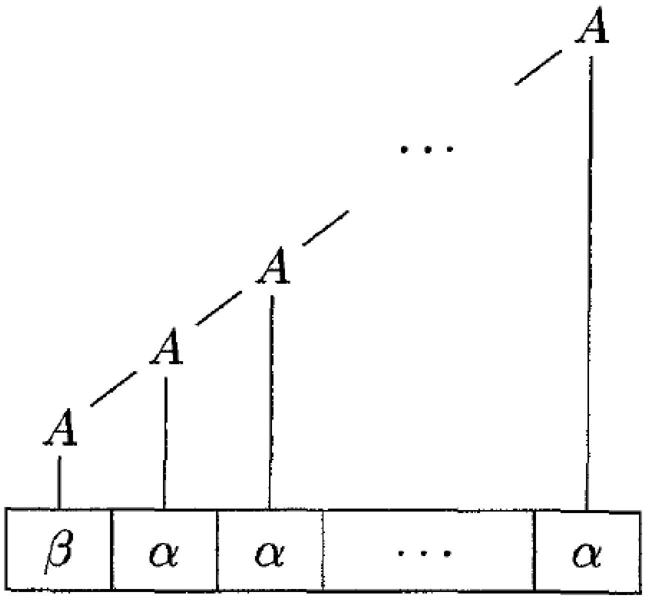
Left Recursion

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

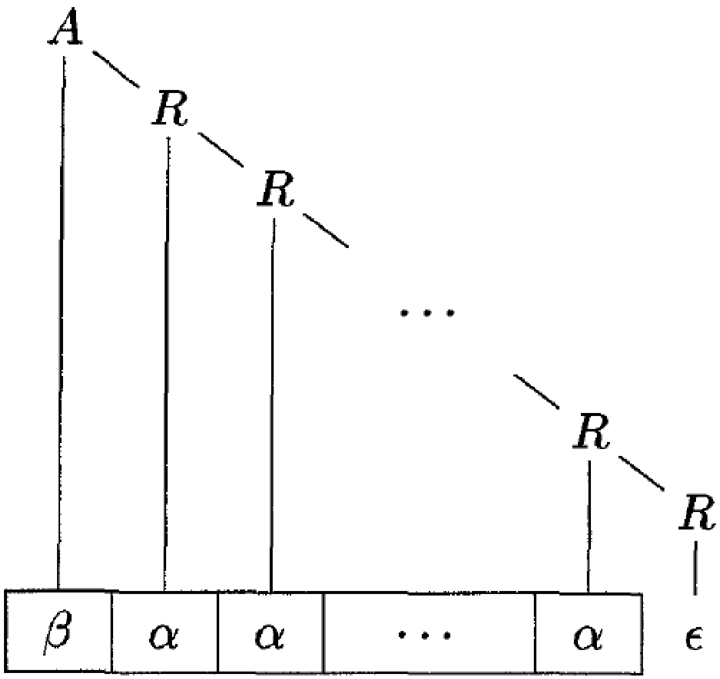
- The left recursion can be removed by rewriting the grammar as follows;

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

- Both Parse tree generate $\beta\alpha^{*}$



With Left Recursion



Without Left Recursion

Left Recursion Removal

Remove left recursion from the following grammar:

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

Left Recursion Removal

Remove left recursion from the following grammar:

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

$$E \to TE'$$

Left Recursion Removal

Remove left recursion from the following grammar:

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

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

Left Recursion Removal

Remove left recursion from the following grammar:

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

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

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

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

- Transforms to

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

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

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

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

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

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

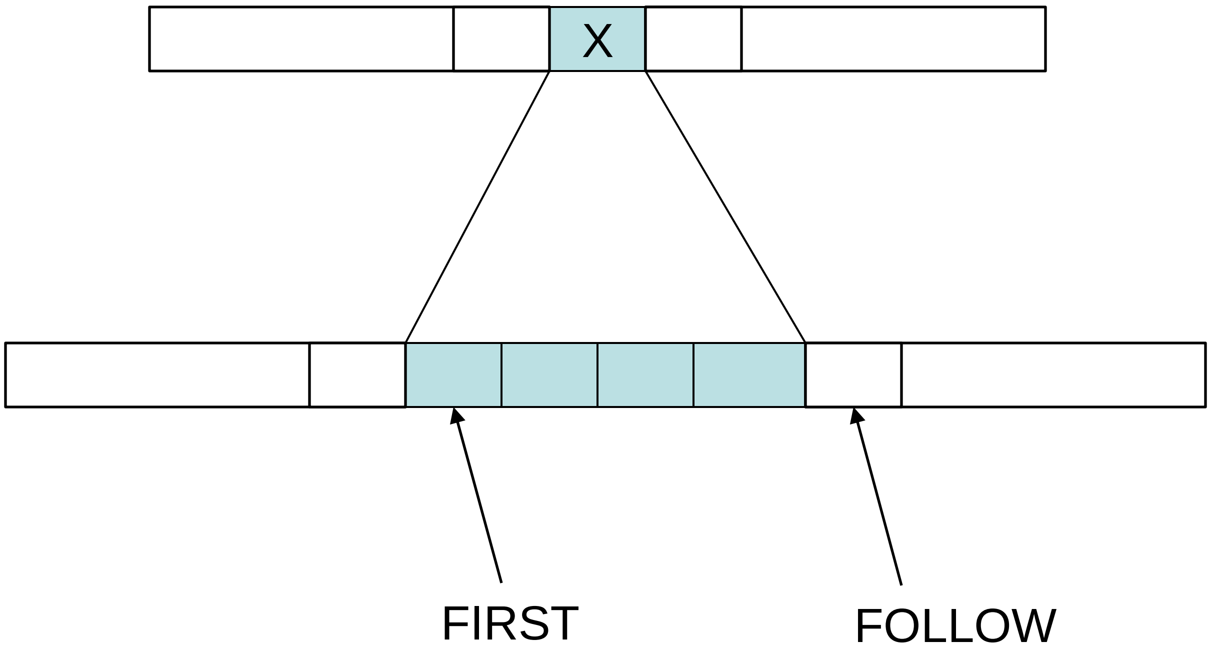
FIRST & FOLLOW

- 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

FIRST & FOLLOW



Algorithm for FIRST

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

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

- If X is a non terminal & $X \rightarrow Y_1 Y_2 \dots Y_i Y_j \dots Y_k$ 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 $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production then

  If ϵ is in all Y_i $\forall\ 1 \le i \le k$ then

  ϵ is in First(X)

Example: FIRST

- For the following grammar:

$$
\begin{gathered}
E \rightarrow TE' \\
E' \rightarrow +TE' \mid \varepsilon \\
T \rightarrow FT' \\
T' \rightarrow * FT' \mid \varepsilon \\
F \rightarrow (E) \mid id
\end{gathered}
$$

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)

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;

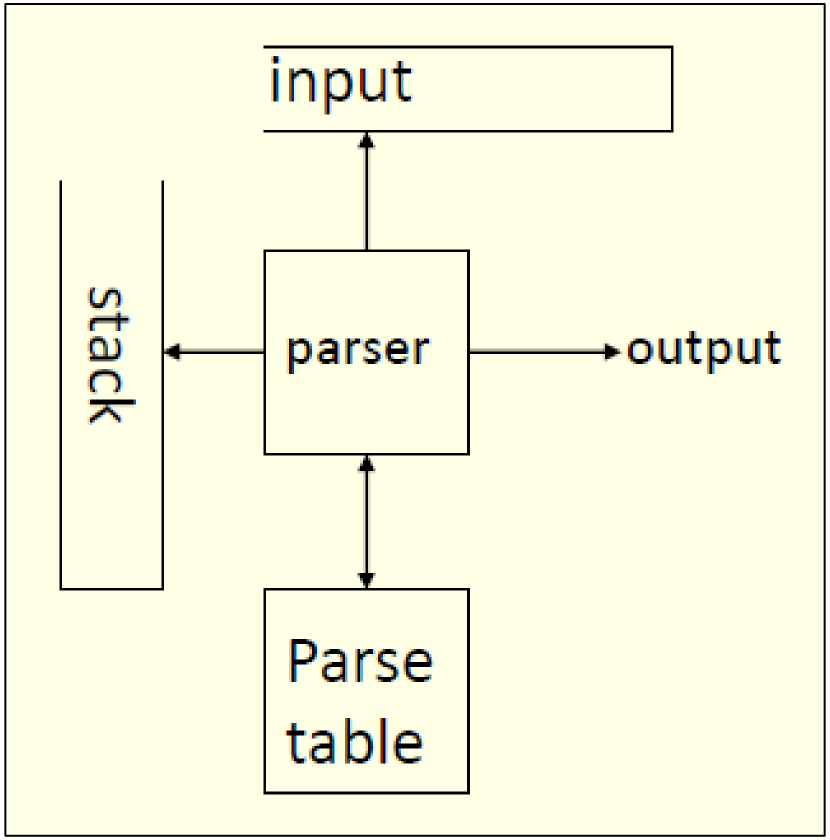
$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \epsilon$$
$$B \rightarrow \epsilon$$

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



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

Non 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	$\ldots$	$\ldots$	$\ldots$	$\ldots$

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	INPUT SYMBOL					
	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	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

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.

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

Pop X from the stack

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

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

else

error()

Until $X = \$$

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 $FIRST(\alpha) \cap FOLLOW(A) = \phi$

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 \$	$S \rightarrow AS'$

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' \$	\$	$S' \rightarrow \epsilon$

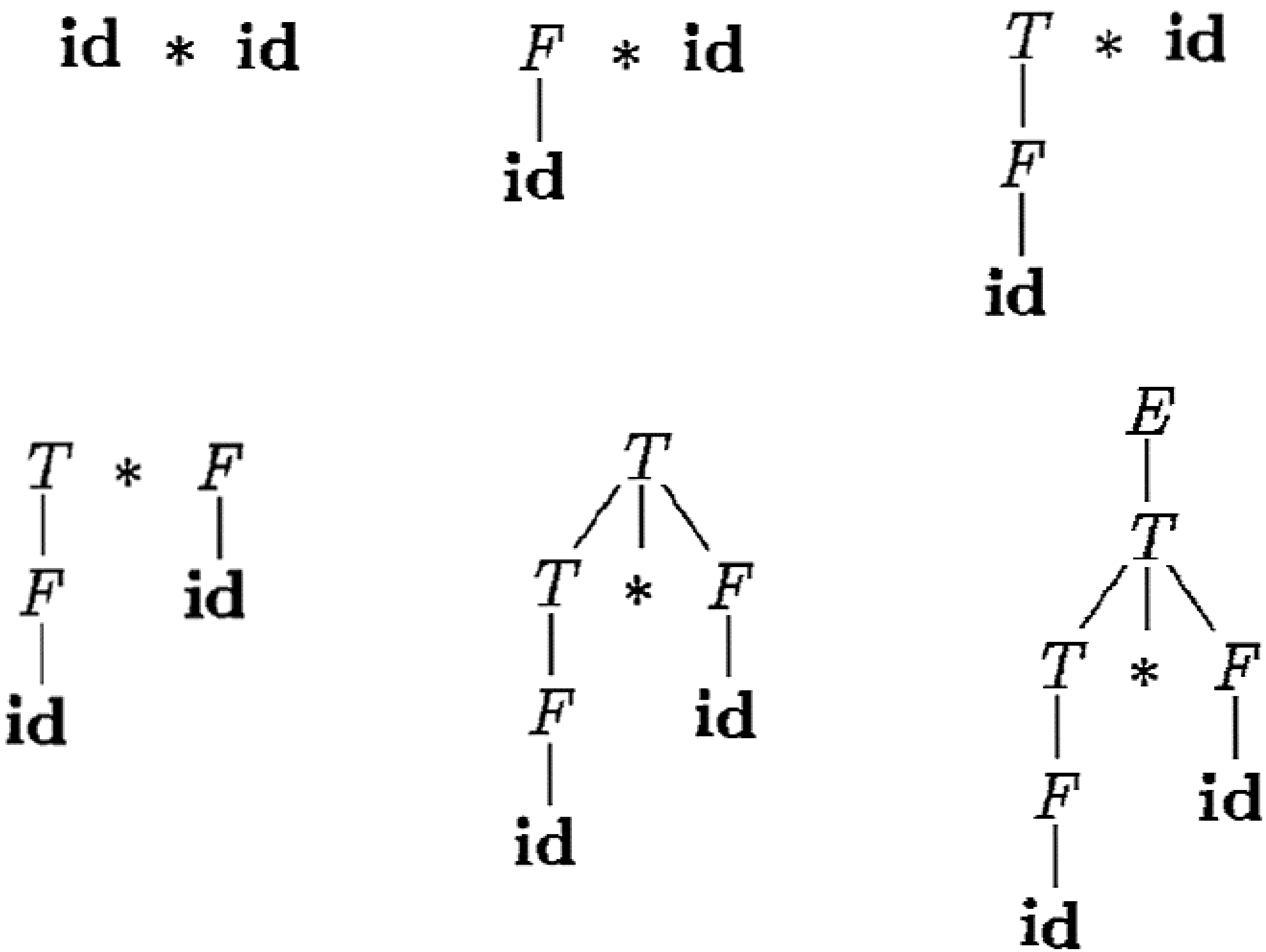
Example: Predictive Parsing

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



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
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

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

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

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

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)

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

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

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

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

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

Bottom up Parsing

- How does Shift-Reduce parsers decide when to Shift/Reduce?
  - Items
  - Closures
  - Canonical LR(0) collection / SLR automaton

$$A \to XYZ$$

This can produce four items:

$$A \to .XYZ$$
$$A \to X.YZ$$
$$A \to XY.Z$$
$$A \to XYZ.$$

Bottom up Parsing

If I is a set of items for a grammar G, then CLOSURE(I) is constructed using the following algorithm

1. Initially, add every item in I to CLOSURE(I)

2. If $A \rightarrow \alpha.B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production
   a. If $B \rightarrow .\gamma$ is not in CLOSURE(I)
      i. Add the item $B \rightarrow .\gamma$ to CLOSURE(I)

3. Repeat step 2 until no further items can be added

Bottom up Parsing

- Consider the following grammar:

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

- Augmented Grammar is as follows:

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

Bottom up Parsing

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

- For the following item set:

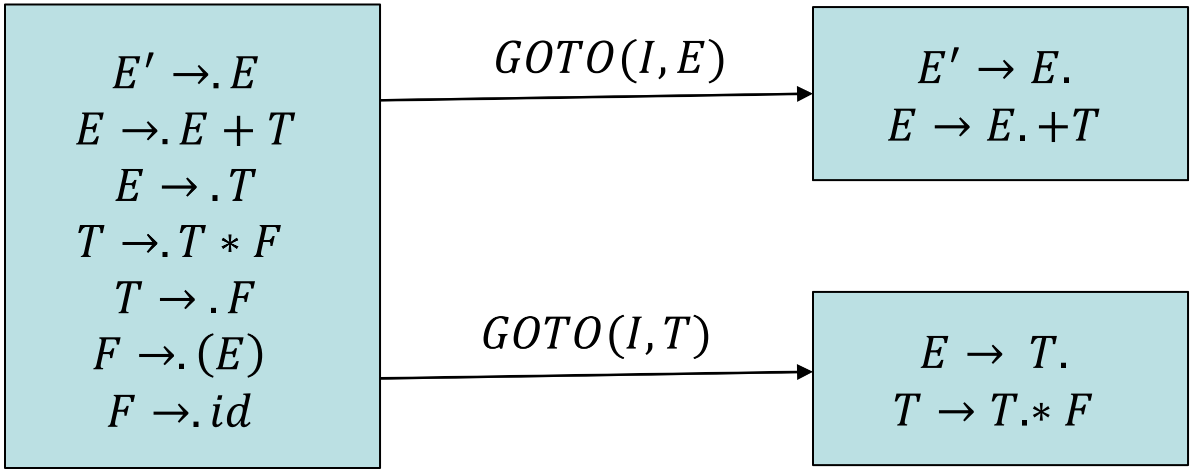
$$I_0 = \{E' \rightarrow .E\}$$

- The closure of the itemset will be:

$$I_0 = \left\{ \begin{array}{c} E' \rightarrow .E \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array} \right\}$$

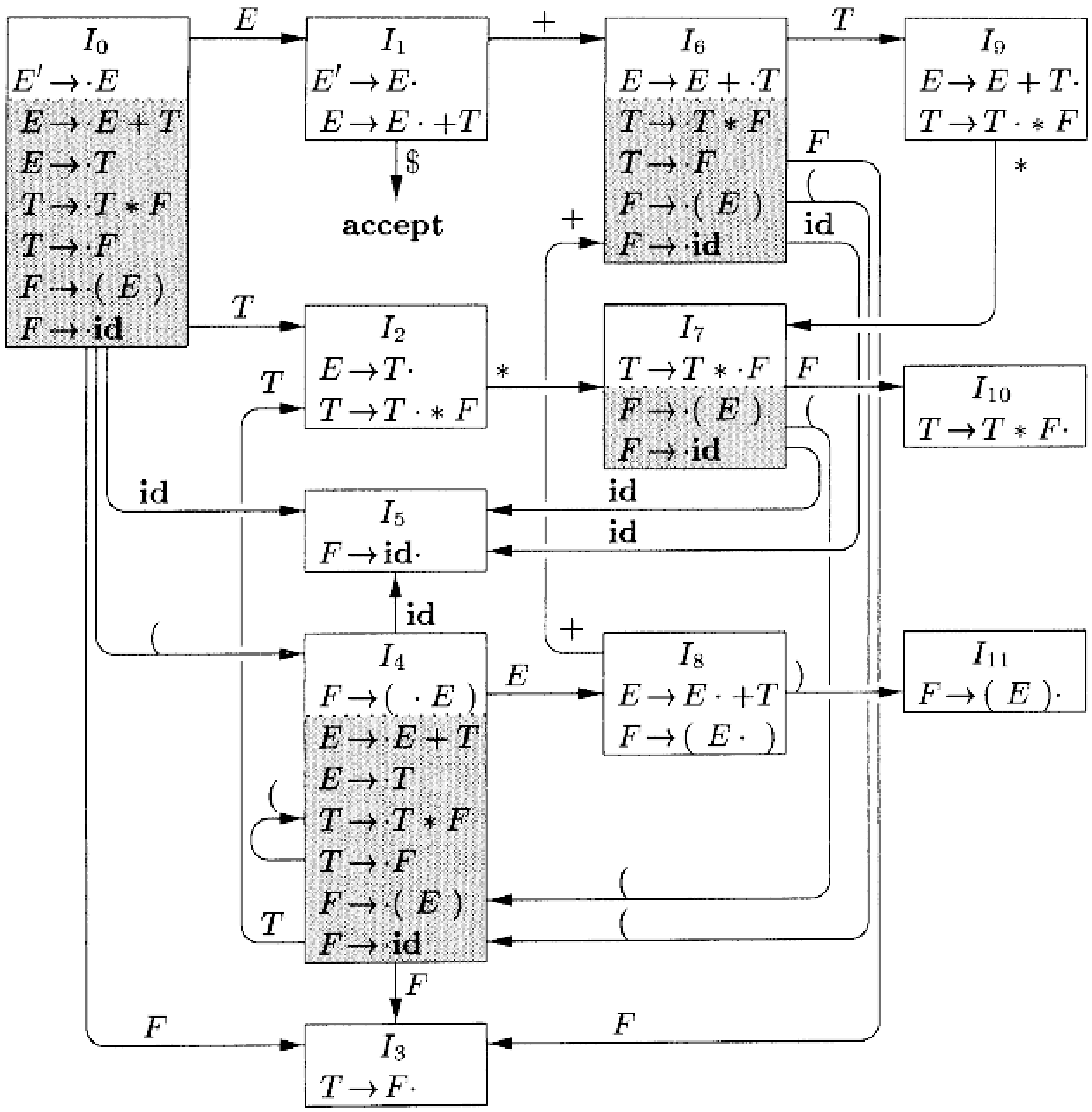
GOTO

- $GOTO(I, X)$, where I is a set of items and X is a grammar symbol,
  - is closure of set of item $A \rightarrow \alpha X.\beta$
  - such that $A \rightarrow \alpha.X\beta$ is in I



UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Bottom up Parsing



SLR Parsing Table Generation

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \rightarrow \alpha \cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A); here A may not be S'.

   (c) If $[S' \rightarrow S \cdot]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

SLR Parsing Table Generation

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

Step 1: Find out the FOLLOW sets of all Non-terminals of the Grammar

Step 2: Apply the SLR parsing table generation algorithm

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

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

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

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

UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

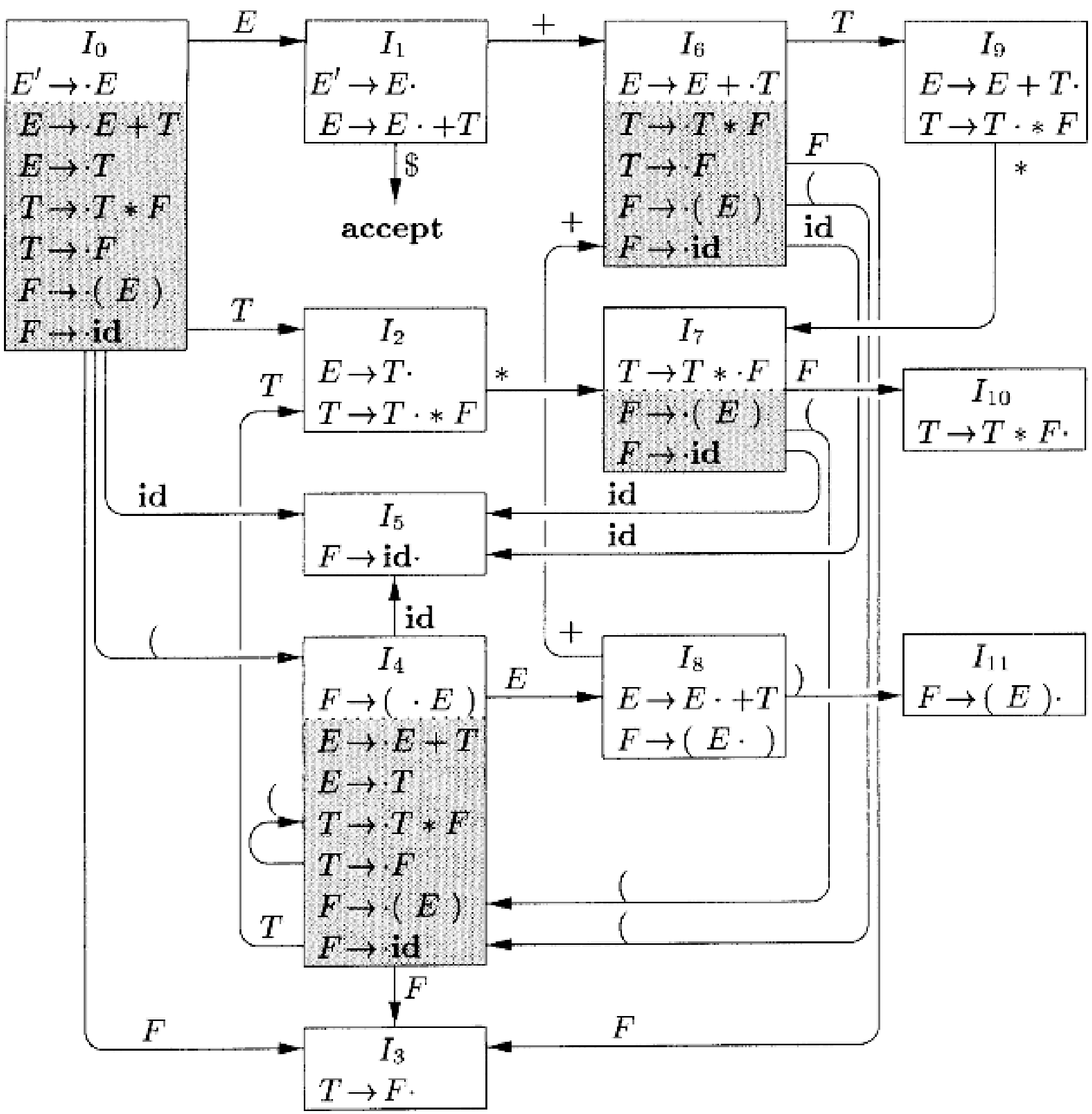
SLR Parsing Table Generation

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

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

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

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



SLR Parsing Table

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

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

LR Parsing Algorithm

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer.

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

LR Parsing Algorithm

- Parse the following string using the given SLR(1) parsing table: $id * id$

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

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

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	$	**id** * **id** $	shift to 5
(2)	0 5	$ **id**	* **id** $	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	$ F	* **id** $	reduce by $T \rightarrow F$
(4)	0 2	$ T	* **id** $	shift to 7
(5)	0 2 7	$ $T *$	**id** $	shift to 5
(6)	0 2 7 5	$ $T *$ **id**	$	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	$ $T * F$	$	reduce by $T \rightarrow T * F$
(8)	0 2	$ T	$	reduce by $E \rightarrow T$
(9)	0 1	$ E	$	accept