Top-down Parsing

Dinesh Gopalani
dgopalani.cse@mnit.ac.in

Top-down Parsing

- It is a process of finding a leftmost derivation for an input string w starting with the start symbol S.
- At each step an appropriate production is applied to a non-terminal (say A).
- After selecting A-production, next the process consists of matching the terminal symbols in the production body with the input string.

Top-down Parsing

Approach:

Grammar G:

$S \rightarrow c\ A\ d$

$A \rightarrow a\ b \mid a$

Input String w : c a d

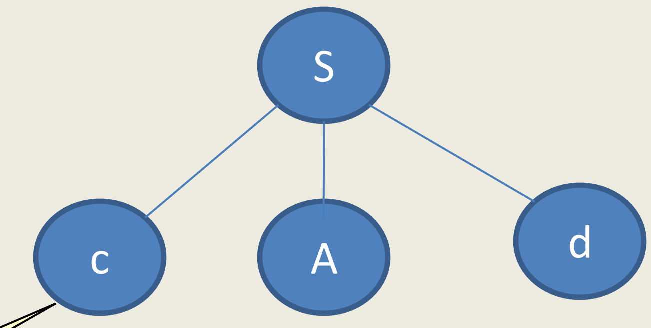
Top-down Parsing

Approach:

Grammar G:

$S \rightarrow c\ A\ d$

$A \rightarrow a\ b \mid a$



Match is successful

Input String w : c a d

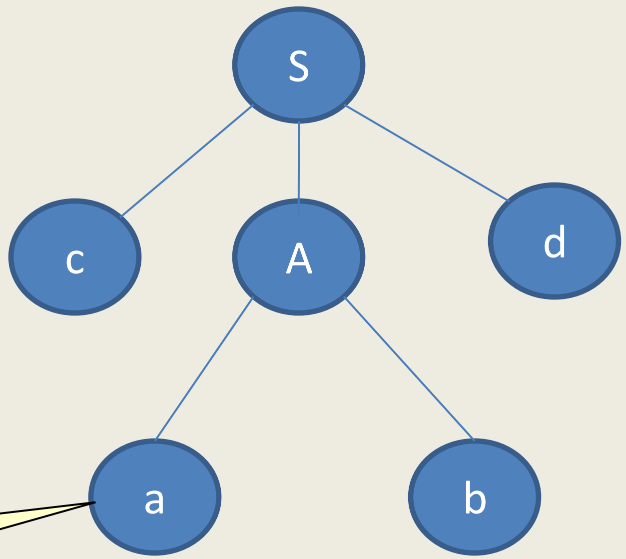
Top-down Parsing

Approach:

Grammar G:

$S \rightarrow c\ A\ d$

$A \rightarrow a\ b\ |\ a$



Match is successful

Input String w : c a d

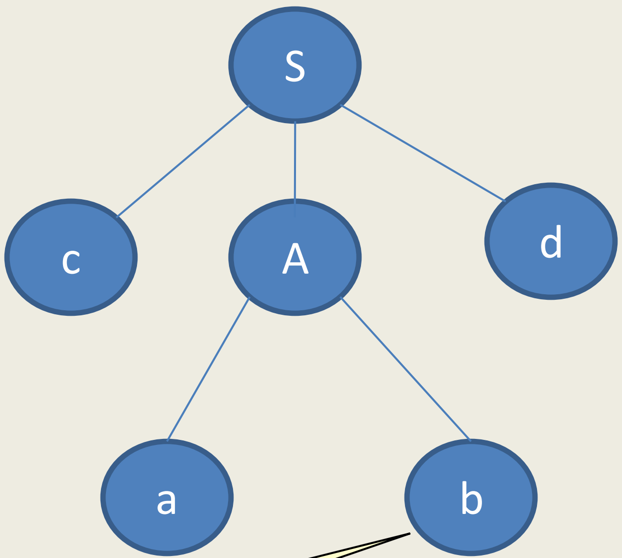
Top-down Parsing

Approach:

Grammar G:

$S \to c\ A\ d$

$A \to a\ b \mid a$



Match is failed
- Backtracking

Input String w : c a d

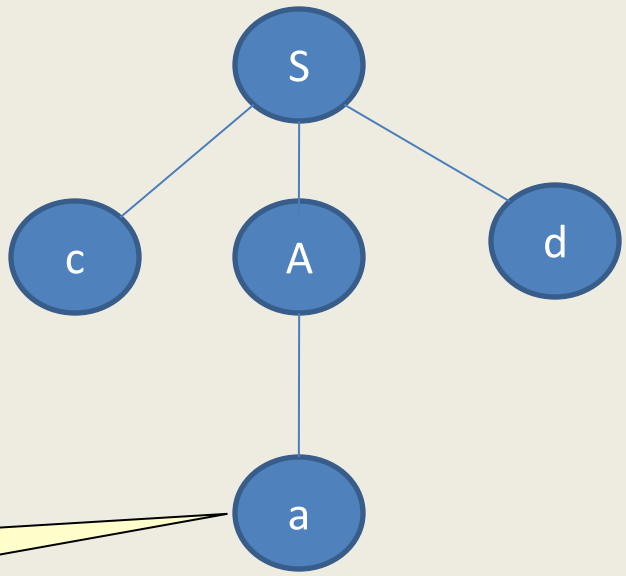
Top-down Parsing

Approach:

Grammar G:

$S \rightarrow c\ A\ d$

$A \rightarrow a\ b \mid a$



Match is successful

Input String w : c a d

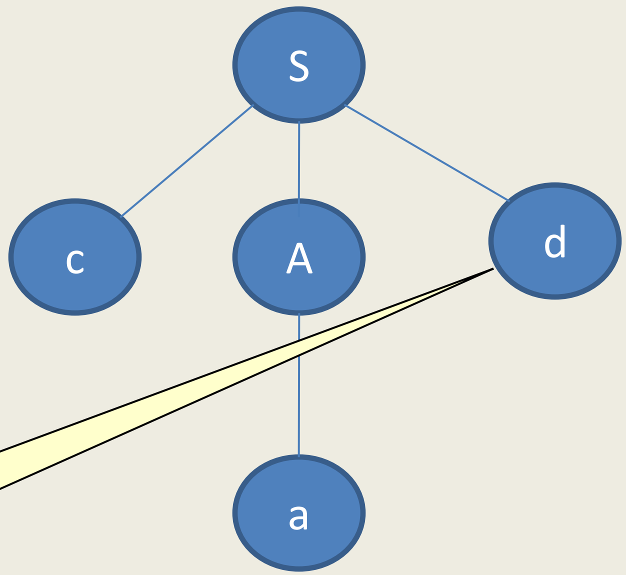
Top-down Parsing

Approach:

Grammar G:

$S \rightarrow c\ A\ d$

$A \rightarrow a\ b \mid a$



Match is successful

Input String w : c a d

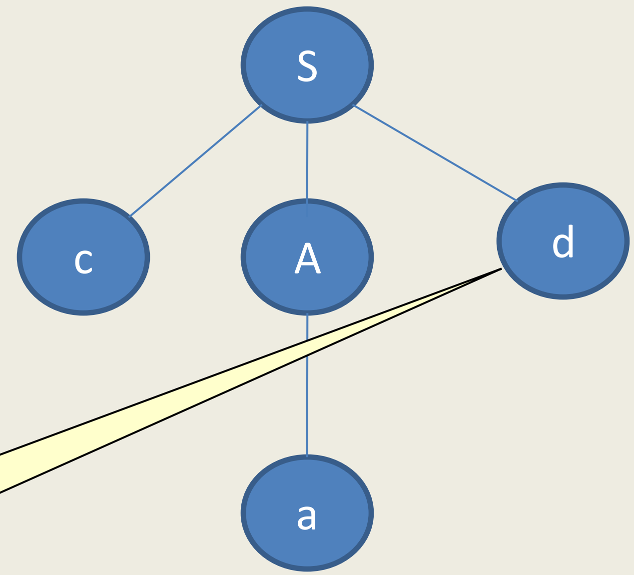
Top-down Parsing

Approach:

Grammar G:

$S \to c\ A\ d$

$A \to a\ b \mid a$



Match is successful

Input String w : c a d

Leftmost Derivation : $S \Rightarrow c\ A\ d \Rightarrow c\ a\ d$

Recursive-Descent Parser

- It consists of set of procedures (functions) for each non-terminal.
- The process starts with the function for the start symbol.
- It may require backtracking – may require repeated scans over the input.
- The backtracking shall be avoided as it will make parsing inefficient.

Recursive-Descent Parser

Example:

$D \rightarrow T\ L$

$T \rightarrow$ int | char

$L \rightarrow$ id , L | id ;

Recursive-Descent Parser

Example:

$D \rightarrow T\ L$

$T \rightarrow$ int | char

$L \rightarrow$ id , L | id ;

```
int D()
{
   T();
   L();
   return 1;
}
```

Recursive-Descent Parser

Example:

$D \rightarrow T\ L$

$T \rightarrow$ int | char

$L \rightarrow$ id , L | id ;

```
int D()
{
   T();
   L();
   return 1;
}
```

```
int T()
{
   if (input=="int"){
        Advance();
        return 1; }
   else if (input=="char"){
        Advance();
        return 1; }
   else
        return 0;
}
```

Recursive-Descent Parser

Example:

$D \rightarrow T\ L$

$T \rightarrow$ int | char

$L \rightarrow$ id , L | id ;

```
int D()
{
    T();
    L();
    return 1;
}
```

```
int T()
{
    if (input=="int"){
         Advance();
         return 1; }
    else if (input=="char"){
         Advance();
         return 1; }
    else
         return 0;
}
```

```
int L()
{
    isave=ip;
    if (input=="id"){
         Advance();
         if (input==","){
             Advance();
             L();
             return 1; }
    ip=isave;
    if (input=="id"){
         Advance();
         if (input==";"){
             Advance();
             return 1; }
    else
         return 0;
}
```

Problems with Top-down Parsing

Backtracking:

- If a sequence of wrong productions are used and subsequently a mismatch is discovered then the parser has to backtrack and try other set of productions.
- One major reason is common prefixes present in the input grammar.
- Example:

$D \rightarrow T\ L$

$T \rightarrow \text{int} \mid \text{char}$

$L \rightarrow \text{id}\ ,\ L \mid \text{id}\ ;$

Problems with Top-down Parsing

Backtracking:

- To fix the common prefixes problem, the process of left factoring may be used.
- Left Factoring :

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

$$A \rightarrow \alpha A'$$

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

Problems with Top-down Parsing

Backtracking:

- To fix the common prefixes problem, the process of left factoring may be used.
- Left Factoring :

$D \rightarrow T\ L$

$T \rightarrow \text{int} \mid \text{char}$

$L \rightarrow \text{id}\ ,\ L \mid \text{id}\ ;$

$L \rightarrow \text{id}\ L'$

$L' \rightarrow\ ,\ L \mid\ ;$

Problems with Top-down Parsing

Left-Recursion:

- A grammar G is said to be left-recursive if it has a non-terminal A such that there is a derivation A $\overset{+}{\Rightarrow}$ A α for some α.
- A left-recursive grammar can cause a top-down parser to go into an infinite loop – while trying to expand A, it may eventually find again trying to expand A without consuming any input.
- Therefore in order to use top-down parsing, all left-recursions must be eliminated from the input grammar.

Elimination of Left-Recursion

Immediate Left-Recursion:

- If there are left-recursive pair of productions

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

  (here β does not begin with A)

- Then left-recursion can be eliminated by replacing this pair of productions with the following

  $A \rightarrow \beta\ A'$

  $A' \rightarrow \alpha\ A' \mid \wedge$

Elimination of Left-Recursion

Immediate Left-Recursion:

(General Case)

- If there are left-recursion among A-productions as

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

  (here no β_i begins with A)

- Then left-recursion can be eliminated by replacing as

  $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

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

Elimination of Left-Recursion

Left-Recursion:

1. Arrange all non-terminals in some order $A_1, A_2, \ldots, A_n$.
2. Eliminate immediate left-recursion among A_1-productions.
3. for i = 2 to n do

    begin

    for j = 1 to i-1

    replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$, where

    $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are current A_j-productions.

    Eliminate immediate left-recursion among A_i-productions.

    end

Elimination of Left-Recursion

Left-Recursion:

1. Arrange all non-terminals in some order $A_1, A_2, \ldots, A_n$.
2. Eliminate immediate left-recursion among A_1-productions.
3. for i = 2 to n do

   begin

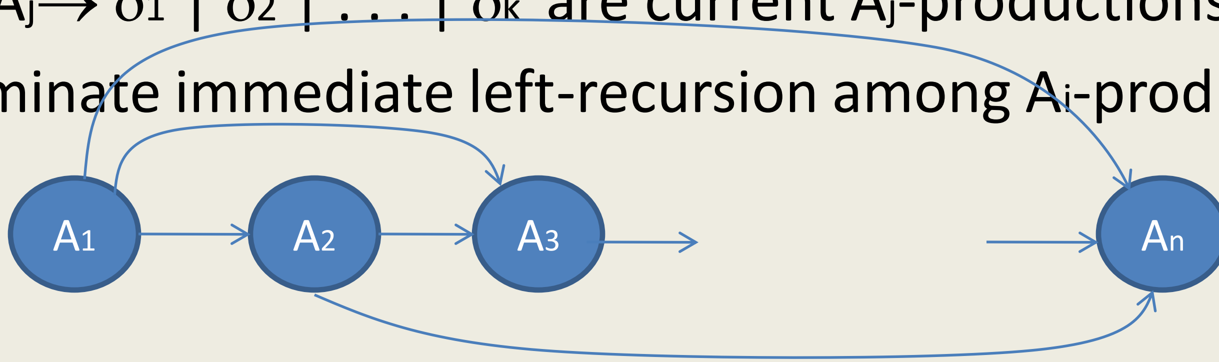
   for j = 1 to i-1

   replace each production of the form $Ai \rightarrow Aj\ \gamma$ by the productions $A_i \rightarrow \delta_1\ \gamma \mid \delta_2\ \gamma \mid \ldots \mid \delta_k\ \gamma$, where

   $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are current A_j-productions.

   Eliminate immediate left-recursion among A_i-productions.

   end



Elimination of Left-Recursion

Left-Recursion:

1. Arrange all non-terminals in some order $A_1, A_2, \ldots, A_n$.
2. Eliminate immediate left-recursion among A_1-productions.
3. for i = 2 to n do

   begin

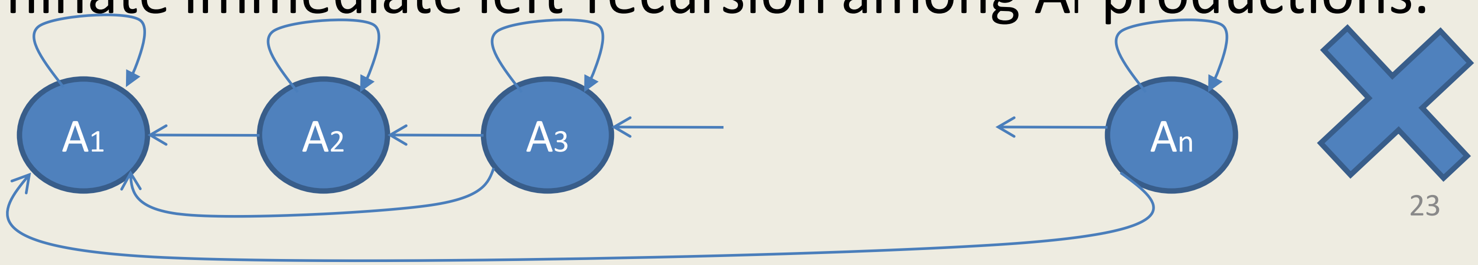
   for j = 1 to i-1

   replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$, where

   $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are current A_j-productions.

   Eliminate immediate left-recursion among A_i-productions.

   end



Elimination of Left-Recursion

Example:

$S \to A\ a \mid b$

$A \to A\ c \mid S\ d \mid e$

1. Rename S as A_1 and A as A_2.

   $A_1 \to A_2\ a \mid b$

   $A_2 \to A_2\ c \mid A_1\ d \mid e$

2. Eliminate immediate left-recursion among A_1-productions – Not Applicable here.
3. for (i=2, j=1)

   Replace $A_2 \to A_1\ d$ by

   $A_2 \to A_2\ a\ d \mid b\ d$

   Now A_2 productions are $A_2 \to A_2\ a\ d \mid b\ d \mid A_2\ c \mid e$

Elimination of Left-Recursion

Example:

3. (Contd.) for (i=2)

Eliminate immediate left recursions among A_i productions:

$A_2 \rightarrow A_2\ a\ d \mid b\ d \mid A_2\ c \mid e$

$A_2 \rightarrow b\ d\ A_2' \mid e\ A_2'$

$A_2' \rightarrow a\ d\ A_2' \mid c\ A_2' \mid \wedge$

So, final grammar is

$A_1 \rightarrow A_2\ a \mid b$

$A_2 \rightarrow b\ d\ A_2' \mid e\ A_2'$

$A_2' \rightarrow a\ d\ A_2' \mid c\ A_2' \mid \wedge$

LL(1) Parser

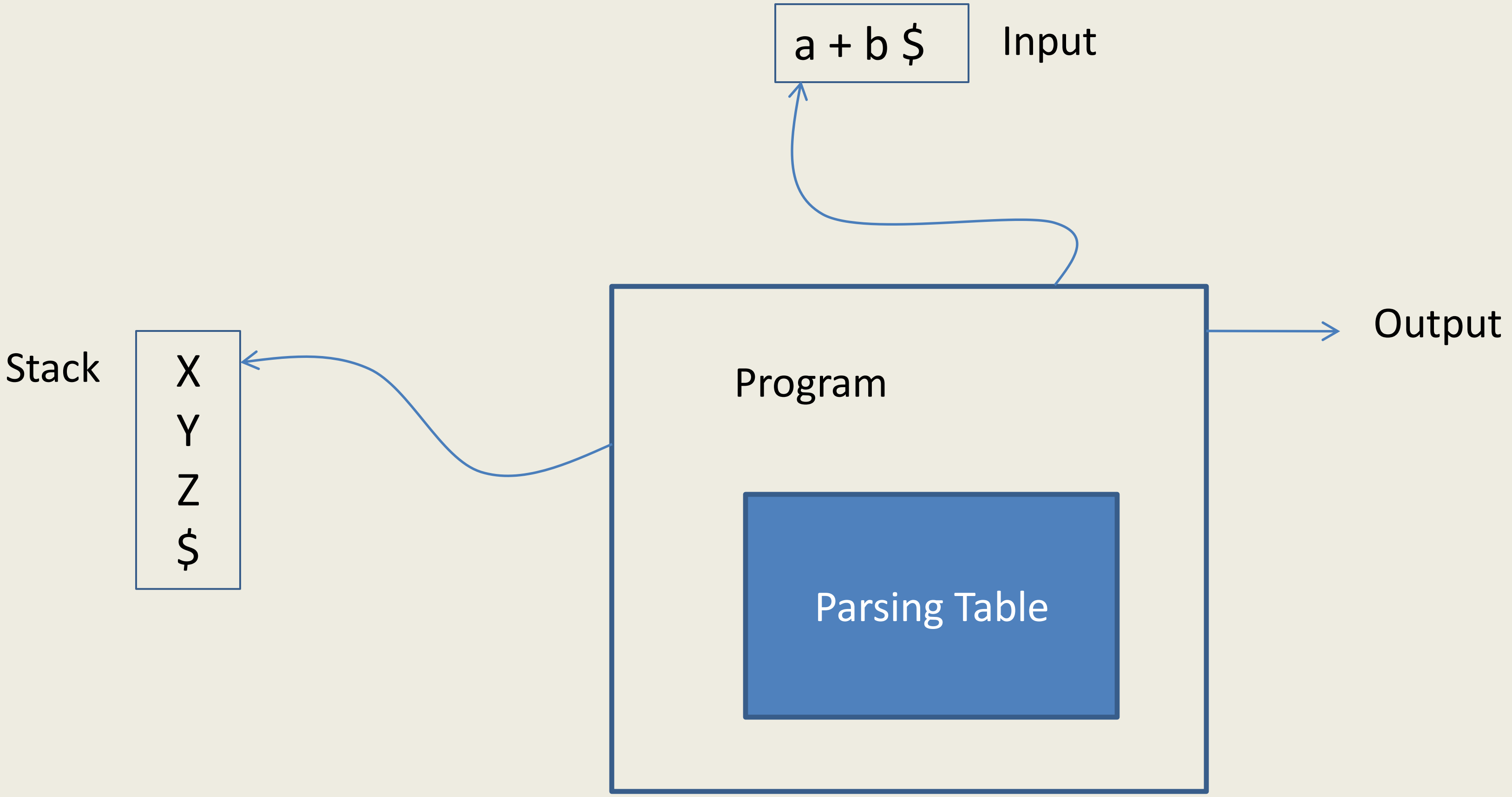
- Based on Top-down approach.
- Significance of the name:

  L - Left-to-right scanning of input

  L - Leftmost derivation

  (1) - Number of input symbol(s) to make parsing decision
- It avoids backtracking.
- It uses Parsing Table to decide what production is to be applied at each step.

LL(1) Parser



Working of LL(1) Parser

The parser determines X, the symbol on TOS and a, the current input symbol :

1. If X = a = \$, the parser halts and announces successful completion of parsing.
2. If X = a $\neq$ \$, the parser pops X off the Stack and input pointer is advanced.
3. If X is a non-terminal then the parser checks parsing table M[X,a] which is either an X-production or an error. If entry is a production (X $\rightarrow$ UVW), the parser replaces X on TOS by WVU (with U on top).

Construction of LL(1) Parsing Table

First and Follow functions:

First(α) – It is the set of terminals that appear in the beginning of the strings derived from α.

If $\alpha \stackrel{*}{\Rightarrow} a\, \beta$, then a is in First($\alpha$).

If $\alpha \stackrel{*}{\Rightarrow}$ ^, then ^ is also in First(α).

Follow(A) – It is the set of terminals that can appear immediately on the right of A in some sentential form.

If $S \stackrel{*}{\Rightarrow} \alpha\, A\, a\, \beta$, for some α and β, then a is in Follow(A).

If $S \stackrel{*}{\Rightarrow} \alpha\, A$, then \$ is in Follow(A).

First Function

First(X):

1. If X is a terminal (say a)

   First(a)={a}

2. If X is a non-terminal (say A)

   (i) If $A \rightarrow a\, \alpha$ is a production then add a to First(A).

   For $A \rightarrow$ ^, add ^ to First(A).

   (ii) If $A \rightarrow Y_1\ Y_2\ \ldots\ Y_k$ is a production, then

   - Add every non-^ symbol in First(Y_1) to First(A)
   - If ^ is in First(Y_1), then add every non-^ symbol in First(Y_2) to First(A)
   - If ^ is in both First(Y_1) and First(Y_2), then add every non-^ symbol in First(Y_3) to First(A)
   - … (The process goes on)
   - Finally, if ^ is in First(Y_i) for all i=1,2, …, k then add ^ to First(A).

First Function

First(α) : $\alpha = X_1 X_2 \ldots X_k$

- Add every non-^ symbol in First(X_1) to First(α)
- If ^ is in First(X_1), then add every non-^ symbol in First(X_2) to First(α)
- If ^ is in both First(X_1) and First(X_2), then add every non-^ symbol in First(X_3) to First(α)
- … (The process goes on)
- Finally, if ^ is in First(X_i) for all i=1,2, …, k then add ^ to First(α).

Follow Function

Follow(A) :

- If A is a start symbol then add $ in Follow(A)
- If there is a production $B \rightarrow \alpha\ A\ \beta$ ($\beta \neq$ ^), then add every non-^ symbol in First(β) to Follow(A)
- If there is a production $B \rightarrow \alpha\ A$, or a production $B \rightarrow \alpha\ A\ \beta$ where First(β) contains ^, then add every symbol in Follow(B) to Follow(A)

Follow Function

Follow(A) :

- If A is a start symbol then add $ in Follow(A)
- If there is a production $B \rightarrow \alpha\ A\ \beta$ ($\beta \neq$ ^), then add every non-^ symbol in First(β) to Follow(A)
- If there is a production $B \rightarrow \alpha\ A$, or a production $B \rightarrow \alpha\ A\ \beta$ where First(β) contains ^, then add every symbol in Follow(B) to Follow(A)

If $S \overset{*}{\Rightarrow} \gamma\ B\ a\ \delta \Rightarrow \gamma\ \alpha\ A\ a\ \delta$

then a which is following B, will also follow A as well.

First and Follow Functions

Example:

$S \to A\ B \mid d\ S$

$A \to a\ A\ b \mid \wedge$

$B \to b\ B\ A \mid c$

First (S) = {}

First (A) = {a, ^}

First (B) = {b, c}

First and Follow Functions

Example:

$S \rightarrow A\ B \mid d\ S$

$A \rightarrow a\ A\ b \mid$ ^

$B \rightarrow b\ B\ A \mid c$

First (S) = {a, b, c, d}

First (A) = {a, ^}

First (B) = {b, c}

First and Follow Functions

Example:

$S \rightarrow A\ B \mid d\ S$

$A \rightarrow a\ A\ b \mid$ ^

$B \rightarrow b\ B\ A \mid c$

First (S) = {a, b, c, d}

First (A) = {a, ^}

First (B) = {b, c}

Follow(S) = {$}

Follow(A) = {b, c, a, $}

Follow(B) = {$, a}

Construction of LL(1) Parsing Table

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in First(α), add $A \rightarrow \alpha$ to M[A,a].
3. If ^ is in First(α), add $A \rightarrow \alpha$ to M[A,b] for each terminal b (including $) in Follow(A).
4. Make all remaining entries of M as Errors.

Construction of LL(1) Parsing Table

Example:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Eliminate left-recursion:

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid$ ^

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid$ ^

$F \rightarrow (E) \mid id$

Construction of LL(1) Parsing Table

Example:

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid$ ^

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid$ ^

$F \rightarrow (E) \mid id$

First (E) = { (, id }

First (E') = { +, ^ }

First (T) = { (, id }

First (T') = { *, ^ }

First (F) = { (, id }

Follow(E) = { $,) }

Follow(E') = { $,) }

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

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

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

Construction of LL(1) Parsing Table

Example:

E → T E’

E’ → + T E’ | ^

T → F T’

T’ → * F T’ | ^

F → (E) | id

	id	+	*	(	)	$
E	E → T E’			E → T E’		
E’		E’ → + T E’			E’ → ^	E’ → ^
T	T → F T’			T → F T’		
T’		T’ → ^	T’ → * F T’		T’ → ^	T’ → ^
F	F → id			F → (E)		

First (E) = { (, id }

First (E’) = { +, ^ }

First (T) = { (, id }

First (T’) = { *, ^ }

First (F) = { (, id }

Follow(E) = { $,) }

Follow(E’) = { $,) }

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

Follow(T’) = { +, $,) }

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

Working of LL(1) Parser

The parser determines X, the symbol on TOS and a, the current input symbol :

1. If X = a = \$, the parser halts and announces successful completion of parsing.
2. If X = a $\neq$ \$, the parser pops X off the Stack and input pointer is advanced.
3. If X is a non-terminal then the parser checks parsing table M[X,a] which is either an X-production or an error. If entry is a production ($X \rightarrow UVW$), the parser replaces X on TOS by WVU (with U on top).

Working of LL(1) Parser

Example:

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \wedge$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \wedge$

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

Stack	Input	Action
$E	id+id*id$	$E \rightarrow T\ E'$

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

Working of LL(1) Parser

Example:

E → T E'

E' → + T E' | ^

T → F T'

T' → * F T' | ^

F → (E) | id

Stack	Input	Action
$E	id+id*id$	E → T E'
$E'T	id+id*id$	T → F T'

	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ^	E' → ^
T	T → F T'			T → F T'		
T'		T' → ^	T' → * F T'		T' → ^	T' → ^
F	F → id			F → (E)		

Working of LL(1) Parser

Example:

E → T E'

E' → + T E' | ^

T → F T'

T' → * F T' | ^

F → (E) | id

Stack	Input	Action
$E	id+id*id$	E → T E'
$E'T	id+id*id$	T → F T'
$E'T'F	id+id*id$	F → id

	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ^	E' → ^
T	T → F T'			T → F T'		
T'		T' → ^	T' → * F T'		T' → ^	T' → ^
F	F → id			F → (E)		

Working of LL(1) Parser

Example:

E → T E’

E’ → + T E’ | ^

T → F T’

T’ → * F T’ | ^

F → (E) | id

Stack	Input	Action
$E	id+id*id$	E → T E’
$E’T	id+id*id$	T → F T’
$E’T’F	id+id*id$	F → id
$E’T’id	id+id*id$	Match

	id	+	*	(	)	$
E	E → T E’			E → T E’		
E’		E’ → + T E’			E’ → ^	E’ → ^
T	T → F T’			T → F T’		
T’		T’ → ^	T’ → * F T’		T’ → ^	T’ → ^
F	F → id			F → (E)		

Working of LL(1) Parser

Example:

E → T E'

E' → + T E' | ^

T → F T'

T' → * F T' | ^

F → (E) | id

Stack	Input	Action
$E	id+id*id$	E → T E'
$E'T	id+id*id$	T → F T'
$E'T'F	id+id*id$	F → id
$E'T'id	id+id*id$	Match
$E'T'	+id*id$	T' → ^

	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ^	E' → ^
T	T → F T'			T → F T'		
T'		T' → ^	T' → * F T'		T' → ^	T' → ^
F	F → id			F → (E)		

Working of LL(1) Parser

Example:

E → T E'

E' → + T E' | ^

T → F T'

T' → * F T' | ^

F → (E) | id

Stack	Input	Action
$E	id+id*id$	E → T E'
$E'T	id+id*id$	T → F T'
$E'T'F	id+id*id$	F → id
$E'T'id	id+id*id$	Match
$E'T'	+id*id$	T' → ^
$E'	+id*id$	E' → + T E'
$E'T+	+id*id$	Match
$E'T	id*id$	T → F T'
$E'T'F	id*id$	F → id
$E'T'id	id*id$	Match
$E'T'	*id$	T' → * F T'
$E'T'F*	*id$	Match
$E'T'F	id$	F → id
$E'T'id	id$	Match
$E'T'	$	T' → ^
$E'	$	E' → ^
$	$	Accepted

	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ^	E' → ^
T	T → F T'			T → F T'		
T'		T' → ^	T' → * F T'		T' → ^	T' → ^
F	F → id			F → (E)		