

EC3204: Programming Languages and Compilers

Lecture 6 — Syntax Analysis (2): *Top-down Parsing*

Sunbeom So
Fall 2024

Top-Down Parsing

- Parsing is a process of constructing a parse tree of a given input string.
- Top-down parsing begins with the root of the parse tree, and extends the tree downward in preorder (root-left-right), until leaves match the input string.
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Rewriting Grammars for Top-Down Parsing

Unfortunately, not all grammars can be parsed by top-down parsing algorithms. To enable top-down parsing, we should first rewrite the grammar. Two representative transformations are:

- Eliminating ambiguity
- Eliminating left-recursion

Eliminating Ambiguity

Consider the grammar¹:

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

The grammar is ambiguous because it permits multiple parse trees, e.g.,

- $1 + 2 * 3$: $(1 + 2) * 3$ vs. $1 + (2 * 3)$
- $1 + 2 + 3$: $(1 + 2) + 3$ vs. $1 + (2 + 3)$

To eliminate the ambiguity, we should impose:

- (precedence) bind $*$ tighter than $+$
 - ▶ $1 + 2 * 3$ is always parsed by $1 + (2 * 3)$
- (associativity) make $*$ and $+$ left-associative
 - ▶ $1 + 2 + 3$ is always parsed by $(1 + 2) + 3$

Q. How to remove the ambiguity?

¹In this course, we rely on P to define $G = (V, T, S, P)$, if the first three are clear.

General facts about Ambiguity

We hope to transform ambiguous grammars into unambiguous ones.

However,

- There is no algorithm to remove ambiguity from a CFG.
- There is no algorithm that can even tell us whether a CFG is ambiguous or not.
- There are context-free languages that are inherently ambiguous, i.e., there are context-free languages for which removing the ambiguity is impossible.

In practice, we can manually design an unambiguous grammar.

- Our original ambiguous grammar:

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

- An equivalent unambiguous grammar:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F \quad /* \text{introduce terms } (T) \text{ to prefer } * */$$

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

Exercises

- Draw a parse tree for $\text{id} + \text{id} + \text{id}$.
- Draw a parse tree for $\text{id} + \text{id} * \text{id}$.

- Transform the grammar

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

so that $*$ associate to the right.

Exercises

- Draw a parse tree for $\text{id} + \text{id} + \text{id}$.

$$E \Rightarrow_l E + T \Rightarrow_l E + T + T \Rightarrow_l T + T + T \Rightarrow_l \dots \Rightarrow_l \text{id} + \text{id} + \text{id}$$

- Draw a parse tree for $\text{id} + \text{id} * \text{id}$.

$$\begin{aligned} E &\Rightarrow_l E + T \Rightarrow_l T + T \Rightarrow_l F + T \Rightarrow_l \text{id} + T \\ &\Rightarrow_l \text{id} + (T * F) \Rightarrow_l \text{id} + (F * F) \Rightarrow_l \dots \Rightarrow_l \text{id} + \text{id} * \text{id} \end{aligned}$$

- Transform the grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \text{id} \mid (E) \end{aligned}$$

so that $*$ associate to the right.

Replace $T * F$ with $F * T$.

Eliminating Left-Recursion

- A grammar is **left-recursive** if it has a non-terminal A such that there A appears as the first right-hand-side symbol in the production of A . For example, the grammar below is left-recursive.

$$E \rightarrow E + T \mid T$$

Eliminating Left-Recursion

- A grammar is **left-recursive** if it has a non-terminal A such that there A appears as the first right-hand-side symbol in the production of A . For example, the grammar below is left-recursive.

$$E \rightarrow E + T \mid T$$

- The left-recursive grammars are not suitable for top-down parsing; we may go into an infinite loop. For example, to parse T , we may apply $E \rightarrow E + T$ forever.

Eliminating Left-Recursion

- A grammar is **left-recursive** if it has a non-terminal A such that there A appears as the first right-hand-side symbol in the production of A . For example, the grammar below is left-recursive.

$$E \rightarrow E + T \mid T$$

- The left-recursive grammars are not suitable for top-down parsing; we may go into an infinite loop. For example, to parse T , we may apply $E \rightarrow E + T$ forever.
- We can remove left-recursion using transformation rules that produce non-left-recursive productions. For example, we can transform the left-recursive production $A \rightarrow A\alpha \mid \beta$ into

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

Eliminating Left-Recursion

- A grammar is **left-recursive** if it has a non-terminal A such that there A appears as the first right-hand-side symbol in the production of A . For example, the grammar below is left-recursive.

$$E \rightarrow E + T \mid T$$

- The left-recursive grammars are not suitable for top-down parsing; we may go into an infinite loop. For example, to parse T , we may apply $E \rightarrow E + T$ forever.
- We can remove left-recursion using transformation rules that produce non-left-recursive productions. For example, we can transform the left-recursive production $A \rightarrow A\alpha \mid \beta$ into

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

- Using the transformation above, we can rewrite the grammar that has the right recursion:

$$E \rightarrow T E', \quad E' \rightarrow + T E' \mid \epsilon$$

Exercise

Describe an equivalent grammar without the left recursion.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

Your answer:

Exercise

Describe an equivalent grammar without the left recursion.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

Your answer:

$$E \rightarrow T E'$$

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

$$T \rightarrow F T'$$

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

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

Recap

So far, we learned transformations to obtain grammars suitable for top-down parsing.

Expression grammar:

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

Unambiguous version:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

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

Non-left-recursive version:

$$E \rightarrow T E'$$

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

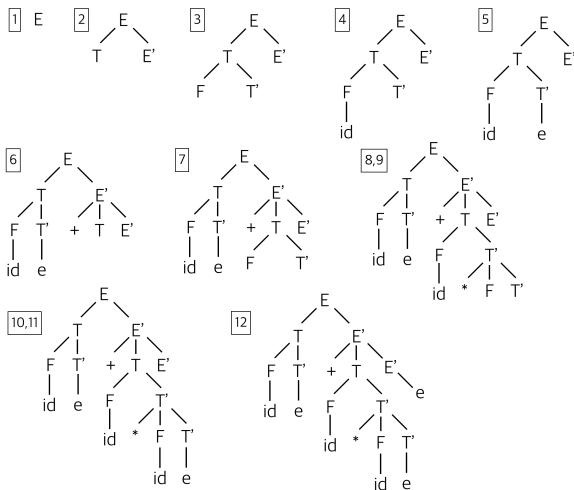
$$T \rightarrow F T'$$

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

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

Example

Top-down parsing sequence for the input string **id + id * id**:



The Key Problem in Top-Down Parsing

Top-down parsing replaces the leftmost nonterminals with the body of some production. How to determine which production to use?

- **Recursive-decent parsing** uses backtracking.
- **Predictive parsing** uses a parsing table without backtracking (more efficient).

In particular, we will cover $LL(1)^2$ parsing, a type of predictive parsing looking at 1 symbol ahead in the input.³

²Left-to-right scanning, Leftmost derivation, 1-symbol lookahead

³Predictive parsers looking at k symbols ahead are called $LL(k)$ parsers.

Parsing Table

To look ahead at the input string, a predictive parser uses a **parsing table** for the expression grammar:

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

- The leftmost column indicates non-terminal symbols.
- The topmost row indicates the next input symbols (look-ahead tokens).
- \$ is a special “endmarker” to indicate the end of an input string.
- If a nonterminal symbol X is given and the next input symbol is y , apply the production in the entry (X, y) of the table.

Predictive Parsing Example

The sequence of predictive parsing for $\text{id} + \text{id} * \text{id}$:

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

Stack	Input	Action
$E\$$	$\text{id} + \text{id} * \text{id}\$$	
$TE'\$$	$\text{id} + \text{id} * \text{id}\$$	
$FT'E'\$$	$\text{id} + \text{id} * \text{id}\$$	
$\text{id}T'E'\$$	$\text{id} + \text{id} * \text{id}\$$	match
$T'E'\$$	$+ \text{id} * \text{id}\$$	
$E'\$$	$+ \text{id} * \text{id}\$$	
$+TE'\$$	$+ \text{id} * \text{id}\$$	match
$TE'\$$	$\text{id} * \text{id}\$$	
$FT'E'\$$	$\text{id} * \text{id}\$$	
$\text{id}T'E'\$$	$\text{id} * \text{id}\$$	match
$T'E'\$$	$* \text{id}\$$	
$*FT'E'\$$	$* \text{id}\$$	match
$FT'E'\$$	$\text{id}\$$	
$\text{id}T'E'\$$	$\text{id}\$$	match
$T'E'\$$	$\$$	
$E'\$$	$\$$	
$\$$	$\$$	accept

Predictive Parsing Algorithm

Input: a string w and a parsing table M for grammar G

Output: a leftmost derivation of w or an error indication

let a be the first symbol of w

let X be the top stack symbol

while ($X \neq \$$) { /* repeat until stack becomes empty */

if ($X = a$) pop the stack and let a be the next symbol of w

else if (X is a terminal) error

else if ($M[X, a]$ is empty) error

else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {

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

pop the stack (remove the top stack symbol)

push Y_k, Y_{k-1}, \dots, Y_1 onto the stack (with Y_1 on top)

}

let X be the top stack symbol /* update the top stack symbol */

}

Constructing Parsing Table

A predictive parser processes an input string using a parsing table. How to construct the parsing table?

- ① Compute ***FIRST*** and ***FOLLOW*** sets of the grammar.
 - ▶ Both sets are used to look 1 symbol ahead.
- ② Construct the parsing table using these sets.

FIRST and *FOLLOW*

Definition

Given a string α of terminal and non-terminal symbols, $FIRST(\alpha)$ is the set of all terminal symbols that can begin any string derived from α .

- If $\alpha \Rightarrow^* c\beta$, then $c \in FIRST(\alpha)$.
- As an exception, if $\alpha \Rightarrow^* \epsilon$, $\epsilon \in FIRST(\alpha)$.

Definition

For a non-terminal X , $FOLLOW(X)$ is the set of terminals a that can appear immediately to the right of X in some sentential form.

- If $S \Rightarrow^* \alpha X a \beta$, then $a \in FOLLOW(X)$.
- As an exception, if $S \Rightarrow^* \alpha X$, $\$ \in FOLLOW(X)$.

Example

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

- $FIRST(E)$
- $FIRST(E')$
- $FIRST(T)$
- $FIRST(T')$
- $FIRST(F)$
- $FOLLOW(E)$
- $FOLLOW(E')$
- $FOLLOW(T)$
- $FOLLOW(T')$
- $FOLLOW(F)$

Example

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

- $FIRST(E) =$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') =$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') = FIRST(+TE') \cup \{ \epsilon \} = \{ +, \epsilon \}$
- $FIRST(T) =$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') = FIRST(+TE') \cup \{ \epsilon \} = \{ +, \epsilon \}$
- $FIRST(T) = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(T') =$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

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

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') = FIRST(+TE') \cup \{ \epsilon \} = \{ +, \epsilon \}$
- $FIRST(T) = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(T') = FIRST(*FT') \cup \{ \epsilon \} = \{ *, \epsilon \}$
- $FIRST(F) = FIRST("(E)") \cup FIRST(\text{id}) = \{ (, \text{id} \}$
- $FOLLOW(E) =$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') = FIRST(+TE') \cup \{ \epsilon \} = \{ +, \epsilon \}$
- $FIRST(T) = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(T') = FIRST(*FT') \cup \{ \epsilon \} = \{ *, \epsilon \}$
- $FIRST(F) = FIRST("(E)") \cup FIRST(\text{id}) = \{ (, \text{id} \}$
- $FOLLOW(E) = \{ \$ \} \cup FIRST(")") = \{ \$,) \}$
- $FOLLOW(E') = FOLLOW(E) = \{ \$,) \}$
- $FOLLOW(T) = FOLLOW(E) \cup FOLLOW(E')$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') = FIRST(+TE') \cup \{ \epsilon \} = \{ +, \epsilon \}$
- $FIRST(T) = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(T') = FIRST(*FT') \cup \{ \epsilon \} = \{ *, \epsilon \}$
- $FIRST(F) = FIRST("(E)") \cup FIRST(\text{id}) = \{ (, \text{id} \}$
- $FOLLOW(E) = \{ \$ \} \cup FIRST(")") = \{ \$,) \}$
- $FOLLOW(E') = FOLLOW(E) = \{ \$,) \}$
- $FOLLOW(T) =$
 $FOLLOW(E) \cup FOLLOW(E') \cup (FIRST(E') \setminus \{ \epsilon \}) = \{ +, \$,) \}$
- $FOLLOW(T') = FOLLOW(T) = \{ +, \$,) \}$
- $FOLLOW(F) =$
 $FOLLOW(T) \cup FOLLOW(T')$

Example

$$\begin{aligned}E &\rightarrow T E' \\E' &\rightarrow + T E' \mid \epsilon \\T &\rightarrow F T' \\T' &\rightarrow * F T' \mid \epsilon \\F &\rightarrow (E) \mid \text{id}\end{aligned}$$

- $FIRST(E) = FIRST(TE') = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(E') = FIRST(+TE') \cup \{ \epsilon \} = \{ +, \epsilon \}$
- $FIRST(T) = FIRST(FT') = \{ (, \text{id} \}$
- $FIRST(T') = FIRST(*FT') \cup \{ \epsilon \} = \{ *, \epsilon \}$
- $FIRST(F) = FIRST("(E)") \cup FIRST(\text{id}) = \{ (, \text{id} \}$
- $FOLLOW(E) = \{ \$ \} \cup FIRST(")") = \{ \$,) \}$
- $FOLLOW(E') = FOLLOW(E) = \{ \$,) \}$
- $FOLLOW(T) =$
 $FOLLOW(E) \cup FOLLOW(E') \cup (FIRST(E') \setminus \{ \epsilon \}) = \{ +, \$,) \}$
- $FOLLOW(T') = FOLLOW(T) = \{ +, \$,) \}$
- $FOLLOW(F) =$
 $FOLLOW(T) \cup FOLLOW(T') \cup (FIRST(T') \setminus \{ \epsilon \}) = \{ *, +, \$,) \}$

Algorithm for computing *FIRST*

To compute *FIRST*(X) for any string X from a grammar, apply the following rules until no more terminals or ϵ can be added to *FIRST* set for every grammar symbol:

- If X is a terminal, then $FIRST(X) = \{X\}$.
- When X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, add the followings to $FIRST(X) = FIRST(Y_1 Y_2 \cdots Y_k)$.
 - ▶ $FIRST(Y_1) \setminus \{\epsilon\}$, i.e., all non- ϵ symbols of $FIRST(Y_1)$
 - ▶ $FIRST(Y_2) \setminus \{\epsilon\}$ if $\epsilon \in FIRST(Y_1)$
 - ▶ $FIRST(Y_3) \setminus \{\epsilon\}$ if $\epsilon \in FIRST(Y_1), \epsilon \in FIRST(Y_2)$
 - ▶ ...
 - ▶ $FIRST(Y_k) \setminus \{\epsilon\}$ if $\epsilon \in FIRST(Y_j)$ for every $j \in [1, k - 1]$
 - ▶ ϵ if $\epsilon \in FIRST(Y_j)$ for every $j \in [1, k]$
- If $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$.

Algorithm for computing *FOLLOW*

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

Part 1. Compute an interim *FOLLOW*(X).

- If X is the start symbol, add \$ to *FOLLOW*(X).
- For every production $A \rightarrow \alpha X \beta$, add everything in *FIRST*(β) except for ϵ to *FOLLOW*(X).

$$FOLLOW(X) \leftarrow (FIRST(\beta) \setminus \{\epsilon\}) \cup FOLLOW(X)$$

Part 2. Propagate the interim result.

- For every production $X \rightarrow \alpha A$, add *FOLLOW*(X) to *FOLLOW*(A).
- For every production $X \rightarrow \alpha A \beta$, if $\epsilon \in FIRST(\beta)$, add *FOLLOW*(X) to *FOLLOW*(A).

Construction of Parsing Table

- **Goal:** Fill in each entry $M[A, a]^4$ of a predictive parsing table M by adding a production $A \rightarrow \alpha$ only when that production can be useful for processing the next input symbol a .
- **Input:** a context-free grammar G
- **Output:** a predictive parsing table M
- **Algorithm:** For each production $A \rightarrow \alpha$, do the following.
 - 1 For every terminal $a \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
 - 2 If $\epsilon \in FIRST(\alpha)$, for each terminal $b \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.
 - 3 Similarly, if $\epsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.

If a resulting parsing table for a grammar contains at most one production (i.e., does not contain conflicting productions), the grammar is in **LL(1)** grammar.

⁴ A is a nonterminal and a is a terminal or $\$$.

Example

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

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

Summary

- Transformations for top-down parsing: eliminating ambiguity and left-recursion
- $LL(k)$ is a kind of top-down parsing, which predicts productions by looking at the next k input symbols.
- $LL(1)$ parsing algorithm: use *FIRST* and *FOLLOW* to predict a production by looking at the next input symbol.