

Syntax Analyzer -Top-Down Parsing

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
 - Recursive-Descent Parsing
 - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
 - It is a general parsing technique, but not widely used.
 - Not efficient
 - Predictive Parsing
 - no backtracking
 - efficient
 - needs a special form of grammars (LL(1) grammars).
 - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
 - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

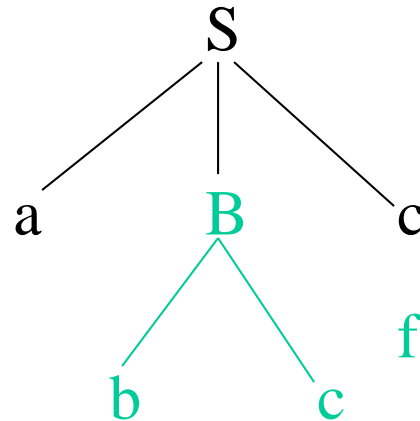
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

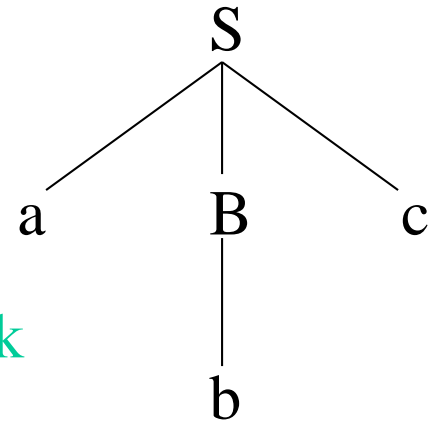
$S \rightarrow aBc$

$B \rightarrow bc \mid b$

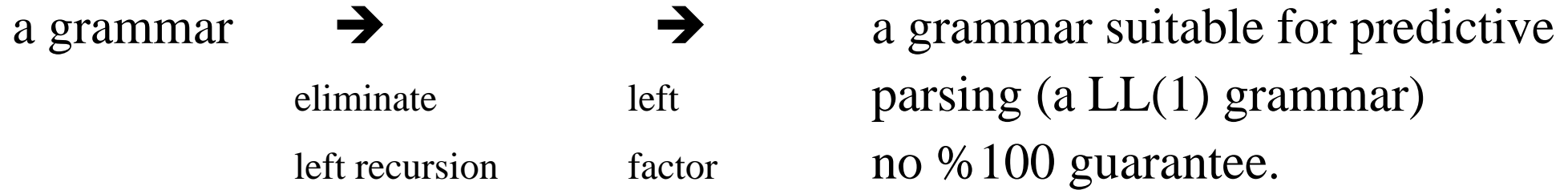
input: abc



fails, backtrack



Predictive Parser



- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$

input: ... a

↑
current token

Predictive Parser (example)

```
stmt → if ..... |  
      while ..... |  
      begin ..... |  
      for .....
```

- When we are trying to write the non-terminal *stmt*, if the current token is *if* we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {  
    - match the current token with a, and move to the next token;  
    - call 'B';  
    - match the current token with b, and move to the next token;  
}
```

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {  
  case of the current token {  
    'a': - match the current token with a, and move to the next token;  
        - call 'B';  
        - match the current token with b, and move to the next token;  
    'b': - match the current token with b, and move to the next token;  
        - call 'A';  
        - call 'B';  
  }  
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε -productions.

$$A \rightarrow aA \mid bB \mid \varepsilon$$

- If all other productions fail, we should apply an ε -production. For example, if the current token is not a or b, we may apply the ε -production.
- Most correct choice: We should apply an ε -production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

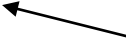
Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$


$C \rightarrow f$

```
proc A {  
  case of the current token {  
    a: - match the current token with a,  
        and move to the next token;  
        - call B;  
        - match the current token with e,  
        and move to the next token;  
    c: - match the current token with c,  
        and move to the next token;  
        - call B;  
        - match the current token with d,  
        and move to the next token;  
    f: - call C  
  }  
}
```

 **first set of C**

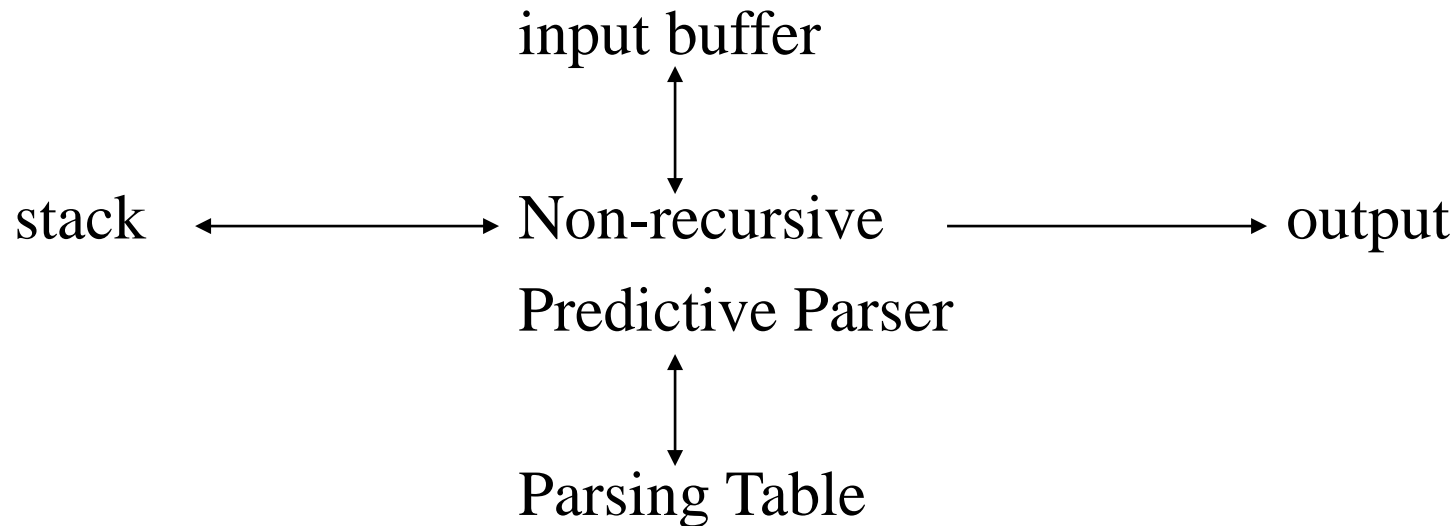
```
proc C {  match the current token with f,  
          and move to the next token; }
```

```
proc B {  
  case of the current token {  
    b: - match the current token with b,  
        and move to the next token;  
        - call B  
    e,d: do nothing  
  }  
}
```

 **follow set of B**

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol \$.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol \$.
- initially the stack contains only the symbol \$ and the starting symbol S. $\$S \leftarrow$ initial stack
- when the stack is emptied (ie. only \$ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array $M[A,a]$
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol \$
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.
 1. If X and a are $\$$ \rightarrow parser halts (successful completion)
 2. If X and a are the same terminal symbol (different from $\$$)
 \rightarrow parser pops X from the stack, and moves the next symbol in the input buffer.
 3. If X is a non-terminal
 \rightarrow parser looks at the parsing table entry $M[X,a]$. If $M[X,a]$ holds a production rule $X \rightarrow Y_1 Y_2 \dots Y_k$, it pops X from the stack and pushes Y_k, Y_{k-1}, \dots, Y_1 into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \dots Y_k$ to represent a step of the derivation.
 4. none of the above \rightarrow error
 - all empty entries in the parsing table are errors.
 - If X is a terminal symbol different from a , this is also an error case.

LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

	a	b	\$
S	$S \rightarrow aBa$		
B	$B \rightarrow \epsilon$	$B \rightarrow bB$	

LL(1) Parsing
Table

stack

\$S
\$aBa
\$aB
\$aBb
\$aB
\$aBb
\$aB
\$a
\$

input

abba\$
abba\$
bba\$
bba\$
ba\$
ba\$
a\$
a\$
\$

output

$S \rightarrow aBa$

 $B \rightarrow bB$

 $B \rightarrow bB$

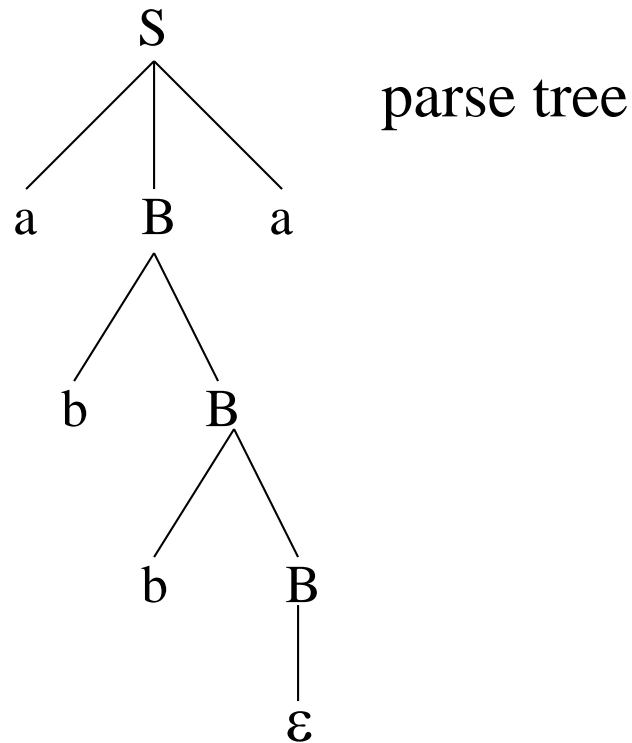
 $B \rightarrow \epsilon$

accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



LL(1) Parser – Example2

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

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

LL(1) Parser – Example2

<u>stack</u>	<u>input</u>	<u>output</u>
\$E	id+id\$	$E \rightarrow TE'$
\$E'T	id+id\$	$T \rightarrow FT'$
\$E'T'F	id+id\$	$F \rightarrow id$
\$E'T'id	id+id\$	
\$E'T'	+id\$	$T' \rightarrow \varepsilon$
\$E'	+id\$	$E' \rightarrow +TE'$
\$E'T+	+id\$	
\$E'T	id\$	$T \rightarrow FT'$
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	
\$E'T'	\$	$T' \rightarrow \varepsilon$
\$E'	\$	$E' \rightarrow \varepsilon$
\$	\$	accept

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
 - FIRST FOLLOW
- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ϵ , then ϵ is also in FIRST(α) .
- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal* A in the strings derived from the starting symbol.
 - a terminal a is in FOLLOW(A) if $S \xRightarrow{*} \alpha A a \beta$
 - \$ is in FOLLOW(A) if $S \xRightarrow{*} \alpha A$

Compute FIRST for Any String X

- If X is a terminal symbol \rightarrow $\text{FIRST}(X) = \{X\}$
- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule
 \rightarrow ε is in $\text{FIRST}(X)$.
- If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 \dots Y_n$ is a production rule
 \rightarrow if a terminal **a** in $\text{FIRST}(Y_i)$ and ε is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then **a** is in $\text{FIRST}(X)$.
 \rightarrow if ε is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then ε is in $\text{FIRST}(X)$.
- If X is ε \rightarrow $\text{FIRST}(X) = \{\varepsilon\}$
- If X is $Y_1 Y_2 \dots Y_n$
 \rightarrow if a terminal **a** in $\text{FIRST}(Y_i)$ and ε is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, i-1$ then **a** is in $\text{FIRST}(X)$.
 \rightarrow if ε is in all $\text{FIRST}(Y_j)$ for $j=1, \dots, n$ then ε is in $\text{FIRST}(X)$.

FIRST Example

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \text{id}$$
$$\text{FIRST}(F) = \{ (, \text{id} \}$$
$$\text{FIRST}(T') = \{ *, \varepsilon \}$$
$$\text{FIRST}(T) = \{ (, \text{id} \}$$
$$\text{FIRST}(E') = \{ +, \varepsilon \}$$
$$\text{FIRST}(E) = \{ (, \text{id} \}$$
$$\text{FIRST}(TE') = \{ (, \text{id} \}$$
$$\text{FIRST}(+TE') = \{ + \}$$
$$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$$
$$\text{FIRST}(FT') = \{ (, \text{id} \}$$
$$\text{FIRST}(*FT') = \{ * \}$$
$$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$$
$$\text{FIRST}((E)) = \{ (\}$$
$$\text{FIRST}(\text{id}) = \{ \text{id} \}$$

Compute FOLLOW (for non-terminals)

- If S is the start symbol \rightarrow $\$$ is in $\text{FOLLOW}(S)$
- if $A \rightarrow \alpha B \beta$ is a production rule
 \rightarrow everything in $\text{FIRST}(\beta)$ is $\text{FOLLOW}(B)$ except ϵ
- If ($A \rightarrow \alpha B$ is a production rule) or
($A \rightarrow \alpha B \beta$ is a production rule and ϵ is in $\text{FIRST}(\beta)$)
 \rightarrow everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

We apply these rules until nothing more can be added to any follow set.

FOLLOW Example

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid \text{id}$$

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

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

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

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

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

Constructing LL(1) Parsing Table -- Algorithm

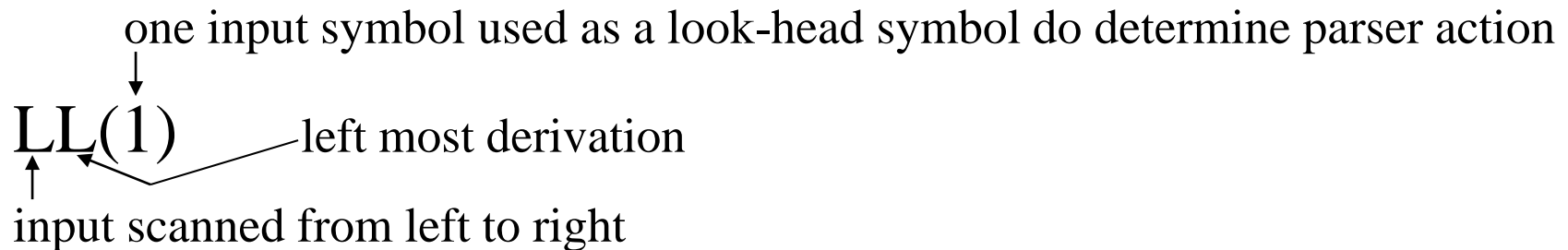
- for each production rule $A \rightarrow \alpha$ of a grammar G
 - for each terminal a in $\text{FIRST}(\alpha)$
 - ➔ add $A \rightarrow \alpha$ to $M[A,a]$
 - If ϵ in $\text{FIRST}(\alpha)$
 - ➔ for each terminal a in $\text{FOLLOW}(A)$ add $A \rightarrow \alpha$ to $M[A,a]$
 - If ϵ in $\text{FIRST}(\alpha)$ and $\$$ in $\text{FOLLOW}(A)$
 - ➔ add $A \rightarrow \alpha$ to $M[A,\$]$
- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$	$\text{FIRST}(TE') = \{ (, \text{id} \}$	$\rightarrow E \rightarrow TE' \text{ into } M[E, (] \text{ and } M[E, \text{id}]$
$E' \rightarrow +TE'$	$\text{FIRST}(+TE') = \{ + \}$	$\rightarrow E' \rightarrow +TE' \text{ into } M[E', +]$
$E' \rightarrow \varepsilon$	$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$ but since ε in $\text{FIRST}(\varepsilon)$ and $\text{FOLLOW}(E') = \{ \$,) \}$	\rightarrow none $\rightarrow E' \rightarrow \varepsilon \text{ into } M[E', \$] \text{ and } M[E',)]$
$T \rightarrow FT'$	$\text{FIRST}(FT') = \{ (, \text{id} \}$	$\rightarrow T \rightarrow FT' \text{ into } M[T, (] \text{ and } M[T, \text{id}]$
$T' \rightarrow *FT'$	$\text{FIRST}(*FT') = \{ * \}$	$\rightarrow T' \rightarrow *FT' \text{ into } M[T', *]$
$T' \rightarrow \varepsilon$	$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$ but since ε in $\text{FIRST}(\varepsilon)$ and $\text{FOLLOW}(T') = \{ \$,), + \}$	\rightarrow none $\rightarrow T' \rightarrow \varepsilon \text{ into } M[T', \$], M[T',)]$ and $M[T', +]$
$F \rightarrow (E)$	$\text{FIRST}((E)) = \{ (\}$	$\rightarrow F \rightarrow (E) \text{ into } M[F, (]$
$F \rightarrow \text{id}$	$\text{FIRST}(\text{id}) = \{ \text{id} \}$	$\rightarrow F \rightarrow \text{id} \text{ into } M[F, \text{id}]$

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.



- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

$S \rightarrow i C t S E \mid a$

$E \rightarrow e S \mid \varepsilon$

$C \rightarrow b$

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

$\text{FOLLOW}(E) = \{ \$, e \}$

$\text{FOLLOW}(C) = \{ t \}$

$\text{FIRST}(iCtSE) = \{ i \}$

$\text{FIRST}(a) = \{ a \}$

$\text{FIRST}(eS) = \{ e \}$

$\text{FIRST}(\varepsilon) = \{ \varepsilon \}$

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

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for $M[E,e]$

Problem → ambiguity

A Grammar which is not LL(1) (cont.)

- What do we have to do if the resulting parsing table contains multiply defined entries?
 - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
 - If the grammar is not left factored, we have to left factor the grammar.
 - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
 - $A \rightarrow A\alpha \mid \beta$
 - ➔ any terminal that appears in $\text{FIRST}(\beta)$ also appears in $\text{FIRST}(A\alpha)$ because $A\alpha \Rightarrow \beta\alpha$.
 - ➔ If β is ϵ , any terminal that appears in $\text{FIRST}(\alpha)$ also appears in $\text{FIRST}(A\alpha)$ and $\text{FOLLOW}(A)$.
- A grammar is not left factored, it cannot be a LL(1) grammar
 - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
 - ➔ any terminal that appears in $\text{FIRST}(\alpha\beta_1)$ also appears in $\text{FIRST}(\alpha\beta_2)$.
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
 1. Both α and β cannot derive strings starting with same terminals.
 2. At most one of α and β can derive to ϵ .
 3. If β can derive to ϵ , then α cannot derive to any string starting with a terminal in $\text{FOLLOW}(A)$.

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
 - if the terminal symbol on the top of stack does not match with the current input symbol.
 - if the top of stack is a non-terminal A , the current input symbol is a , and the parsing table entry $M[A,a]$ is empty.
- What should the parser do in an error case?
 - The parser should be able to give an error message (as much as possible meaningful error message).
 - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- **Panic-Mode Error Recovery**
 - Skipping the input symbols until a synchronizing token is found.
- **Phrase-Level Error Recovery**
 - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- **Error-Productions**
 - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
 - When an error production is used by the parser, we can generate appropriate error diagnostics.
 - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- **Global-Correction**
 - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
 - We have to globally analyze the input to find the error.
 - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
 - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
 - All the empty entries are marked as *synch* to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
 - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery - Example

$S \rightarrow AbS \mid e \mid \varepsilon$

$A \rightarrow a \mid cAd$

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

$\text{FOLLOW}(A) = \{b, d\}$

	a	b	c	d	e	\$
S	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	<i>sync</i>	$A \rightarrow cAd$	<i>sync</i>	<i>sync</i>	<i>sync</i>

<u>stack</u>	<u>input</u>	<u>output</u>
\$S	aab\$	$S \rightarrow AbS$
\$SbA	aab\$	$A \rightarrow a$
\$Sba	aab\$	
\$Sb	ab\$	Error: missing b, inserted
\$S	ab\$	$S \rightarrow AbS$
\$SbA	ab\$	$A \rightarrow a$
\$Sba	ab\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \varepsilon$
\$	\$	accept

<u>stack</u>	<u>input</u>	<u>output</u>
\$S	ceadb\$	$S \rightarrow AbS$
\$SbA	ceadb\$	$A \rightarrow cAd$
\$SbdAc	ceadb\$	
\$SbdA	eadb\$	Error: unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
\$Sbd	db\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \varepsilon$
\$	\$	accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care of that error case.
- These error routines may:
 - change, insert, or delete input symbols.
 - issue appropriate error messages
 - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.