Parsing …

Top-down parsing

- Regular languages
  - The weakest formal languages widely used
  - Many applications

- Lexical Analysis
- VLSI design → minimimum DFA is useful
- Scene understanding – components, relationships.
  - Reconstructing a crime event.
- Etc

But,

Consider the language:

$$\{(^i)^i \mid i \geq 0\}$$

$$\left\{\begin{array}{l} () \\ (()) \\ (((\ ))) \\ \vdots \end{array}\right.$$

((1+2) * 3)

if then

if then

if then

fi

fi

fi

Palindromes, Etc.

CFGs

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

CFGs

- Programming languages have recursive structure

- An EXPR is

  if EXPR then EXPR else EXPR fi

  while EXPR loop EXPR pool

  ...

- Context-free grammars are a natural notation for this recursive structure

CFGs

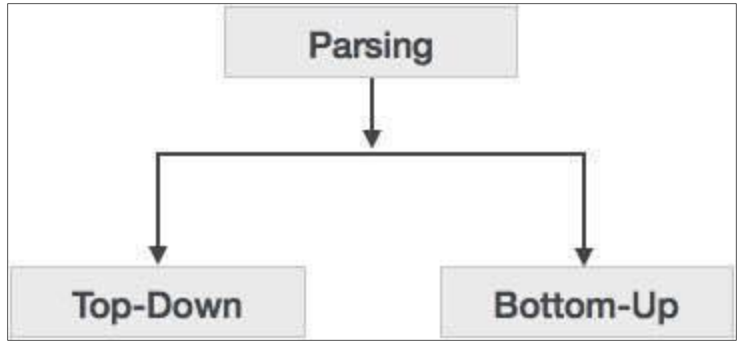
Which of the strings are in the language of the given CFG?

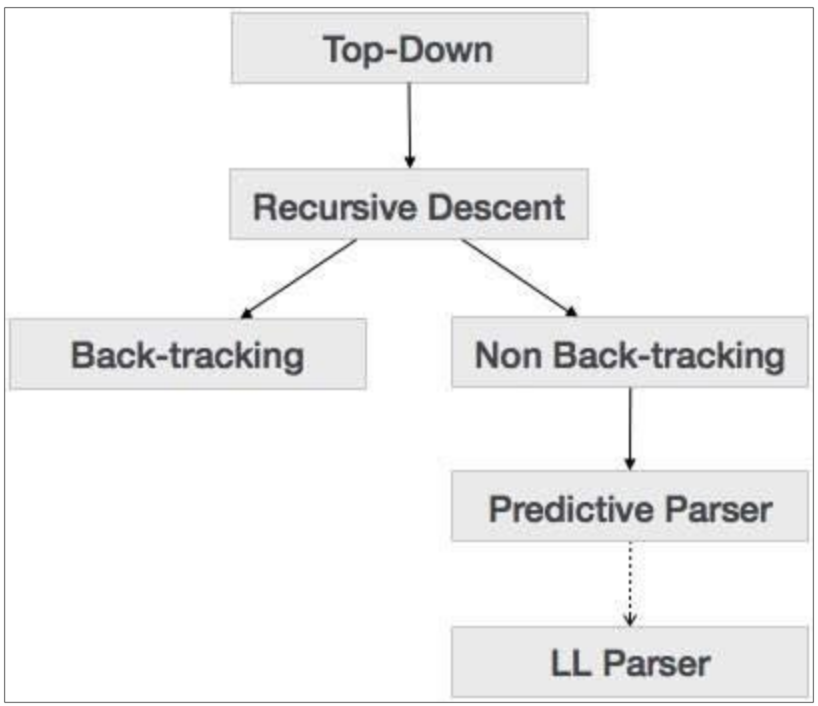
- ☐ abcba
- ☐ acca
- ☐ aba
- ☐ abcbcba

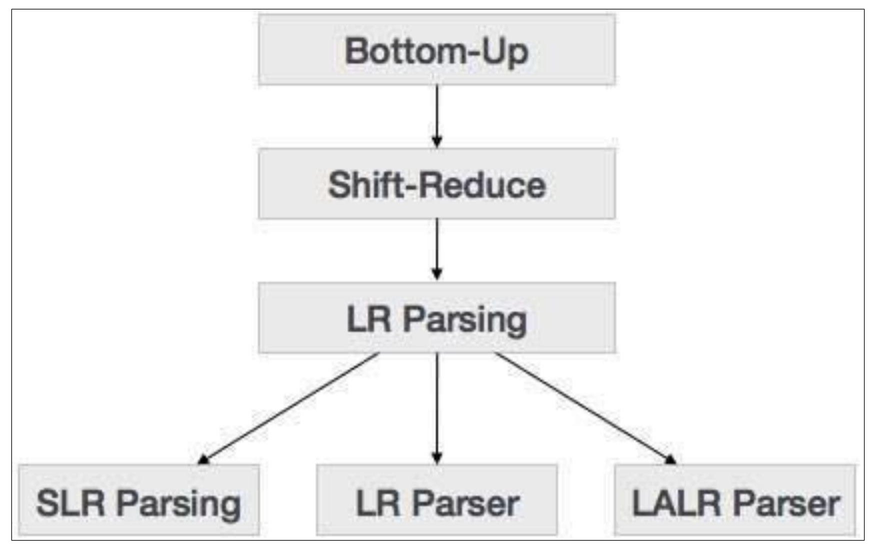
$$
\begin{aligned}
S &\rightarrow aXa \\
X &\rightarrow \varepsilon \\
&\mid\ bY \\
Y &\rightarrow \varepsilon \\
&\mid\ cXc
\end{aligned}
$$

Derivations

- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation


Parsing
Top-Down
Bottom-Up


Top-Down
Recursive Descent
Back-tracking
Non Back-tracking
Predictive Parser
LL Parser


Bottom-Up
Shift-Reduce
LR Parsing
SLR Parsing
LR Parser
LALR Parser

Top-Down Parsing

Top-down Parsing

- Tree is built from root to leaves
- Depth first (preorder)
- Leftmost derivations are used
- General technique requires backtracking
- Recursive algorithms seems alright, but are quite expensive
  - So, we may look ahead in the input string for k characters, do some preprocessing like building some tables, etc (*and thus choose the right production to be used*) and thus may be the need for backtracking can be eliminated.

4.4.1 Recursive-Descent Parsing

Order A productions, and choose according to this order.

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Break the loop and go for the next A production. You need to retract in reading the input string.

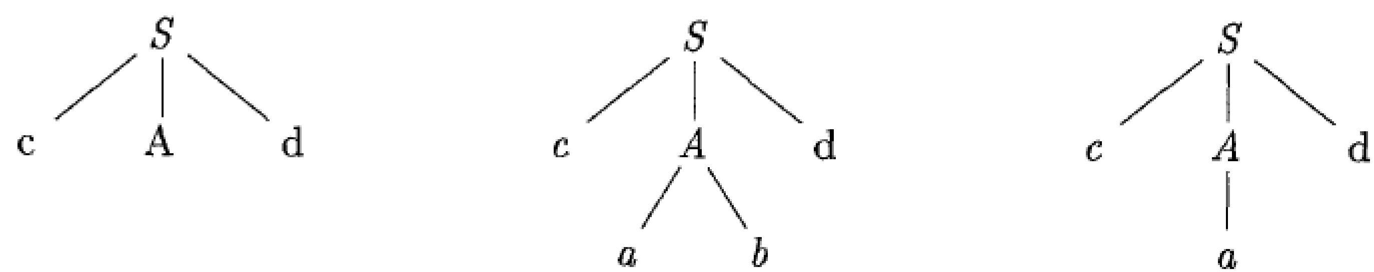
Figure 4.13: A typical procedure for a nonterminal in a top-down parser

For each variable we write a function like this.
First we call S( ); /* S is the start symbol */

Example 4.29 : Consider the grammar

$$\begin{array}{lcl} S & \rightarrow & c\ A\ d \\ A & \rightarrow & a\ b \ \mid \ a \end{array}$$

$\boxed{w = cad}$



(a) (b) (c)

Figure 4.14: Steps in a top-down parse

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

E
|
T
|
int

Mismatch: int does not match (
Backtrack …

$(\; int_5 \;)$

↑

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

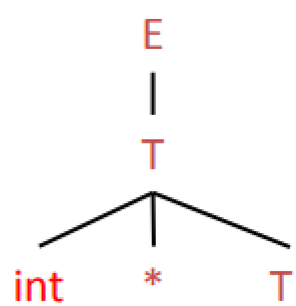
$$\begin{array}{c} E \\ | \\ T \end{array}$$

$(\; int_5 \;)$

$\uparrow$

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



Mismatch: int does not match (
Backtrack …

$(\; int_5 \;)$

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

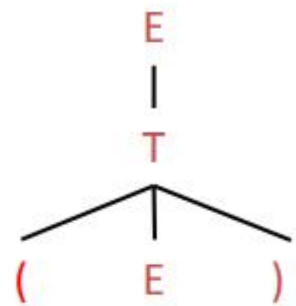
$$\begin{array}{c} E \\ | \\ T \end{array}$$

$(\; int_5 \;)$

$\uparrow$

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

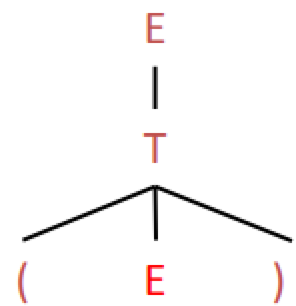


Match! Advance input.

$(\; int_5 \;)$

$E \rightarrow T \mid T + E$

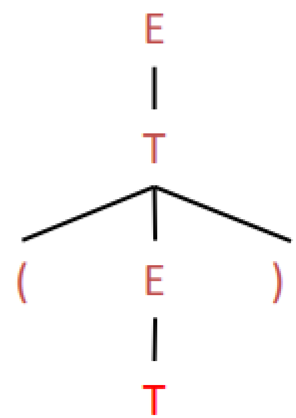
$T \rightarrow int \mid int * T \mid (E)$



(int_5)

$E \rightarrow T \mid T + E$

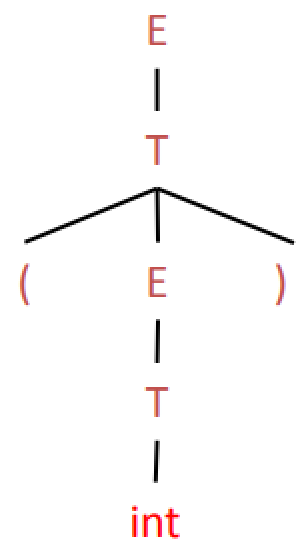
$T \rightarrow int \mid int * T \mid (E)$



$(\; int_5 \;)$

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

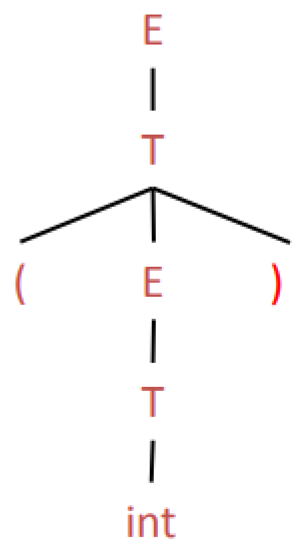


Match! Advance input.

(int_5)

$$E \to T \mid T + E$$

$$T \to \text{int} \mid \text{int} * T \mid (E)$$

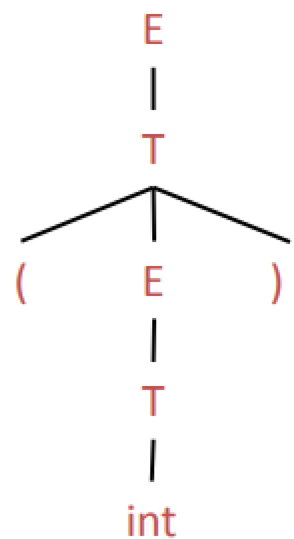


Match! Advance input.

(int_5)

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



End of input, accept.

(int_5)

Predictive Parsing

- We want to predict the correct production to be used (at a given stage of parsing).
- We do not want backtracking.
- These are called LL(1)
- A **parse table** is built which gives us the production to be used (in a given stage of parsing).

Parse table

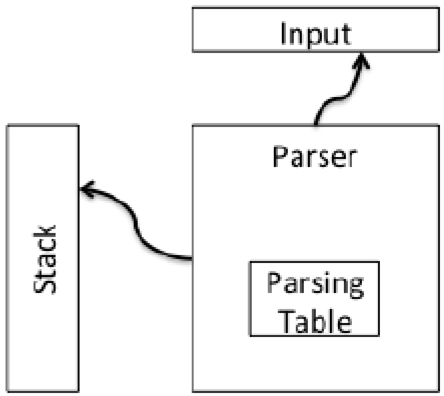
The parse table tells a top-down parser which productions might possibly be applicable to the current input token.

So if we are doing top-down parsing, possibly with backtracking, the parse table limits the search — we only try productions that can actually lead (eventually) to production of the current input symbol.

In the best case — with predictive parsing — the parse table is deterministic, eliminating the need for backtracking.

So, if A is the current leftmost variable, then the parse table tells us which A-productions can be used to produce the current input token. (Or, if A is nullable, whether it would be a good idea to derive ϵ from A.)

LL(1) Parsing Algorithm




```
Initial configuration: Stack = S, Input = w$,
where, S = start symbol, $ = end of file marker
repeat {
   let X be the top stack symbol;
   let a be the next input symbol /*may be $*/;
   if X is a terminal symbol or $ then
        if X == a then {
              pop X from Stack;
              remove a from input;
        } else ERROR();
   else  /* X is a non-terminal symbol */
         if M[X,a] == X → Y1Y2... Yk then {
                pop X from Stack;
                push Yk, Yk-1, ..., Y1 onto Stack;
                        (Y1 on top)
         }
} until Stack has emptied;
```

FIRST(α)

In order to conveniently specify the parse table for a grammar, we define two auxiliary functions: FIRST and FOLLOW.

For every string α over $V \cup \Sigma$, FIRST(α) is the set consisting of

- all terminals a s.t.

$$\alpha \overset{*}{\Rightarrow} a\beta$$

  for some string β over $V \cup \Sigma$, along with

- ϵ, if

$$\alpha \overset{*}{\Rightarrow} \epsilon .$$

Remember this. There is a scope for confusion here.

ϵ is in FIRST(α), if $\alpha \overset{*}{\Rightarrow} \epsilon$.
That is, Entire sentential form α can vanish

Example

$$\begin{aligned} S &\rightarrow XSa \mid Yc \\ X &\rightarrow aY \mid YY \\ Y &\rightarrow bSa \mid cX \mid \epsilon \end{aligned}$$

FIRST(S) = {a, b, c}

$\text{FIRST}(X) = \{a, b, c, \epsilon\}$ so $\text{FIRST}(XSa) = \{a, b, c\}$

$\text{FIRST}(Y) = \{b, c, \epsilon\}$ so $\text{FIRST}(Yc) = \{b, c\}$

$\text{FIRST}(a) = \{a\} = \text{FIRST}(aY)$ $\quad$ $\text{FIRST}(cX) = \{c\}$

$\text{FIRST}(YY) = \{b, c, \epsilon\}$ $\quad$ $\text{FIRST}(\epsilon) = \{\epsilon\}$

$\text{FIRST}(bSa) = \{b\}$

- **Find nullable variables first. It helps a lot.**

To compute $\text{FIRST}(X)$ for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.

2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.

3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to $\text{FIRST}(X_1 X_2 \cdots X_n)$ all non-ϵ symbols of $\text{FIRST}(X_1)$. Also add the non-ϵ symbols of $\text{FIRST}(X_2)$, if ϵ is in $\text{FIRST}(X_1)$; the non-ϵ symbols of $\text{FIRST}(X_3)$, if ϵ is in $\text{FIRST}(X_1)$ and $\text{FIRST}(X_2)$; and so on. Finally, add ϵ to $\text{FIRST}(X_1 X_2 \cdots X_n)$ if, for all i, ϵ is in $\text{FIRST}(X_i)$.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}
\tag{4.28}
$$

Variable	FIRST
F	{ (, id }
T	FIRST(FT’) = FIRST(F) = { (, id } /* F is not nullable */
E	FIRST(TE’) = FIRST(T) = { (, id } /* T is not nullable */
E’	{ +, ε }
T’	{ *, ε }

FOLLOW(X)

For every variable A, FOLLOW(A) is the set consisting of

- all terminals a s.t.

$$S \stackrel{*}{\Rightarrow} \alpha A a \beta$$

  for some strings α, β over $V \cup \Sigma$, along with

- \$, if

$$S \stackrel{*}{\Rightarrow} \alpha A$$

  for some string α over $V \cup \Sigma$.

(Note: Here we have assumed that S *is the start symbol!*)

So, \$ is in FOLLOW(S) always.

In practice, we act as if the input string is always terminated with the special token \$. (This helps explain the fact that we include \$ in FOLLOW(A) iff A appears at the end of some sentential form.) Therefore, ...

Example

$$
\begin{aligned}
S &\rightarrow XSa \mid Yc \\
X &\rightarrow aY \mid YY \\
Y &\rightarrow bSa \mid cX \mid \epsilon
\end{aligned}
$$

$a \in \text{FOLLOW}(S)$ since $S \stackrel{*}{\Rightarrow} XSa$ $\qquad$ $c \notin \text{FOLLOW}(S)$

$b \notin \text{FOLLOW}(S)$ $\qquad$ $\$ \in \text{FOLLOW}(S)$ since $S \stackrel{*}{\Rightarrow} S$

$a, b, c \in \text{FOLLOW}(X)$

since $S \stackrel{*}{\Rightarrow} XXSaa$ and $\text{FIRST}(X) = \{a, b, c, \epsilon\}$

$\$ \notin \text{FOLLOW}(X)$

since every sentential form (except S) ends with a or c

$a, b, c \in \text{FOLLOW}(Y)$

since $S \stackrel{*}{\Rightarrow} aYXSaa$ and $\text{FIRST}(X) = \{a, b, c, \epsilon\}$

$\$ \notin \text{FOLLOW}(Y)$

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

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

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $\beta \stackrel{*}{\Rightarrow} \epsilon$, then everything in FOLLOW(A) is in FOLLOW(B).

We can also say this, as: FIRST(β) contains ϵ

Note, \$ is never in FIRST of anything. ε is never in FOLLOW of anything.
FOLLOW(S) always contains \$.
FIRST of something may contain ε, or may not contain.

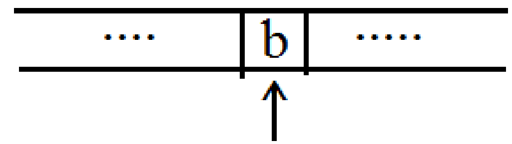
$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
\tag{4.28}
$$

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

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

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

Specification of parse table



Current input character

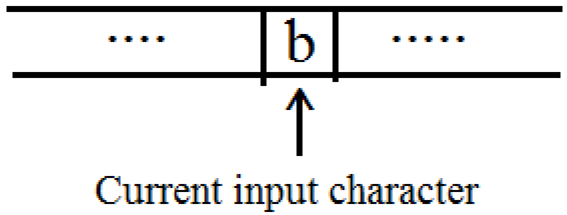
For leftmost variable A and current input token b, the *applicable productions* are

- all productions $A \to \alpha$ s.t. $b \in \mathrm{FIRST}(\alpha)$, and
- all productions $A \to \alpha$ s.t. $\epsilon \in \mathrm{FIRST}(\alpha)$ and $b \in \mathrm{FOLLOW}(A)$.

ϵ is in $\mathrm{FIRST}(\alpha)$, if $\alpha \overset{*}{\Rightarrow} \epsilon$.
That is, Entire sentential form α can vanish

For leftmost variable A and current input token \$, the *applicable productions* are

- all productions $A \to \alpha$ s.t. $\epsilon \in \mathrm{FIRST}(\alpha)$ and $\$ \in \mathrm{FOLLOW}(A)$.



More concise characterization of the applicable productions

For leftmost variable A and current input token $b \in \Sigma \cup \{\$\}$, the applicable productions are

- all productions $A \to \alpha$ s.t. $b \in \mathrm{FIRST}(\alpha)$, and
- all productions $A \to \alpha$ s.t. $\epsilon \in \mathrm{FIRST}(\alpha)$ and $b \in \mathrm{FOLLOW}(A)$.

Let's construct the parse table for the grammar we've been looking at.

$$\begin{aligned} S &\rightarrow XSa \mid Yc \\ X &\rightarrow aY \mid YY \\ Y &\rightarrow bSa \mid cX \mid \epsilon \end{aligned}$$

$\text{FIRST}(XSa) = \{a, b, c\}$

$\text{FIRST}(Yc) = \{b, c\}$

$\text{FIRST}(YY) = \{b, c, \epsilon\}$

$\text{FOLLOW}(S) = \{a, \$\}$

$\text{FOLLOW}(X) = \{a, b, c\}$

$\text{FOLLOW}(Y) = \{a, b, c\}$

LEFTMOST VARIABLE	CURRENT INPUT TOKEN			
	a	b	c	$\$$
S	$S \rightarrow XSa$	$S \rightarrow XSa \mid Yc$	$S \rightarrow XSa \mid Yc$	none
X	$X \rightarrow aY \mid YY$	$X \rightarrow YY$	$X \rightarrow YY$	none
Y	$Y \rightarrow \epsilon$	$Y \rightarrow bSa \mid \epsilon$	$Y \rightarrow cX \mid \epsilon$	none

LEFTMOST VARIABLE	CURRENT INPUT TOKEN			
	a	b	c	\$
S	$S \to XSa$	$S \to XSa \mid Yc$	$S \to XSa \mid Yc$	none
X	$X \to aY \mid YY$	$X \to YY$	$X \to YY$	none
Y	$Y \to \epsilon$	$Y \to bSa \mid \epsilon$	$Y \to cX \mid \epsilon$	none

Input string: *aca*

Note: There is no need to add ***S'→S\$*** *always. If we add this then an entry in the parse table with S' should exist!*

STACK	CURRENT INPUT	PRODUCTION TO APPLY
$S\$$	$aca\$$	$S \to XSa$
$XSa\$$	$aca\$$	$X \to aY$ (backtrack $X \to YY$)
$aYSa\$$	$aca\$$	match a
$YSa\$$	$ca\$$	$Y \to \epsilon$ (backtrack $Y \to cX$)
$Sa\$$	$ca\$$	$S \to Yc$ (backtrack $S \to XSa$)
$Yca\$$	$ca\$$	$Y \to \epsilon$ (backtrack $Y \to cX$)
$ca\$$	$ca\$$	match c
$a\$$	$a\$$	match a
$\$$	$\$$	successful parse

LL(1) grammars, for predictive parsing

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$. Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in $\text{FOLLOW}(A)$.

Definition A grammar is LL(1) if there is at most one production in the parsing table for each variable, token pair.

LL(1) grammars are without left recursion and are left factored.

It is a good practice to eliminate left recursion, and doing left factoring, before building the parse table.

Left Recursion and its elimination

- A grammar is left recursive, if for a variable A, there is a derivation $A \overset{+}{\Rightarrow} A\alpha$
- Top-down parsing can fall into infinite loop because of this
  - Hence these type of derivation should not be allowed.

Immediate left-recursion

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Replace these by the following

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

$$\begin{array}{lcl} E & \to & E + T \mid E - T \mid T \\ T & \to & T * F \mid T \,/\, F \mid F \\ F & \to & (\, E \,) \mid \mathbf{id} \end{array}$$

Can be replaced by

$$E \to TE'$$
$$E' \to +TE' \mid -TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to * FT' \mid /FT' \mid \epsilon$$
$$F \to (E) \mid id$$

But, several productions can lead to left-recursion

- $A_1 \rightarrow A_2\alpha; A_2 \rightarrow A_3\beta; A_3 \rightarrow A_1\gamma$

- Can give rise to $A_1 \Rightarrow\ A_1\gamma\beta\alpha$

- This happened because of the third production $A_3 \rightarrow A_1\gamma$
- If we order variables $A_1, A_2, \ldots,$
- A production $A_i \rightarrow A_j\gamma\ where\ j \leq i$ is problematic. So clean-up these!

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

Cyclic , if $A \overset{+}{\Rightarrow} A$
Remove unit productions it removes this too ☺

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 $A_i \to A_j\gamma$ by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions
7) }

Figure 4.11: Algorithm to eliminate left recursion from a grammar

$$A \longrightarrow BA$$
$$B \longrightarrow \epsilon$$

Can cause left recursion.

Consider the grammar

$$
\begin{aligned}
S &\rightarrow SX \mid SSb \mid XS \mid a \\
X &\rightarrow Xb \mid Sa \mid b
\end{aligned}
$$

Let's order the variables S, X:

The first time through we simply eliminate immediate left recursion in S-productions, yielding

$$
\begin{aligned}
S &\rightarrow XSS' \mid aS' \\
S' &\rightarrow XS' \mid SbS' \mid \epsilon \\
X &\rightarrow Xb \mid Sa \mid b
\end{aligned}
$$

So at this point we have grammar

$$\begin{aligned} S &\rightarrow XSS' \mid aS' \\ S' &\rightarrow XS' \mid SbS' \mid \epsilon \\ X &\rightarrow Xb \mid Sa \mid b \end{aligned}$$

and the next obligation is to replace the production

$$X \rightarrow Sa$$

with the productions

$$X \rightarrow XSS'a \mid aS'a\,.$$

We then eliminate immediate left recursion among

$$X \rightarrow XSS'a \mid aS'a \mid Xb \mid b\,.$$

Eliminating immediate left recursion among

$$X \to XSS'a \mid Xb \mid b \mid aS'a$$

yields

$$\begin{aligned} X &\to bX' \mid aS'aX' \\ X' &\to SS'aX' \mid bX' \mid \epsilon \end{aligned}$$

So the final result is

$$\begin{aligned} S &\to XSS' \mid aS' \\ S' &\to XS' \mid SbS' \mid \epsilon \\ X &\to bX' \mid aS'aX' \\ X' &\to SS'aX' \mid bX' \mid \epsilon \end{aligned}$$

Left Factoring

- Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule

$$A \longrightarrow \alpha\ \beta \mid \alpha\ \gamma$$

- Which one to use when we want to replace A?
- If wrong choice is made we need to backtrack.
- So, let us postpone the choice making moment.

– replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example 4.22: The following grammar abstracts the "dangling-else" problem:

$$\begin{aligned} S &\to i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E &\to b \end{aligned} \tag{4.23}$$

Here, i, t, and e stand for **if**, **then**, and **else**; E and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

$$\begin{aligned} S &\to i\ E\ t\ S\ S' \mid a \\ S' &\to e\ S \mid \epsilon \\ E &\to b \end{aligned} \tag{4.24}$$

$$
\begin{aligned}
S &\rightarrow AaS \mid b \\
A &\rightarrow c \mid d \mid B \\
B &\rightarrow AgC \mid AhC \mid DgC \mid DhC \\
C &\rightarrow c \mid d \mid D \\
D &\rightarrow eBf
\end{aligned}
$$

Is this grammar left-recursive?

Let's eliminate left recursion, with variable ordering S, A, B, C, D.

1. There's no immediate left recursion among S-productions.

2a. There are no productions from A whose rhs begins with S.

2b. There's no immediate left recursion among A-productions.

3a. There are no productions from B whose rhs begins with S.

3b. There are two productions from B whose rhs begins with A.

We replace

$$B \rightarrow AgC \mid AhC$$

with what?

After eliminating left recursion and after doing left factoring …

$$\begin{aligned} S &\rightarrow AaS \mid b \\ A &\rightarrow cB' \mid dB' \mid eCDB'fDB' \\ B' &\rightarrow DB' \mid \epsilon \\ C &\rightarrow c \mid d \mid eCDB'f \\ D &\rightarrow gC \mid hC \end{aligned}$$

$$\begin{aligned} S &\rightarrow AaS \mid b \\ A &\rightarrow cB \mid dB \mid eCDBfDB \\ B &\rightarrow DB \mid \epsilon \\ C &\rightarrow c \mid d \mid eCDBf \\ D &\rightarrow gC \mid hC \end{aligned}$$

$\mathrm{FIRST}(AaS) = \{c, d, e\}$

$\mathrm{FIRST}(DB) = \{g, h\}$

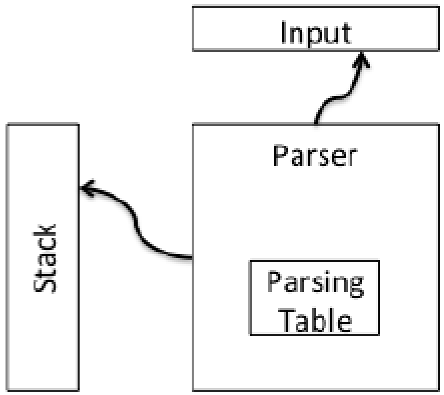
$\mathrm{FIRST}(\epsilon) = \{\epsilon\}$

$\mathrm{FOLLOW}(B) = \{a, f\}$

	CURRENT INPUT TOKEN								
VAR	a	b	c	d	e	f	g	h	$
S		b	AaS	AaS	AaS				
A			cB	dB	$eCDBfDB$				
B	ϵ					ϵ	DB	DB	
C			c	d	$eCDBf$				
D							gC	hC	

So, the grammar is LL(1)

LL(1) Parsing Algorithm




```
Initial configuration: Stack = S, Input = w$,
where, S = start symbol, $ = end of file marker
repeat {
   let X be the top stack symbol;
   let a be the next input symbol /*may be $*/;
   if X is a terminal symbol or $ then
        if X == a then {
             pop X from Stack;
             remove a from input;
        } else ERROR();
   else /* X is a non-terminal symbol */
         if M[X,a] == X → Y1Y2... Yk then {
               pop X from Stack;
               push Yk, Yk-1, ..., Y1 onto Stack;
                      (Y1 on top)
         }
} until Stack has emptied;
```

LL(1) Parsing Algorithm Example

Grammar

$S' \rightarrow S\$$

$S \rightarrow aAS \mid c$

$A \rightarrow ba \mid SB$

$B \rightarrow bA \mid S$

string: *acbbac*$

LL(1) Parsing Table

	a	b	c	$
S′	S′ → S$		S′ → S$	
S	S → aAS		S → c	
A	A → SB	A → ba	A → SB	
B	B → S	B → bA	B → S