

CS202 – System Software

Dr. Manish Khare

Lecture 10-11



Parsing



➤ Two Types of parsing

- Top-down parsing
- Bottom-up parsing

➤ These terms refers to the order in which nodes in the parse tree are considered.

Top-Down parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth first).
- Equivalently, top -down parsing can be viewed as finding finding a leftmost derivation for an input string.

Example

➤ Consider the grammar

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid \text{id}$

➤ Construct parse tree for the input string $\text{id}+\text{id}*\text{id}$

➤ First we check left-recursion in grammar, because this grammar is left recursive, so we will remove left-recursion.

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid \text{id}$

➤ Now we construct parse tree

Example

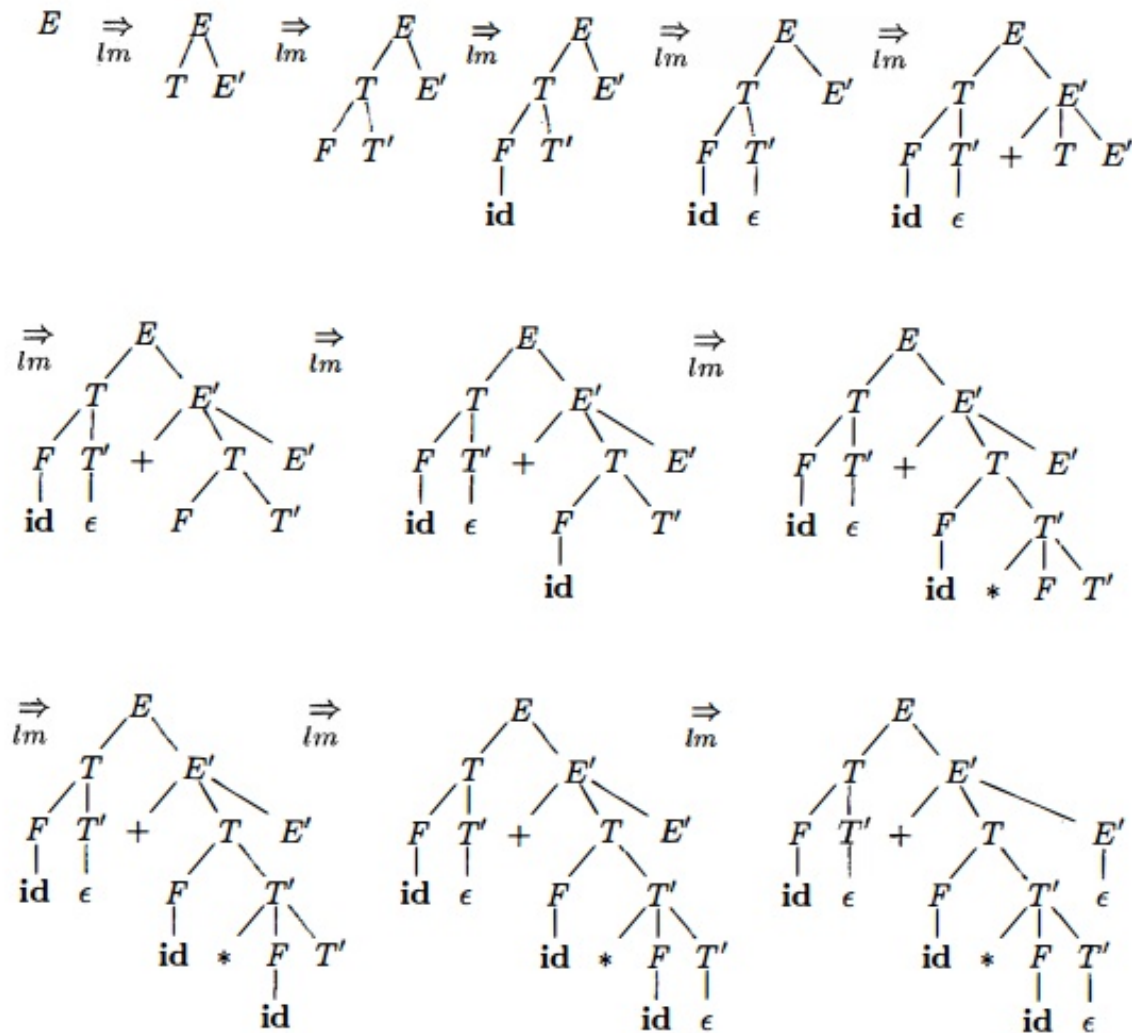


Figure 4.12: Top-down parse for `id + id * id`

Types of Top-Down Parser

- Mainly two types of top-down parser
- Recursive predictive parser
 - This type of parser may require backtracking to find the correct A-production to be applied.
- Non -Recursive predictive parser
 - In this type of parser no backtracking is required. Here predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look only at one.
 - LL(k) is the example of non-recursive predictive parser, where k is the symbol ahead in the input. So best example is LL(1)parser.

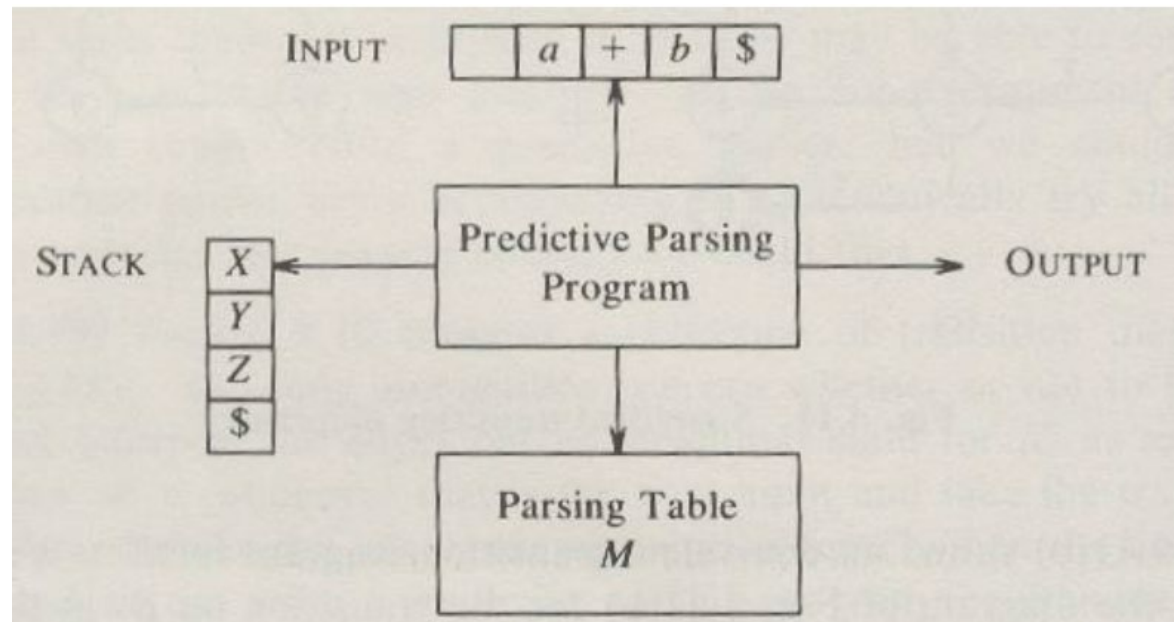
Predictive LL(1) parser

- A predictive parser is an efficient way of implementing recursive-decent parsing since a stack is maintained in predictive parsing for handling the activation records.
- In top down predictive parsers, the grammar will be able to predict the right alternative for the expansion of non-terminal during the parsing process, and hence, hence, it need no backtrack.
- For LL(1) – the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for input string and the number 1 in the input symbol means it uses only one input symbol (look ahead) to predict the parsing.

Predictive LL(1) parser


➤ The predictive parser has following terms

- Input buffer
- Stack
- Parsing table
- Input stream



Predictive LL(1) parser

- The input buffer contains the strings to be parsed, followed by \$, a symbol used as a right end marker to indicate the end of the input string.
- The stack contains a sequence of grammar symbols with \$ on the bottom, indicating the bottom of a stack. Initially the stack contains the start symbol of the grammars on top of \$.
- The parsing table is a 2-D array $M[A,a]$, where A is a non-terminal and a is a terminal on symbol \$. The parser is controlled by a program that behaves as follows
 - The program consider X , the symbol on top of the stack, and a the current input symbol. These two symbol determine the action of the parser.
 - There are three possibilities

- 
-
- 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 advances the input pointer to the next input symbol .
 - 3. if X is a non-terminal, the program consults entry $M[X,a]$ of the parsing table M , this entry will be either an X -production of the grammar or error entry.

Construction of Predictive Parser LL(1) Table

- The construction of a predictive parser is aided by two functions associated with a grammar G . these functions are **FIRST** and **FOLLOW**, allow us to fill in the entries of a predictive parsing table for G , whenever possible.

Construction of Predictive Parser LL(1) Table

FIRST Computations

- We define a function $\text{FIRST}(X)$, where X is in $(V \cup \Sigma)^*$ as follows:
 - $\text{FIRST}(X)$ is the set of terminal symbols that are first symbols appearing at R.H.S. in derivation of X .
- To compute $\text{FIRST}(X)$ for all grammar symbol X , apply rule, given in next slide, until no more terminals on ϵ can be added to any FIRST set.

Construction of Predictive Parser LL(1) Table

FIRST Computations

- 1. if X is a terminal, then $\text{FIRST}(X) = \{X\}$
- 2. if X is a non-terminal, and $X \rightarrow \epsilon\alpha$ is a production, then add ϵ to $\text{FIRST}(X)$. i.e. $\text{FIRST}(X) = \{\epsilon\}$
- 3. if $X \rightarrow aA$, where a is terminal, then $\text{FIRST}(X) = \{a\}$
- 3. If there is a Production $X \rightarrow Y_1, Y_2, \dots, Y_k$ **then** add $\text{FIRST}(Y_1, Y_2, \dots, Y_k)$ to $\text{FIRST}(X)$
- 4. $\text{FIRST}(Y_1, Y_2, \dots, Y_k)$ is **either**

- $\text{FIRST}(Y_1)$ (if $\text{First}(Y_1)$ doesn't contain ϵ)

OR

(if $\text{FIRST}(Y_1)$ does contain ϵ) then $\text{FIRST}(Y_1, Y_2, \dots, Y_k)$ is everything in $\text{FIRST}(Y_1)$ <except for ϵ > as well as everything in $\text{FIRST}(Y_2, \dots, Y_k)$

- If $\text{FIRST}(Y_1) \text{ FIRST}(Y_2) \dots \text{FIRST}(Y_k)$ all contain ϵ **then** add ϵ to $\text{FIRST}(Y_1, Y_2, \dots, Y_k)$ as well.

Construction of Predictive Parser LL(1) Table

FOLLOW Computation

- We define a function FOLLOW(x), where A is a non-terminal as follow:
 - FOLLOW(X) is a set of terminals that immediately follow A in any string occurring on the right side of productions of the grammar.

- To compute FOLLOW(X) for all non-terminal A, apply rule given in next slide, until nothing can be added to any FOLLOW set.

Construction of Predictive Parser LL(1) Table

FOLLOW Computation

- 1. Place \$ (the end of input marker) in Follow(S), where S is the start symbol.
- 2. If there is a production $A \rightarrow aBb$, (where a can be a whole string) **then** everything in FIRST(b) except for ϵ is placed in FOLLOW(B).
- 3. If there is a production $A \rightarrow aB$, **then** everything in FOLLOW(A) is in FOLLOW(B).
- 4. If there is a production $A \rightarrow aBb$, where FIRST(b) contains ϵ , **then** everything in FOLLOW(A) is in FOLLOW(B).

Exercise

➤ Consider the grammar

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid \text{id}$

➤ First we check left-recursion in grammar, because this grammar is left recursive, so we will remove left-recursion.

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid \text{id}$

➤ Now we Compute First and Follow

Exrercise

➤ Its FIRST is

- $\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$
- $\text{FIRST}(E') = \{ +, \epsilon \}$
- $\text{FIRST}(T') = \{ *, \epsilon \}$

➤ Its FOLLOW is

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

Exercise

➤ Calculate the first and follow functions for the given grammar-

- $S \rightarrow aBDh$
- $B \rightarrow cC$
- $C \rightarrow bC / \epsilon$
- $D \rightarrow EF$
- $E \rightarrow g / \epsilon$
- $F \rightarrow f / \epsilon$



➤ First Functions-

- $\text{First}(S) = \{ a \}$
- $\text{First}(B) = \{ c \}$
- $\text{First}(C) = \{ b, \epsilon \}$
- $\text{First}(D) = \{ \text{First}(E) - \epsilon \} \cup \text{First}(F) = \{ g, f, \epsilon \}$
- $\text{First}(E) = \{ g, \epsilon \}$
- $\text{First}(F) = \{ f, \epsilon \}$




➤ Follow Functions-

- $\text{Follow}(S) = \{ \$ \}$
- $\text{Follow}(B) = \{ \text{First}(D) - \epsilon \} \cup \text{First}(h) = \{ g, f, h \}$
- $\text{Follow}(C) = \text{Follow}(B) = \{ g, f, h \}$
- $\text{Follow}(D) = \text{First}(h) = \{ h \}$
- $\text{Follow}(E) = \{ \text{First}(F) - \epsilon \} \cup \text{Follow}(D) = \{ f, h \}$
- $\text{Follow}(F) = \text{Follow}(D) = \{ h \}$



➤ Calculate the first and follow functions for the given grammar-

- $S \rightarrow A$
- $A \rightarrow aB / Ad$
- $B \rightarrow b$
- $C \rightarrow g$

- 
- The given grammar is left recursive.
 - So, we first remove left recursion from the given grammar.
 - After eliminating left recursion, we get the following grammar-
 - $S \rightarrow A$
 - $A \rightarrow aBA'$
 - $A' \rightarrow dA' / \epsilon$
 - $B \rightarrow b$
 - $C \rightarrow g$



➤ First Functions-

- $\text{First}(S) = \text{First}(A) = \{ a \}$
- $\text{First}(A) = \{ a \}$
- $\text{First}(A') = \{ d, \epsilon \}$
- $\text{First}(B) = \{ b \}$
- $\text{First}(C) = \{ g \}$



➤ Follow Functions-

- $\text{Follow}(S) = \{ \$ \}$
- $\text{Follow}(A) = \text{Follow}(S) = \{ \$ \}$
- $\text{Follow}(A') = \text{Follow}(A) = \{ \$ \}$
- $\text{Follow}(B) = \{ \text{First}(A') - \epsilon \} \cup \text{Follow}(A) = \{ d, \$ \}$
- $\text{Follow}(C) = \text{NA}$

Construction of Predictive Parse Table

-
- After finding the FIRST and FOLLOW, we can construct predictive parse table $M[A, a]$, where A_i^s are non-terminal of the given grammar and a^s are terminals including \$. The entries in this two-dimensional table are made according to following rule.
- Here 'A' represents row and 'a' represents columns.

Construction of Predictive Parse Table

Algorithm for construction of Predictive Parse Table

1. Compare each production of the grammar with $A \rightarrow \alpha$, i.e. everything on right side is taken as α , apply step 2 and 3, for each of them

2. Find $\text{FIRST}(\alpha)$, for each a in $\text{FIRST}(\alpha)$

$$\text{add } M[A, a] = A \rightarrow \alpha$$

which means for each a in $\text{FIRST}(\alpha)$, corresponding to non-terminal A and terminal a , the current production $A \rightarrow \alpha$ is added to table.

3. If ϵ is in $\text{FIRST}(\alpha)$, then

$$\text{add, } M[A, a] = A \rightarrow \alpha \text{ for each } b \text{ in } \text{FOLLOW}(A)$$

If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$ then

$$\text{add, } M[A, \$] = A \rightarrow \alpha$$

4. Make each undefined entry of M , as error.

Construction of Predictive Parse Table

➤ Consider the grammar

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid \text{id}$

➤ First we check left-recursion in grammar, because this grammar is left recursive, so we will remove left-recursion.

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid \mathbf{id}$

Construction of Predictive Parse Table

➤ Its FIRST is

- $\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$
- $\text{FIRST}(E') = \{ +, \epsilon \}$
- $\text{FIRST}(T') = \{ *, \epsilon \}$

➤ Its FOLLOW is

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

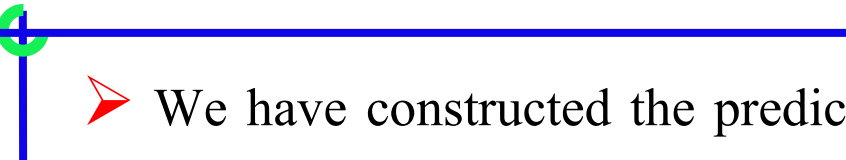
➤ Now for constructing predictive parse table, comparing each production with $A \rightarrow \alpha$

➤ Entries for Predictive Parser table

- $M[E,()] = E \rightarrow TE'$, $M[E, id] = E \rightarrow TE'$
- $M[E',+] = E' \rightarrow +TE'$, $M[E',)] = E' \rightarrow \epsilon$, $M[E',\$] = E' \rightarrow \epsilon$
- $M[T,()] = E \rightarrow FT'$, $M[T, id] = E \rightarrow FT'$
- $M[T',*] = T' \rightarrow *FT'$, $M[T',+] = T' \rightarrow \epsilon$, $M[T',)] = T' \rightarrow \epsilon$, $M[T',\$] = T' \rightarrow \epsilon$
- $M[F,()] = F \rightarrow (E)$, $M[F, id] = F \rightarrow id$

NONTER-MINAL	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	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

- 
- We have constructed the predictive parse table, now our next step is how to use this table to declare that an input string has been accepted by the grammar or not, for this first we see one algorithm for this purpose.

Algo. for predictive parsing

Push start symbol into stack

Repeat

Begin

Make X to be the top stack symbol and \underline{a} the next symbol

If X is a terminal on $\$$ then

If $X=a$, then

Pop X from stack and remove \underline{a} from input

else

Error()

else /* i.e. X is a non-terminal */

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

Begin

Pop X from stack

Push $Y_k, Y_{k-1}, \dots, Y_3, Y_2, Y_1$ onto stack
such that Y_1 is on top

End

Else

error()

End

Until $X = \$$ /* i.e. stack empties */

Predictive Parsing

- This algorithm is quite simple. First push start symbol onto top of stack, call it X . now look for the given input string, in the If condition, if they match Pop X and remove current input a .
- If it is a non-terminal then Pop this non-terminal from stack and look for production corresponding to this non-terminal X and current input symbol a i.e. $M[X,a]$, then after popping X , push the right hand side of production in reverse order into stack ($Y_k Y_{k-1} \text{ ----- } Y_3 Y_2 Y_1$). Repeat this process till $X = \$$. Note that we always call the top stack symbol as X .
- Now we will see this parsing algorithm for the input string $id+id*id$

Predictive Parsing

Stack	Input (a)	Output
\$E	id+id*id\$	--
\$E/T	id+id*id\$	$E \rightarrow TE'$ from $M[E, id]$ - [Push in reverse order]
\$E/T/F	id+id*id\$	$T \rightarrow FT'$ from $M[T, id]$ - [Push in reverse order]
\$E/T/id	id+id*id\$	$F \rightarrow id$ from $M[F, id]$ [Push]
\$E/T/	+id*id\$	$X=id$, and matches with $a=id$ So, Pop X and remove a
\$E/	+id*id\$	$T' \rightarrow \varepsilon$ from $M[T', +]$ [Push]
\$E/T+	+id*id\$	$E' \rightarrow +TE'$ from $M[E', +]$ [Push in reverse order]
\$E/T	id*id\$	$X=+$, and matches with $a=+$ So, Pop X and remove a
\$E/T/F	id*id\$	$E \rightarrow FT'$ from $M[T, id]$ [Push in reverse order]
\$E/T/id	id*id\$	$F \rightarrow id$ from $M[F, id]$ [Push]
\$E/T/	*id\$	$X=id$, and matches with $a=id$ So, Pop X and remove a

Predictive Parsing

Stack	Input (a)	Output
\$E/T'	*id\$	X=id, and matches with a=id So, Pop X and remove a
\$E/T'F*	*id\$	$T' \rightarrow *FT'$ from $M[T',*]$ [Push in reverse order]
\$E/T'F	id\$	X=*, and matches with a=* So, Pop X and remove a
\$E/T'id	id\$	$F \rightarrow id$ from $M[F, id]$ [Push]
\$E/T'	\$	X=id, and matches with a=id So, Pop X and remove a
\$E/	\$	$T' \rightarrow \epsilon$ from $M[T',\$]$ [Push]
\$	\$	$E' \rightarrow \epsilon$ from $M[E',\$]$ [Push]

➤ So, \$ matches and the input string “id+id*id” is accepted by the grammar.

LL(1) Grammar

- If entry $M[A, a]$ contains multiple entries in any cell then that grammar is not LL(1).
- It can also show that a grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct production of G of following condition hold
 - (i). For no terminal 'a' do α and β derives the string's beginning with 'a'.
 - (ii). At most one of α and β can derive the empty string.
 - (iii). If $\beta \rightarrow \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A)

Example

➤ Consider the grammar

- $S \rightarrow iCtSS' \mid a$
- $S' \rightarrow eS \mid \varepsilon$
- $C \rightarrow b$

➤ First Compute FIRST and FOLLOW

- $\text{FIRST}(S) = \{i, a\}$, $\text{FIRST}(S') = \{e\}$, $\text{FIRST}(C) = \{b\}$
- $\text{FOLLOW}(S) = \text{FOLLOW}(S') = \{e, \$\}$, $\text{FOLLOW}(C) = \{t\}$

➤ Predictive Parsing Table Entry Calculation

- | | |
|--|---|
| ■ $M[S, i] = S \rightarrow iCtSS'$ | $M[S, a] = S \rightarrow a$ |
| ■ $M[S', e] = S' \rightarrow eS$ | $M[S', \varepsilon] = S' \rightarrow \varepsilon$ |
| ■ $M[S', \$] = S' \rightarrow \varepsilon$ | $M[C, b] = C \rightarrow b$ |

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iCtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

➤ Here entry $M[S',e]$ contains multiple entry, so this grammar is not LL(1).