# 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 grammar[1]:

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

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

- $\mathbf{1} + \mathbf{2} * \mathbf{3}$: $(\mathbf{1} + \mathbf{2}) * \mathbf{3}$ vs. $\mathbf{1} + (\mathbf{2} * \mathbf{3})$
- $\mathbf{1} + \mathbf{2} + \mathbf{3}$: $(\mathbf{1} + \mathbf{2}) + \mathbf{3}$ vs. $\mathbf{1} + (\mathbf{2} + \mathbf{3})$

To eliminate the ambiguity, we should impose:

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

Q. How to remove the ambiguity?

[1] In this course, we rely on $\boldsymbol{P}$ to define $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{T}, \boldsymbol{S}, \boldsymbol{P})$, 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:

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

- An equivalent unambiguous grammar:

$$E \to E + T \mid T$$
$$T \to T * F \mid F \text{ /* introduce terms } (T) \text{ to prefer } * \text{ */}$$
$$F \to \mathbf{id} \mid (E)$$

# Exercises

- Draw a parse tree for $\mathbf{id} + \mathbf{id} + \mathbf{id}$.
- Draw a parse tree for $\mathbf{id} + \mathbf{id} * \mathbf{id}$.
- Transform the grammar

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

  so that $*$ associate to the right.

# Exercises

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

  $$\boldsymbol{E} \Rightarrow_l \boldsymbol{E}+\boldsymbol{T} \Rightarrow_l \boldsymbol{E}+\boldsymbol{T}+\boldsymbol{T} \Rightarrow_l \boldsymbol{T}+\boldsymbol{T}+\boldsymbol{T} \Rightarrow_l \cdots \Rightarrow_l \mathbf{id}+\mathbf{id}+\mathbf{id}$$

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

  $$\boldsymbol{E} \Rightarrow_l \boldsymbol{E} + \boldsymbol{T} \Rightarrow_l \boldsymbol{T} + \boldsymbol{T} \Rightarrow_l \boldsymbol{T} * \boldsymbol{F} + \boldsymbol{T} \Rightarrow_l \boldsymbol{F} * \boldsymbol{F} + \boldsymbol{T}$$
  $$\Rightarrow_l \mathbf{id} * \boldsymbol{F} + \boldsymbol{T} \Rightarrow_l \cdots \Rightarrow_l \mathbf{id} * \mathbf{id} + \mathbf{id}$$

- Transform the grammar

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

  so that $*$ associate to the right.
  Replace $\boldsymbol{T} * \boldsymbol{F}$ with $\boldsymbol{F} * \boldsymbol{T}$.

# 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.

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

- 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 $E \to E + T$ forever.
- We can remove left-recursion using transformation rules that produce non-left-recursive productions. For example, we can transform the left-recursive production $A \to A\alpha \mid \beta$ into

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

# 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.

$$\boldsymbol{E} \rightarrow \boldsymbol{E} + \boldsymbol{T} \mid \boldsymbol{T}$$
$$\boldsymbol{T} \rightarrow \boldsymbol{T} * \boldsymbol{F} \mid \boldsymbol{F}$$
$$\boldsymbol{F} \rightarrow \mathbf{id} \mid (\boldsymbol{E})$$

Your answer:

# Exercise

Describe an equivalent grammar without the left recursion.

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

Your answer:

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

# Recap

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

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

Unambiguous version:

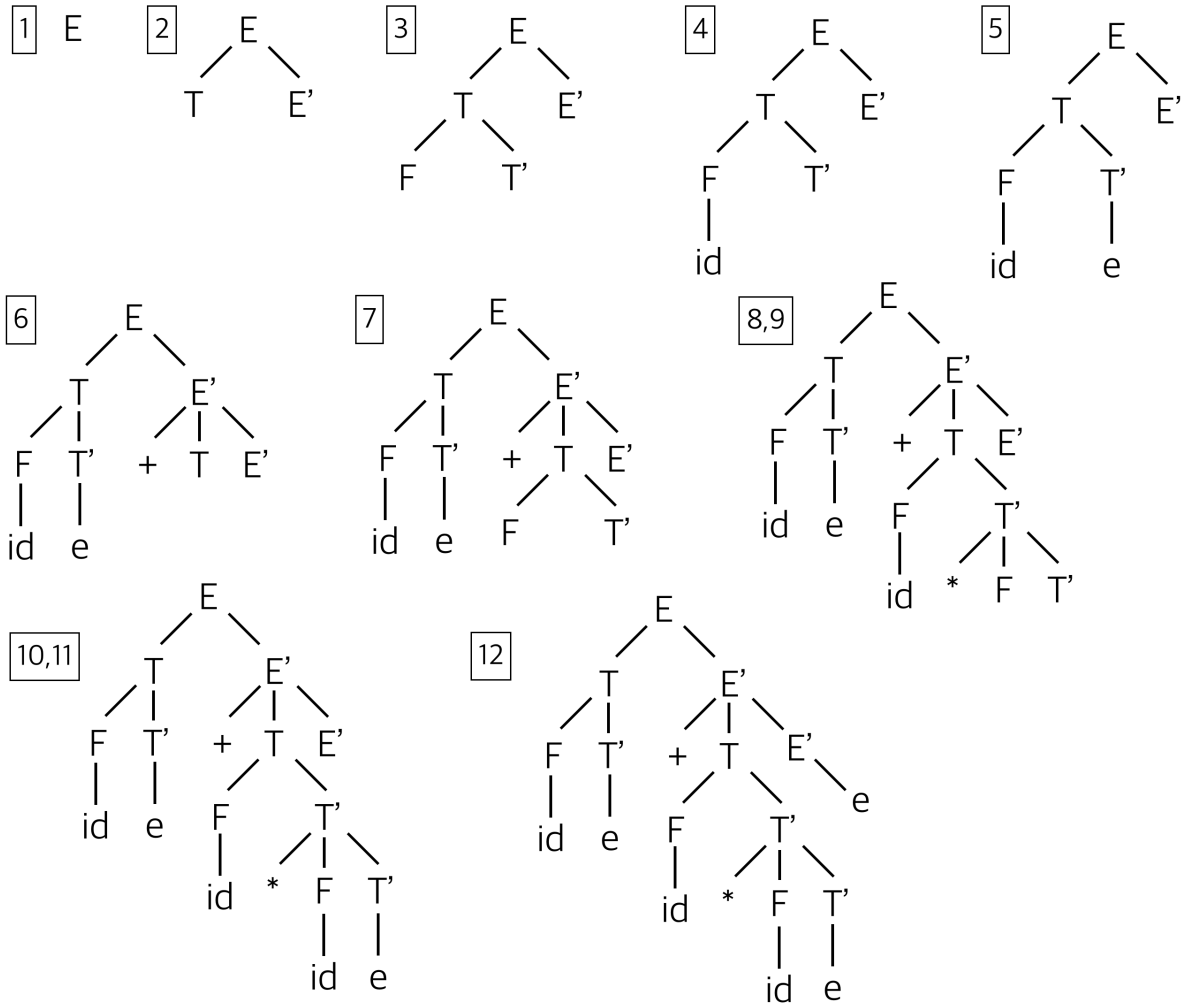
$$\begin{aligned} \boldsymbol{E} &\rightarrow \boldsymbol{E} + \boldsymbol{T} \mid \boldsymbol{T} \\ \boldsymbol{T} &\rightarrow \boldsymbol{T} * \boldsymbol{F} \mid \boldsymbol{F} \\ \boldsymbol{F} &\rightarrow \mathbf{id} \mid (\boldsymbol{E}) \end{aligned}$$

Non-left-recursive version:

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

# Example

Top-down parsing sequence for the input string $\mathbf{id} + \mathbf{id} * \mathbf{id}$:



# 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 $\boldsymbol{LL(1)}$[2] parsing, a type of predictive parsing looking $\mathbf{1}$ symbol ahead in the input.[3]

[2] **L**eft-to-right scanning, **L**eftmost derivation, **1**-symbol lookahead

[3] Predictive parsers looking $\boldsymbol{k}$ symbols ahead are called $\boldsymbol{LL(k)}$ parsers.

# Parsing Table

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

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| $E$ | $E \to T\ E'$ | | | $E \to T\ E'$ | | |
| $E'$ | | $E' \to +\ T\ E'$ | | | $E' \to \epsilon$ | $E' \to \epsilon$ |
| $T$ | $T \to F\ T'$ | | | $T \to F\ T'$ | | |
| $T'$ | | $T' \to \epsilon$ | $T' \to *\ F\ T'$ | | $T' \to \epsilon$ | $T' \to \epsilon$ |
| $F$ | $F \to \mathbf{id}$ | | | $F \to (E)$ | | |

- 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 $\mathbf{id} + \mathbf{id} * \mathbf{id}$:

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| $E$ | $E \to T\ E'$ | | | $E \to T\ E'$ | | |
| $E'$ | | $E' \to +\ T\ E'$ | | | $E' \to \epsilon$ | $E' \to \epsilon$ |
| $T$ | $T \to F\ T'$ | | | $T \to F\ T'$ | | |
| $T'$ | | $T' \to \epsilon$ | $T' \to *\ F\ T'$ | | $T' \to \epsilon$ | $T' \to \epsilon$ |
| $F$ | $F \to \mathbf{id}$ | | | $F \to (E)$ | | |

| Stack | Input | Action |
|---|---|---|
| $E\$$ | $\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ | |
| $TE'\$$ | $\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ | |
| $FT'E'\$$ | $\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ | |
| $\mathbf{id}T'E'\$$ | $\mathbf{id} + \mathbf{id} * \mathbf{id}\$$ | match |
| $T'E'\$$ | $+\mathbf{id} * \mathbf{id}\$$ | |
| $E'\$$ | $+\mathbf{id} * \mathbf{id}\$$ | |
| $+TE'\$$ | $+\mathbf{id} * \mathbf{id}\$$ | match |
| $TE'\$$ | $\mathbf{id} * \mathbf{id}\$$ | |
| $FT'E'\$$ | $\mathbf{id} * \mathbf{id}\$$ | |
| $\mathbf{id}T'E'\$$ | $\mathbf{id} * \mathbf{id}\$$ | match |
| $T'E'\$$ | $*\mathbf{id}\$$ | |
| $*FT'E'\$$ | $*\mathbf{id}\$$ | match |
| $FT'E'\$$ | $\mathbf{id}\$$ | |
| $\mathbf{id}T'E'\$$ | $\mathbf{id}\$$ | 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 $Y_k, Y_{k-1}, \ldots, Y_1$ 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 $\boldsymbol{FIRST}$ and $\boldsymbol{FOLLOW}$ sets of the grammar.
   - Both sets are used to look $\mathbf{1}$ symbol ahead.
2. Construct the parsing table using these sets.

# $FIRST$ and $FOLLOW$

## Definition

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

- 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

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

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

# Example

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

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

# Algorithm for computing $FIRST$

To compute $FIRST(X)$ for all grammar symbol $X$, apply the following rules until no more terminals or $\epsilon$ can be added to any $FIRST$ set:

- If $X$ is a terminal, then $FIRST(X) = \{X\}$.
- When $X$ is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$,
  - If for some $i$, a terminal symbol $a$ is in $FIRST(Y_i)$ and $\epsilon$ is in all of $FIRST(Y_1), \ldots, FIRST(Y_{i-1})$, place $a$ in $FIRST(X)$.
  - If $\epsilon$ is in $FIRST(Y_j)$ for all $j = 1, 2, \ldots, k$, then add $\epsilon$ to $FIRST(X)$.
- If $X \to \epsilon$ is a production, then add $\epsilon$ to $FIRST(X)$.

# Algorithm for computing $\boldsymbol{FIRST}$

To compute $\boldsymbol{FIRST}$ for any string $\boldsymbol{X_1 X_2 \cdots X_n}$, add the followings to $\boldsymbol{FIRST(X_1 X_2 \cdots X_n)}$

- all non-$\epsilon$ symbols of $\boldsymbol{FIRST(X_1)}$
- all non-$\epsilon$ symbols of $\boldsymbol{FIRST(X_2)}$, if $\epsilon \in \boldsymbol{FIRST(X_1)}$
- all non-$\epsilon$ symbols of $\boldsymbol{FIRST(X_3)}$, if $\epsilon \in \boldsymbol{FIRST(X_1)}$ and $\epsilon \in \boldsymbol{FIRST(X_2)}$
- $\cdots$
- $\epsilon$ if, for all $i$, $\epsilon \in \boldsymbol{FIRST(X_i)}$

# Algorithm for computing $\boldsymbol{FOLLOW}$

To compute $\boldsymbol{FOLLOW}(A)$ for all nonterminals $A$, apply the following rules until nothing can be added to any $\boldsymbol{FOLLOW}$ set:

1. Place \$ in $\boldsymbol{FOLLOW}(S)$, where $S$ is the start symbol.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in $\boldsymbol{FIRST}(\beta)$ except for $\epsilon$ is in $\boldsymbol{FOLLOW}(B)$.
3. If there is a production $A \rightarrow \alpha B$, then everything in $\boldsymbol{FOLLOW}(A)$ is in $\boldsymbol{FOLLOW}(B)$.
4. If there is a production $A \rightarrow \alpha B \beta$, where $\boldsymbol{FIRST}(\beta)$ contains $\epsilon$, then everything in $\boldsymbol{FOLLOW}(A)$ is in $\boldsymbol{FOLLOW}(B)$.

# Intuition on Construction of Parsing Table

- Goal: Incorporate the information from $\boldsymbol{FIRST}$ and $\boldsymbol{FOLLOW}$ sets into a predictive parsing table $M[A, a]$, where $A$ is a nonterminal and $a$ is a terminal or \$.
- Basic idea:
  - If the next input symbol $a$ is in $\boldsymbol{FIRST}(\alpha)$, choose $A \to \alpha$.
  - If $\alpha \Rightarrow^* \epsilon$ (i.e., $\epsilon \in \boldsymbol{FIRST}(\alpha)$) and $a \in \boldsymbol{FOLLOW}(A)$, choose $A \to \alpha$.
- Given $A \to \alpha \mid \beta$ and the next symbol $a$, if $a \in \boldsymbol{FIRST}(\alpha)$ and $a \in \boldsymbol{FIRST}(\beta)$ (i.e., $\boldsymbol{FIRST}(\alpha) \cap \boldsymbol{FIRST}(\beta) \neq \emptyset$), the grammar cannot be parsed using predictive parsing.

# Construction of Parsing Table

Algorithm:

- Input: grammar $G$
- Output: parsing table $M$.
- Algorithm: For each production $A \to \alpha$ of the grammar, do the following:
  1. For each terminal $a$ in $FIRST(\alpha)$, add $A \to \alpha$ to $M[A, a]$.
  2. If $\epsilon$ is in $FIRST(\alpha)$, then for each terminal $b$ in $FOLLOW(A)$, add $A \to \alpha$ to $M[A, b]$.
     Similarly, if $\epsilon$ is in $FIRST(\alpha)$ and \$ is in $FOLLOW(A)$, add $A \to \alpha$ to $M[A, \$]$ as well.

# Example

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| $E$ | $E \to T\ E'$ | | | $E \to T\ E'$ | | |
| $E'$ | | $E' \to +\ T\ E'$ | | | $E' \to \epsilon$ | $E' \to \epsilon$ |
| $T$ | $T \to F\ T'$ | | | $T \to F\ T'$ | | |
| $T'$ | | $T' \to \epsilon$ | $T' \to *\ F\ T'$ | | $T' \to \epsilon$ | $T' \to \epsilon$ |
| $F$ | $F \to \text{id}$ | | | $F \to (E)$ | | |

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

# Summary

- Transformations for top-down parsing: eliminating ambiguity and left-recursion
- Some grammars can be parsed in top-down by looking at the next input symbol.
- $LL(1)$ parsing algorithm: $FIRST$, $FOLLOW$, parsing table