Chapter 3: Syntax Analysis

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (to be discussed in lab sessions)

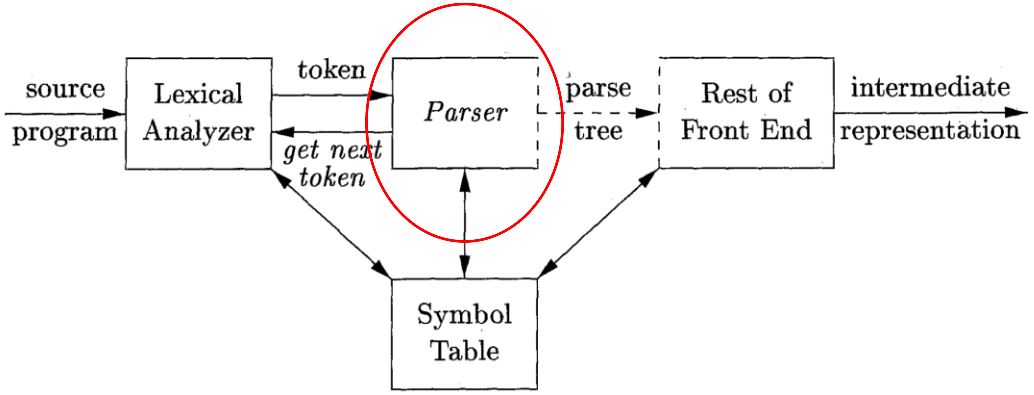
Describing Syntax

- The syntax of programming language constructs can be specified by context-free grammars[1]
  - A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language
  - For certain grammars, we can automatically construct an efficient parser
  - A properly designed grammar defines the structure of a language and helps translate source programs into correct object code and detect errors

[1]Can also be specified using BNF (Backus-Naur Form) notation, which basically can be seen as a variant of CFG: http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node23.html

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language
- Report syntax errors in an intelligent fashion
- For well-formed programs, the parser constructs a parse tree
  - The parse tree need not be constructed explicitly



Classification of Parsers

- **Universal parsers (通用语法分析器)**
  - Some methods (e.g., Earley's algorithm[1]) can parse any grammar
  - However, they are too inefficient to be used in practice
- **Top-down parsers (自顶向下语法分析器)**
  - Construct parse trees from the top (root) to the bottom (leaves)
- **Bottom-up parsers (自底向上语法分析器)**
  - Construct parse trees from the bottom (leaves) to the top (root)

Note: Top-down and bottom-up parsing both scan the input from left to right, one symbol at a time. They work only for certain grammars, which are expressive enough.

[1] http://loup-vaillant.fr/tutorials/earley-parsing/

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp
  - Grammar design (Lab)
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Context-Free Grammar (上下文无关文法)

- **A context-free grammar (CFG) consists of four parts:**
  - **Terminals (终结符号):** Basic symbols from which strings are formed (token names)
  - **Nonterminals (非终结符号):** Syntactic variables that denote sets of strings
    - Usually correspond to a language construct, such as `stmt` (statements)
  - One nonterminal is distinguished as the **start symbol (开始符号)**
    - The set of strings denoted by the start symbol is the language generated by the CFG
  - **Productions (产生式):** Specify how the terminals and nonterminals can be combined to form strings
    - **Format:** `head → body`
    - head must be a nonterminal; body consists of zero or more terminals/nonterminals
    - **Example:** *`expression → expression + term`*

CFG Example

- The grammar below defines simple arithmetic expressions
  - Terminal symbols: `id`, `+`, `-`, `*`, `/`, `(`, `)`
  - Nonterminals: *expression*, *term* (项), *factor* (因子)
  - Start symbol: *expression*
  - Productions:
    - *expression* → *expression* + *term*
    - *expression* → *expression* - *term*
    - *expression* → *term*
    - *term* → *term* * *factor*
    - *term* → *term* / *factor*
    - *term* → *factor*
    - *factor* → (*expression*)
    - *factor* → **id**

Notational Simplification

```
expression → expression + term
expression → expression - term
expression → term
term → term * factor
term → term / factor
term → factor
factor → ( expression )
factor → id
```

⇒

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | id
```

- | is a meta symbol to specify alternatives
- (and) are not meta symbols, they are terminal symbols

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp
  - Grammar design (Lab)
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Derivations

- **Derivation (推导):** Starting with the start symbol, nonterminals are rewritten using productions until only terminals remain

- Example:
  - **CFG:** $E \rightarrow -\ E \mid E + E \mid E * E \mid (\ E\ ) \mid \mathbf{id}$
  - A derivation (a sequence of rewrites) of $-(\mathbf{id})$ from E
    - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

Notations

- $\Rightarrow$ means "derives in one step"
- $\overset{*}{\Rightarrow}$ means "derives in zero or more steps"
  - $\alpha \overset{*}{\Rightarrow} \alpha$ holds for any string α
  - If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$
  - Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$ can be written as $E \overset{*}{\Rightarrow} -(\mathbf{id})$
- $\overset{+}{\Rightarrow}$ means "derives in one or more steps"

Terminologies

- If $S \overset{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, we say that α is ***sentential form*** of G (文法的句型)
  - May contain both terminals and nonterminals, and may be empty
  - **Example:** $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$, here all strings of grammar symbols are sentential forms
- A ***sentence*** **(句子)** of G is a sentential form with no nonterminals
  - In the above example, only the last string $-(\mathbf{id}+\mathbf{id})$ is a sentence
- The *language generated* by a grammar is its set of sentences

Leftmost/Rightmost Derivations

- At each step of a derivation, we need to choose which nonterminal to replace

- In **leftmost derivations (最左推导)**, the leftmost nonterminal in each sentential form is always chosen to be replaced
  - $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$

- In **rightmost derivations (最右推导)**, the rightmost nonterminal is always chosen to be replaced
  - $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\mathbf{id}) \underset{rm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$

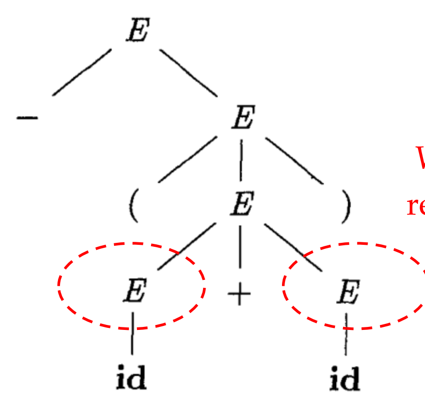
Parse Trees (语法分析树)

- A ***parse tree*** is a graphical representation of a derivation that <u>filters out the order</u> in which productions are applied
  - The root node (根结点) is the start symbol of the grammar
  - Each leaf node (叶子结点) is labeled by terminal symbol*
  - Each interior node (内部结点) is labeled with a nonterminal symbol and represents the application of a production
    - The interior node is labeled with the nonterminal in the head of the production; the children nodes are labeled, from left to right, by the symbols in the body of the production

CFG: $E \rightarrow -E \mid E + E \mid E * E \mid (E) \mid \textbf{id}$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$

* Here, we assume that a derivation always produces a string with only terminals, so leaf nodes cannot be non-terminals.



Which one is replaced first?

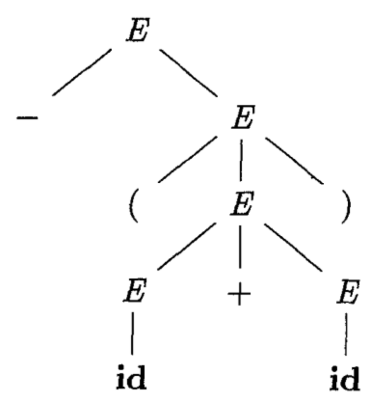
Parse Trees (语法分析树) Cont.

- The leaves, from left to right, constitute a **sentential form** of the grammar, which is called the *yield* or *frontier* of the tree
- There is a **many-to-one** relationship between derivations and parse trees
  - However, there is a **one-to-one** relationship between leftmost/rightmost derivations and parse trees

CFG: $E \rightarrow -\ E \mid E + E \mid E * E \mid (\ E\ ) \mid \textbf{id}$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\textbf{id}) \underset{rm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$



Both derivations correspond to the parse tree.

Outline

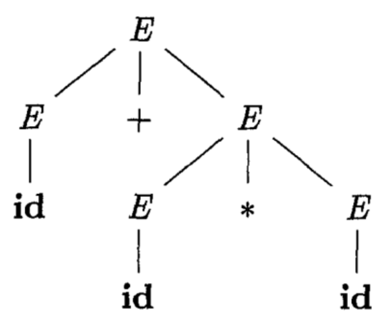
- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

- Formal definition of CFG
- Derivation and parse tree
- Ambiguity
- CFG vs. regexp
- Grammar design (Lab)

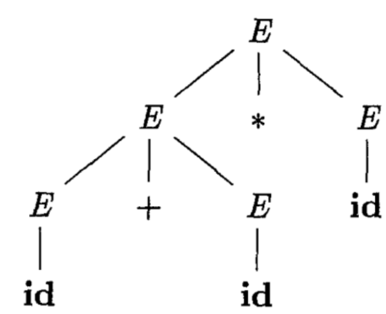
Ambiguity (二义性)

- Given a grammar, if there are more than one parse tree for some sentence, it is ambiguous.
- Example CFG: $E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



Both are leftmost derivations

The left tree corresponds to the commonly assumed precedence.

Ambiguity (二义性) Cont.

- The grammar of a programming language typically needs to be unambiguous
  - Otherwise, there will be multiple ways to interpret a program
  - Given $E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$, how to interpret $a + b * c$?

- In some cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules to discard undesirable parse trees
  - For example: multiplication before addition

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
    - Formal definition of CFG
    - Derivation and parse tree
    - Ambiguity
    - CFG vs. regexp
    - Grammar design (Lab)
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

CFG vs. Regular Expressions

- **CFGs are more expressive than regular expressions**

  1. Every language that can be described by a regular expression can also be described by a grammar (i.e., every regular language is also a context-free language)

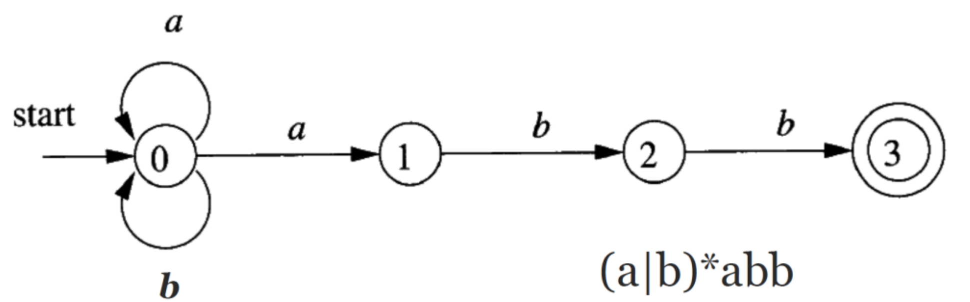
  2. Some context-free languages cannot be described using regular expressions

Any Regular Language Can be Described by a CFG

- **(Proof by Construction)** Each regular language can be accepted by an NFA. We can construct a CFG to describe the language:
  - For each state i of the NFA, create a nonterminal symbol A_i
  - If state i has a transition to state j on input a, add the production $A_i \rightarrow aA_j$
  - If state i goes to state j on input ϵ, add the production $A_i \rightarrow A_j$
  - If i is an accepting state, add $A_i \rightarrow \epsilon$
  - If i is the start state, make A_i be the start symbol of the grammar

Example: NFA to CFG

- $A_0 \to aA_0 \mid bA_0 \mid aA_1$
- $A_1 \to bA_2$
- $A_2 \to bA_3$
- $A_3 \to \epsilon$



(a|b)*abb

Consider the string *baabb*: The process of the NFA accepting the sentence corresponds exactly to the derivation of the sentence from the grammar

Some Context-Free Languages Cannot be Described Using Regular Expressions

- Example: $L = \{a^n b^n \mid n > 0\}$
  - The language L can be described by CFG $S \rightarrow aSb \mid ab$
  - L cannot be described by regular expressions. In other words, we cannot construct a DFA to accept L

Proof by Contradiction

- Suppose there is a DFA D that accepts L and D has k states
- When processing a^{k+1} ..., D must enter a state s more than once (D enters one state after processing a symbol)[1]
- Assume that D enters the state s after reading the ith and jth a ($i \neq j, i \leq k+1, j \leq k+1$)
- Since D accepts L, $a^j b^j$ must reach an accepting state. There must exist a path labeled b^j from s to an accepting state
- Since a^i reaches the state s and there is a path labeled b^j from s to an accepting state, D will accept $a^i b^j$. Contradiction!!!

[1] $a^{k+1}b^{k+1}$ is a string in L so D must accept it

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

- Formal definition of CFG
- Derivation and parse tree
- Ambiguity
- CFG vs. regexp
- Grammar design (Lab)

Grammar Design

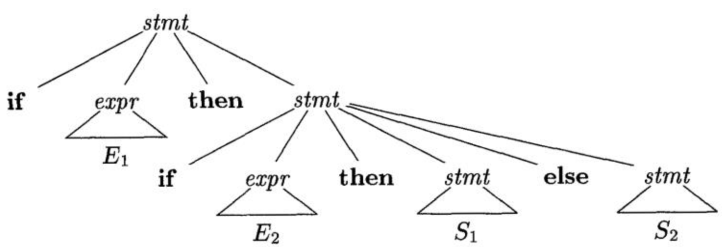
- CFGs are capable of describing most, but not all, of the syntax of programming languages
  - "Identifiers should be declared before use" cannot be described by a CFG
  - Subsequent phases must analyze the output of the parser to ensure compliance with such rules

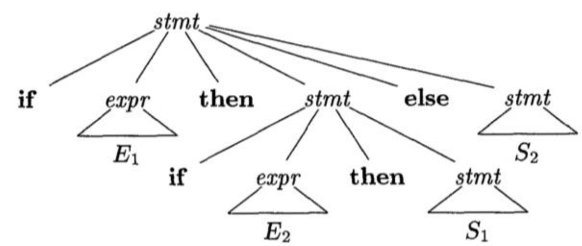
- Before parsing, we typically apply several transformations to a grammar to make it more suitable for parsing
  - Eliminating ambiguity (消除二义性)
  - Eliminating left recursion (消除左递归)
  - Left factoring (提取左公因子)

Eliminating Ambiguity (1)

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
& | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
& | & \textbf{other}
\end{array}
$$

Two parse trees for **`if`** E_1 **`then`** **`if`** E_2 **`then`** S_1 **`else`** S_2





Which parse tree is preferred in programming?
(i.e., else matches which then?)

Eliminating Ambiguity (2)

- **Principle of proximity:** match each `else` with the closest unmatched `then`
  - **Idea of rewriting:** A statement appearing between a `then` and an `else` must be matched (must not end with an unmatched `then`)

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
& | & \mathit{open\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{matched\_stmt} \\
& | & \textbf{other} \\
\mathit{open\_stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
& | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{open\_stmt}
\end{array}
$$

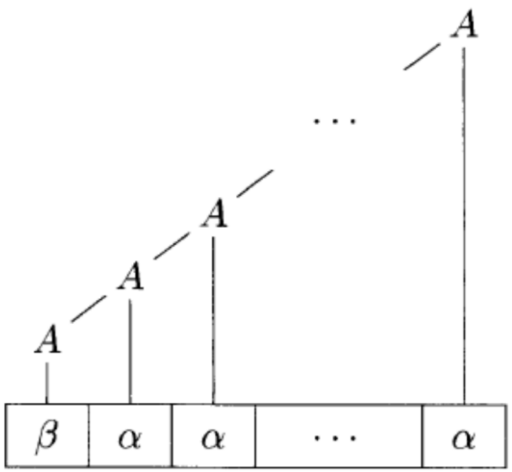
Rewriting grammars to eliminate ambiguity is difficult.
There are no general rules to guide the process.

Eliminating Left Recursion

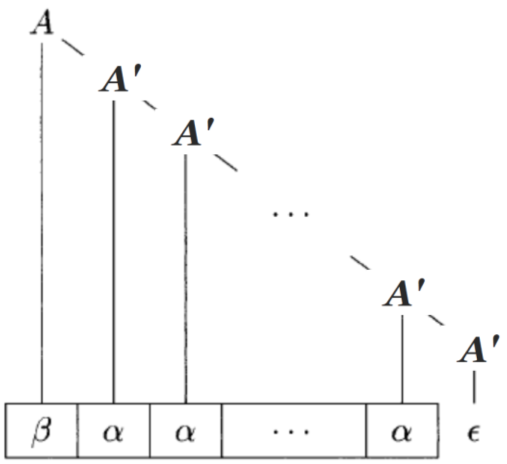
- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
  - Because $S \Rightarrow Aa \Rightarrow Sda$
- **Immediate left recursion (立即左递归):** the grammar has a production of the form $A \rightarrow A\alpha$
- Top-down parsing methods cannot handle left-recursive grammars (bottom-up parsing methods can handle…)

Eliminating Immediate Left Recursion

- Simple grammar: $A \to A\alpha \mid \beta$
  - It generates sentences starting with the symbol β followed by zero or more α's



- Replace the grammar by:
  - $A \to \beta A'$
  - $A' \to \alpha A' \mid \epsilon$
  - It is right recursive now



Eliminating Immediate Left Recursion

- The general case: $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$

- Replace the grammar by:
  - $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$
  - $A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

Example

$$\boxed{E \rightarrow E + T \mid T}$$
$$\boxed{T \rightarrow T * F \mid F}$$
$$F \rightarrow (E) \mid \textbf{id}$$

$$\Longrightarrow$$

$$\boxed{\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +\ TE' \mid \epsilon \end{array}}$$
$$\boxed{\begin{array}{l} T \rightarrow FT' \\ T' \rightarrow *\ FT' \mid \epsilon \end{array}}$$
$$F \rightarrow (\ E\ ) \mid \textbf{id}$$

Eliminating Left Recursion

- The technique for eliminating immediate left recursion does not work for the non-immediate left recursions
- The general left recursion eliminating algorithm (iterative)
  - **Input:** Grammar *G* with no cycles or ϵ-productions
  - **Output:** An equivalent grammar with no left recursion

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for (each i from 1 to n) {
    for (each j from 1 to $i-1$) {
        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
    }
    eliminate the immediate left recursion among the A_i-productions
}

Example

$$S \rightarrow Aa \mid b \quad A \rightarrow Ac \mid Sd \mid \epsilon$$

- Order the nonterminals: S, A
- $i = 1$:
  - The inner loop does not run; there is no immediate left recursion among S-productions
- $i = 2$:
  - $j = 1$, replace the production $A \rightarrow Sd$ by $A \rightarrow Aad \mid bd$
    - $A \rightarrow Aad \mid bd \mid Ac \mid \epsilon$
  - Eliminate immediate left recursion

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

The example grammar contains an ϵ-production, but it is harmless

Left Factoring (提取左公因子)

- If we have the following two productions

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$| \quad \textbf{if } expr \textbf{ then } stmt$$

- On seeing input **if**, we cannot immediately decide which production to choose
- In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two productions, and the input begins with a nonempty string derived from α. We may defer choosing productions by expanding A to $\alpha A'$ first

$$\boxed{\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}}$$

Algorithm: Left Factoring a Grammar

- **Input:** Grammar G
- **Output:** An equivalent left-factored grammar

- For each nonterminal A, find the longest prefix α common to two or more of its alternatives.

- If $\alpha \neq \epsilon$, replace all A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

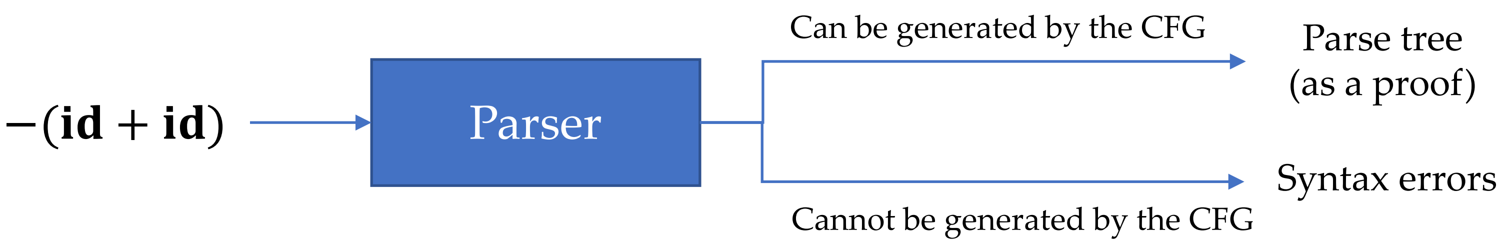
- Repeatedly apply the above transformation until no two alternatives for a nonterminal have a common prefix

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Parsing Revisited

- During program compilation, the syntax analyzer (a.k.a. parser) checks whether the string of token names produced by the lexer can be generated by the grammar for the source language
  - That is, if we can find a parse tree whose frontier is equal to the string, then the parser can declare "success"



CFG: $E \rightarrow -\ E \mid E + E \mid E * E \mid (\ E\ ) \mid \textbf{id}$

Top-Down Parsing

- **Problem definition:** Constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first)

- **Two basic actions of top-down parsing algorithms:**
  - **Predict:** At each step of parsing, determine the production to be applied for the leftmost nonterminal*
  - **Match:** Match the terminals in the chosen production's body with the input string

* So that the sentential forms always contain leading terminals to match with the prefix of the input string

Top-Down Parsing Example

- **Grammar**

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

- **Input string**

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

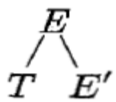
Is the input string a sentence of the grammar?

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

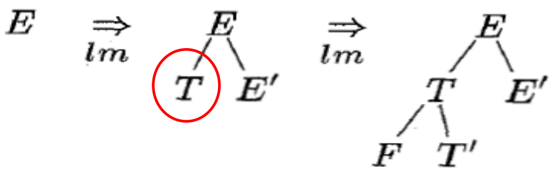
E

- **Grammar:** $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite:** TE'

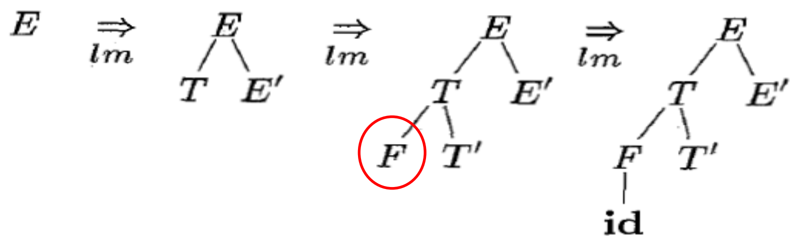
$E \underset{lm}{\Rightarrow}$



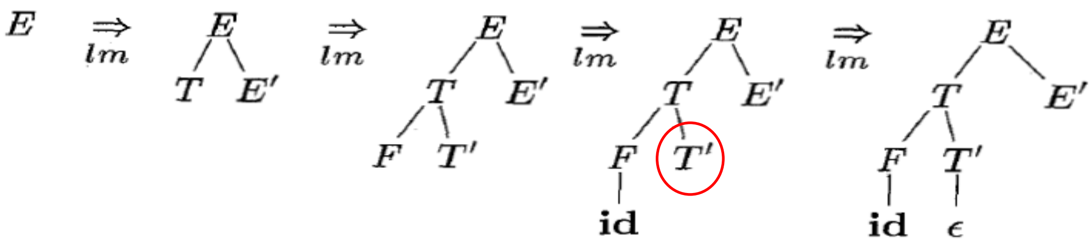
- **Grammar:** $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$ $\quad T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid id$
- **Input string: id + id $*$ id** **The sentential form after rewrite:** $FT'E'$



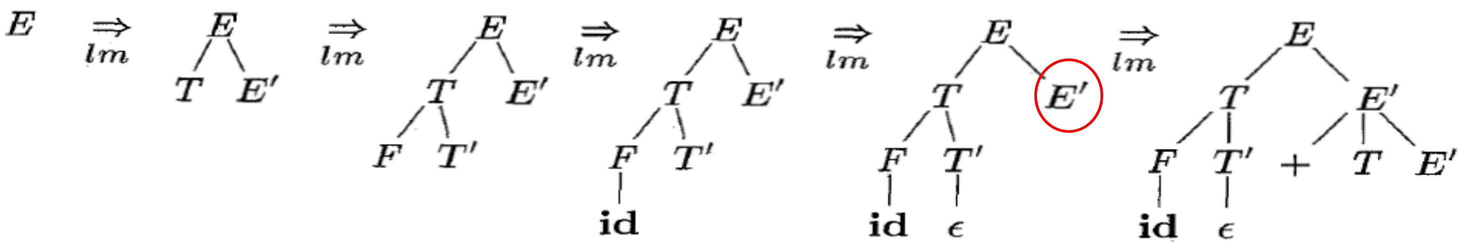
- **Grammar:** $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$ $\quad T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \underline{id}$
- **Input string: id + id ∗ id** **The sentential form after rewrite: id**$T'E'$



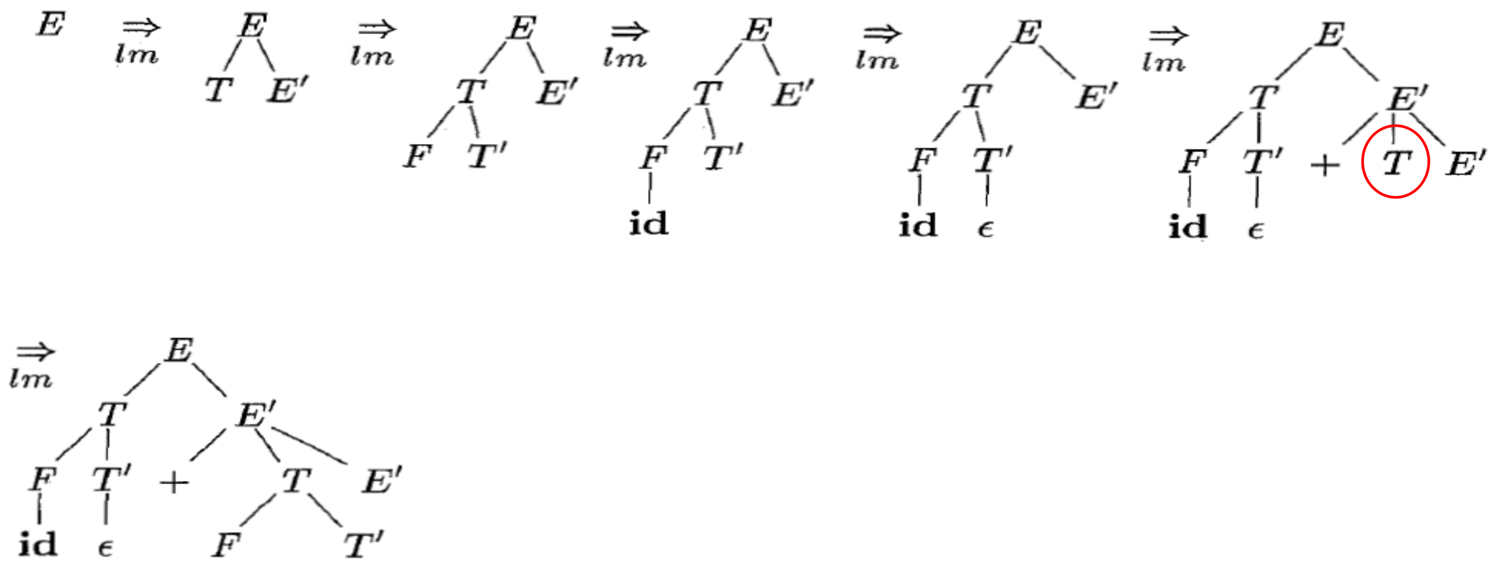
- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string: id + id $*$ id** **The sentential form after rewrite: idE'**



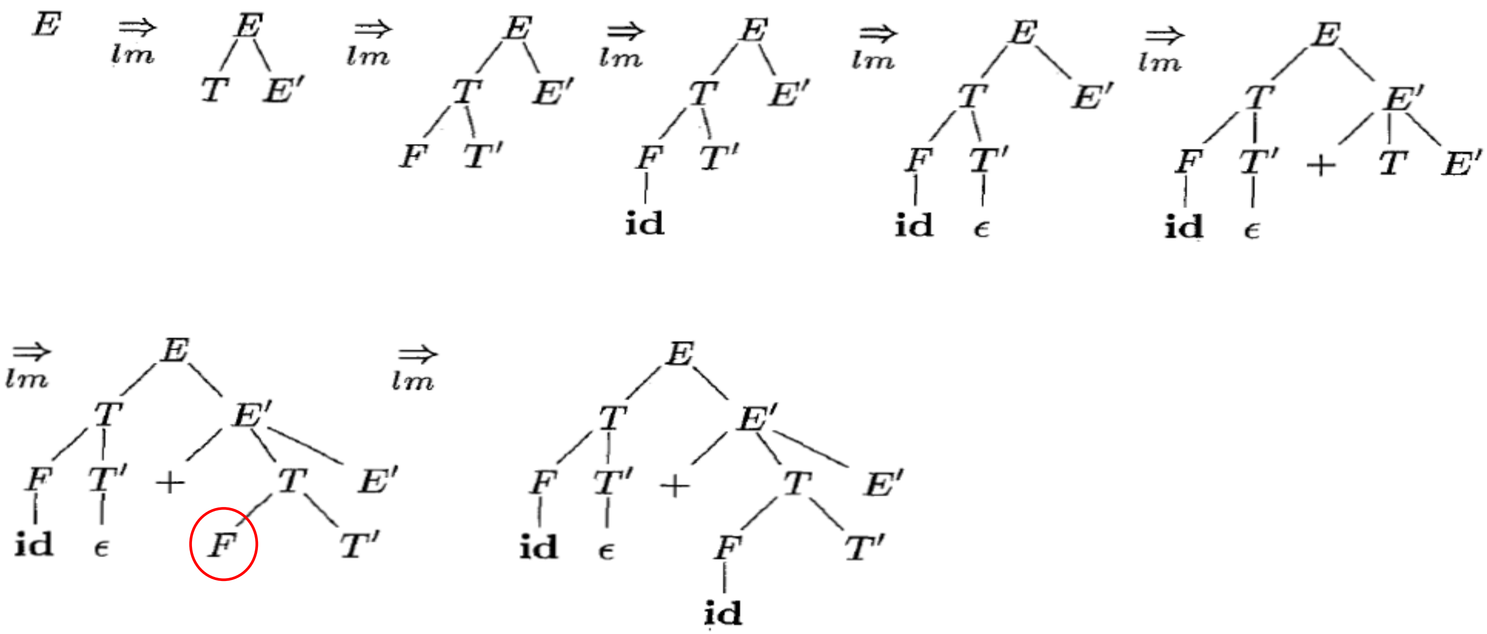
- **Grammar:** $E \to TE'$ $\quad E' \to +TE' \mid \epsilon$ $\quad T \to FT'$ $\quad T' \to * FT' \mid \epsilon$ $\quad F \to (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id +** TE'



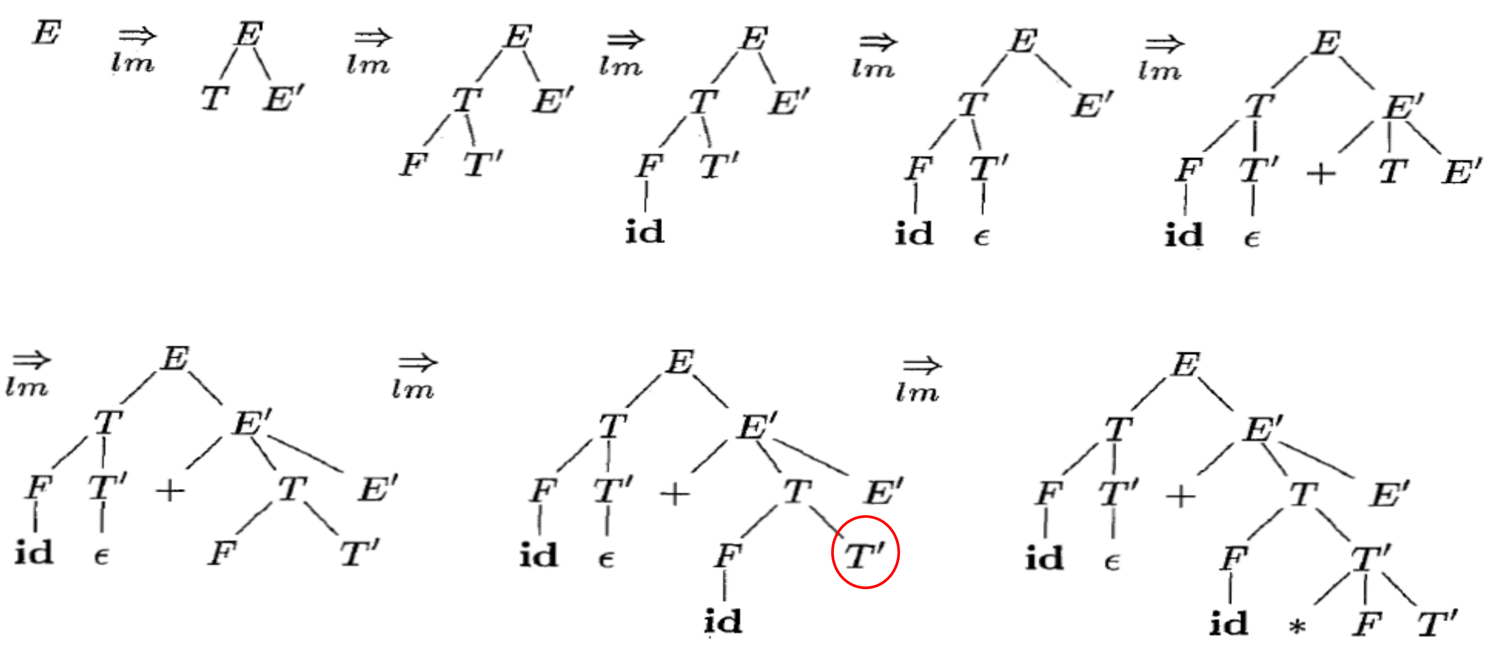
- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id +** $FT'E'$



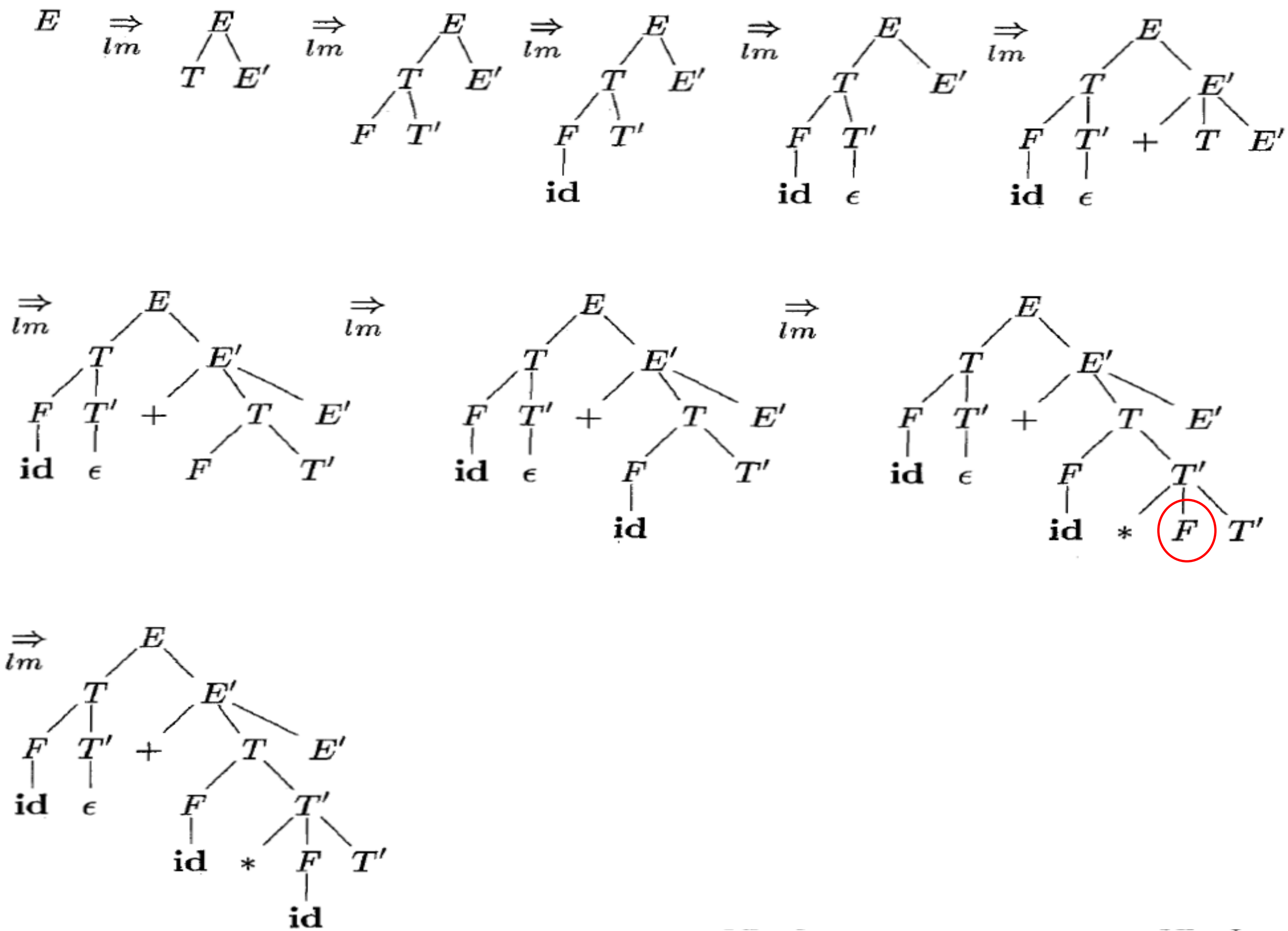
- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid \underline{id}$
- **Input string: id + id** $*$ **id** **The sentential form after rewrite: id + id**$T'E'$



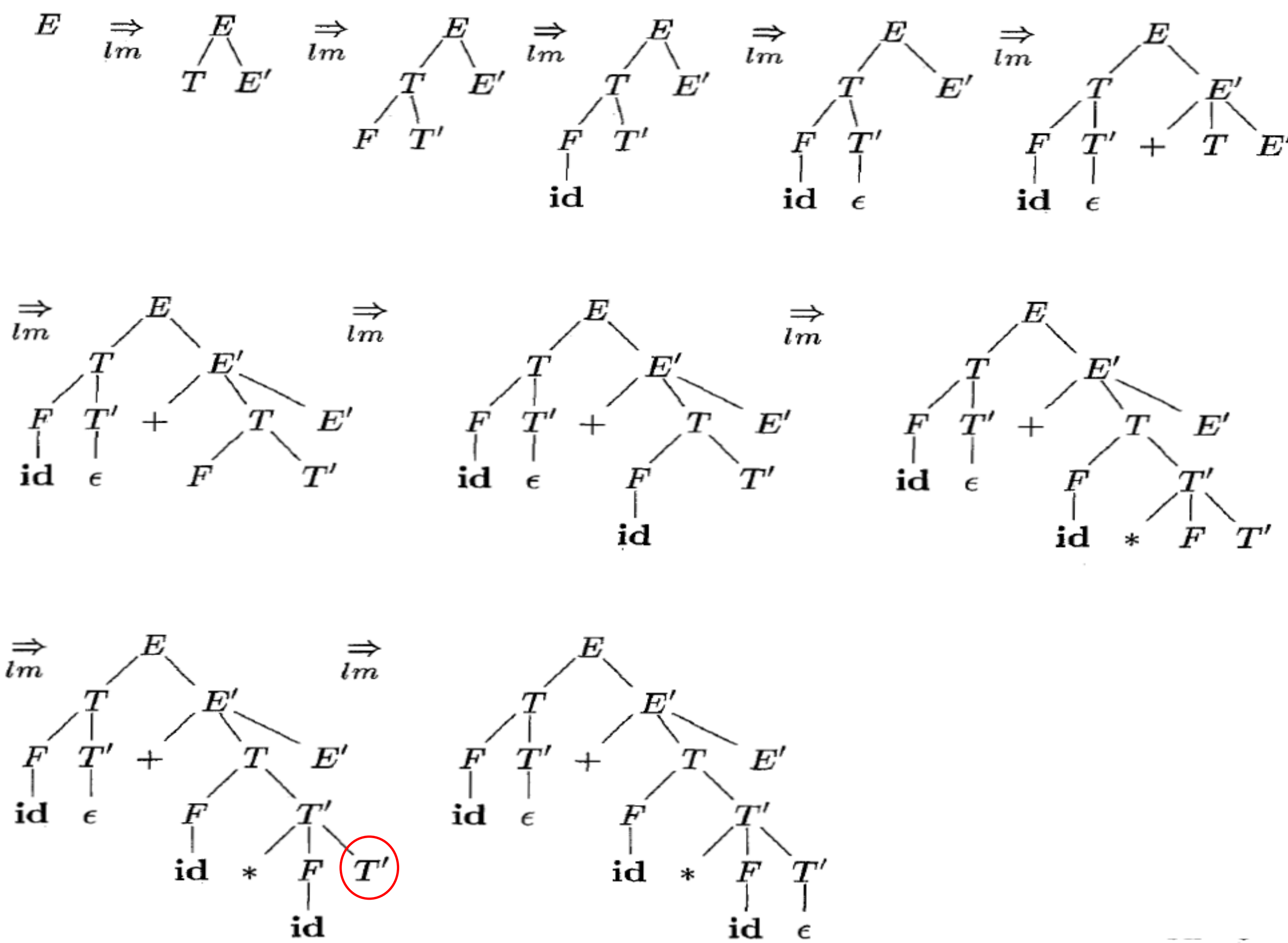
- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id + id** $* \; FT'E'$



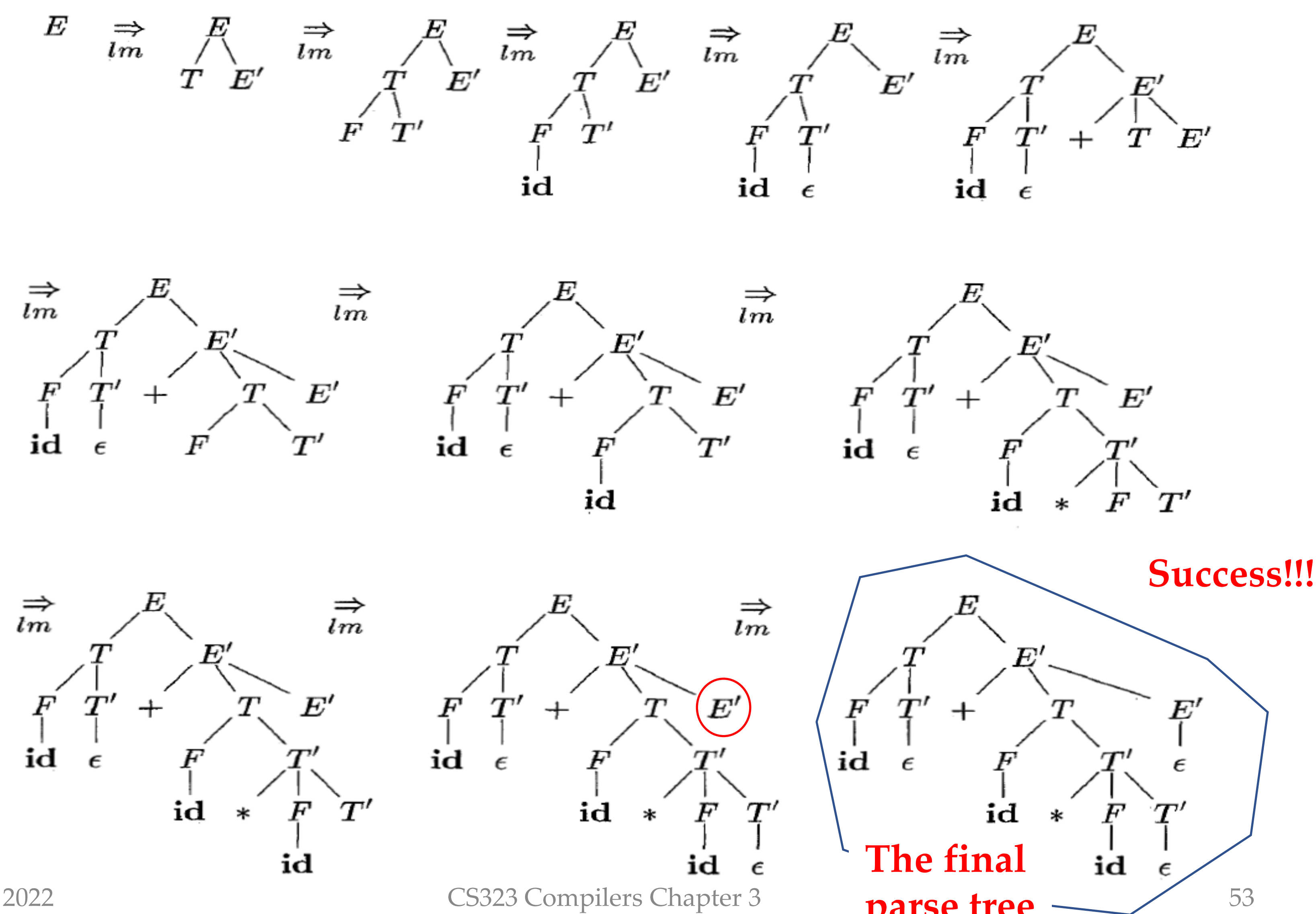
- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string:** **id + id * id** **The sentential form after rewrite:** **id + id * id** $T'E'$



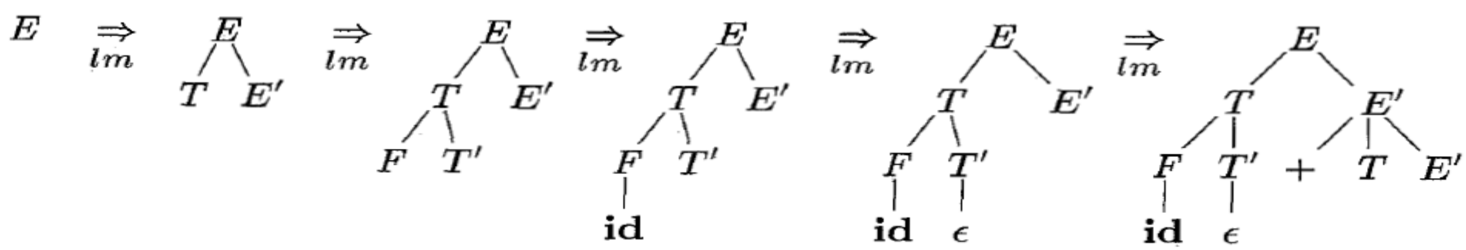
- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string: id + id ∗ id** **The sentential form after rewrite: id + id ∗ id** E'



- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id + id * id**

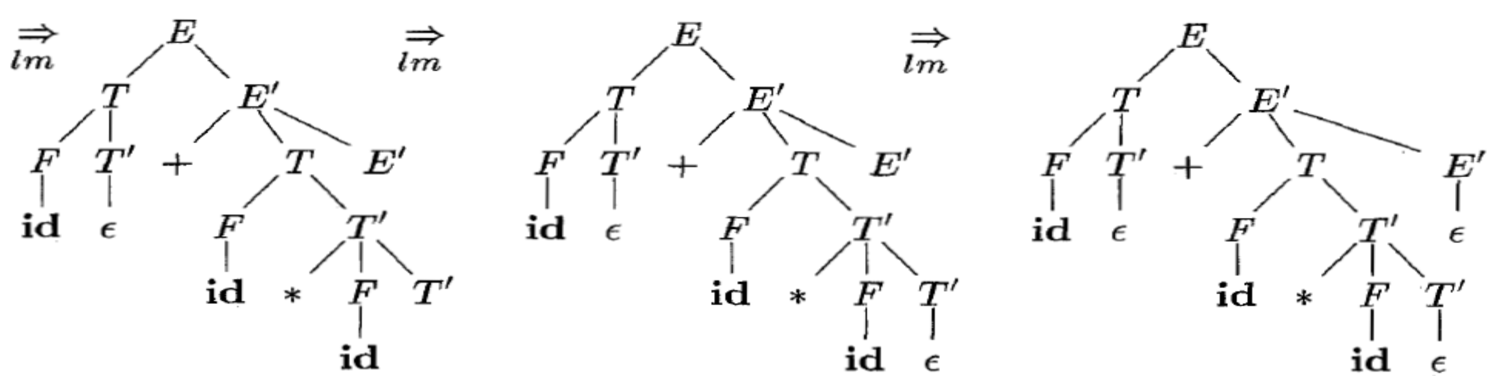


- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string:** **id + id $*$ id**



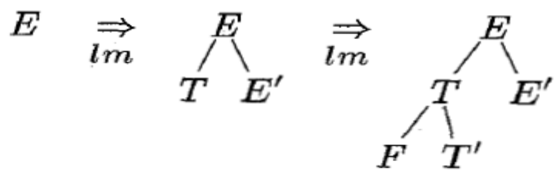
We can make two observations from the example:

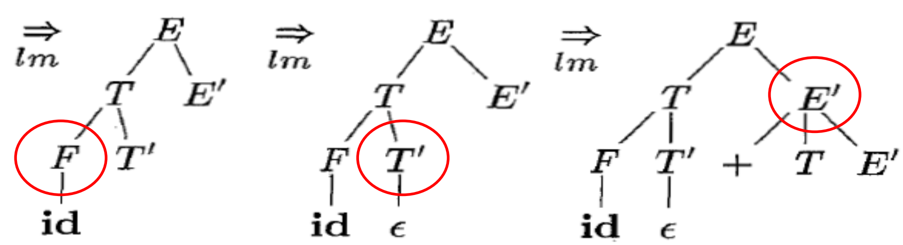
- Top-down parsing is equivalent to finding a leftmost derivation.
- At each step, the frontier of the tree is a left-sentential form.



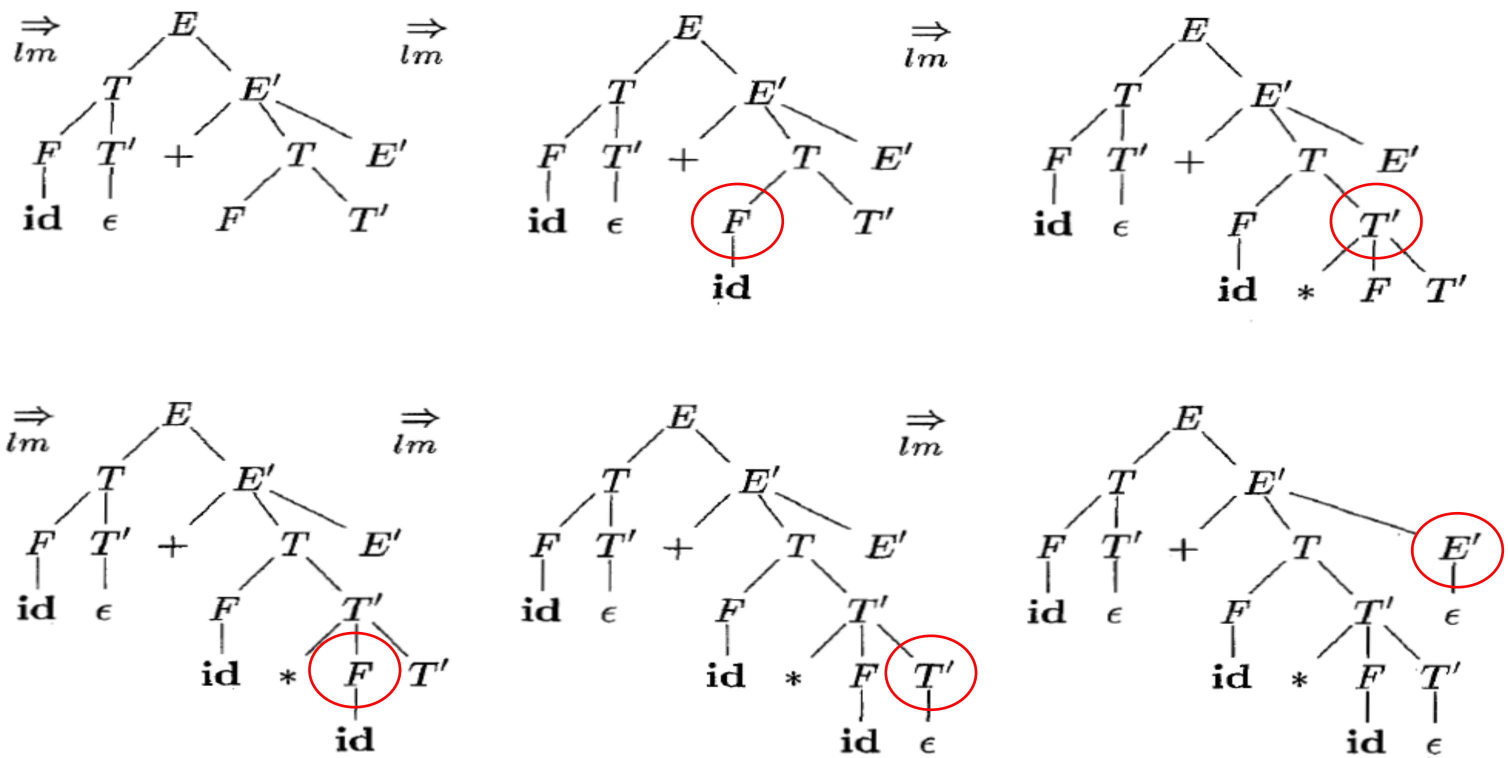
Key decision in top-down parsing: Which production to apply at each step?

Grammar: $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$





F has two productions, why do we choose the second one?



Bottom-Up Parsing

- **Problem definition:** Constructing a parse tree for an input string beginning at the leaves (terminals) and working up towards the root (start symbol of the grammar)

- Shift-reduce parsing (移入-归约分析) is a general style of bottom-up parsing (using a stack to hold grammar symbols). Two basic actions:
  - **Shift:** Move an input symbol onto the stack
  - **Reduce:** Replace a string at the stack top with a non-terminal that can produce the string (the reverse of a rewrite step in a derivation)

Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

$\mathbf{id} * \mathbf{id}$

Initially, the tree only contain leaf nodes

Shift-Reduce Parsing Example

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

Parsing steps on input $\textbf{id}_1 * \textbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\textbf{id}_1 * \textbf{id}_2 \$$	shift
$\$ \textbf{id}_1$	$* \textbf{id}_2 \$$	reduce by $F \rightarrow \textbf{id}$

$F \quad * \quad \textbf{id}$

$\textbf{id}$

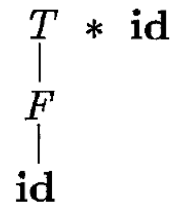
Tree "grows" when reduction happens

Shift-Reduce Parsing Example

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

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$



Tree "grows" when reduction happens

Shift-Reduce Parsing Example

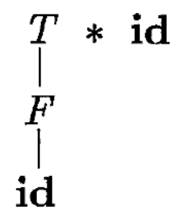
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift



Tree does not change when shift happens

Shift-Reduce Parsing Example

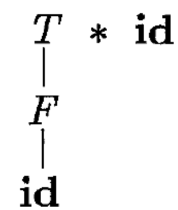
$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid \mathbf{id}$

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift



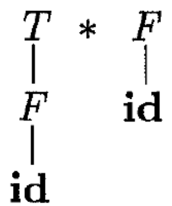
Tree does not change when shift happens

Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

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

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\, \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$\, F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$\, T$	$* \mathbf{id}_2 \$$	shift
$\$\, T *$	$\mathbf{id}_2 \$$	shift
$\$\, T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$



Tree "grows" when reduction happens

Shift-Reduce Parsing Example

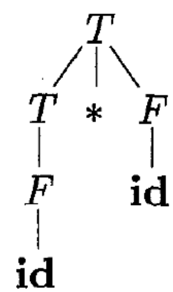
Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$



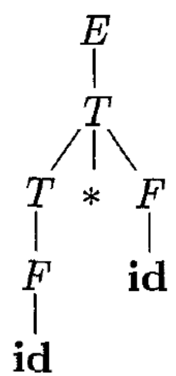
Tree "grows" when reduction happens

Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

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

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T *$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$\,T$	$\$$	reduce by $E \rightarrow T$



Tree "grows" when reduction happens

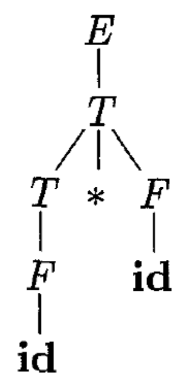
Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T *$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

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

Success!!!



The final parse tree

Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\, \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$\, F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$\, T$	$* \mathbf{id}_2 \$$	shift
$\$\, T *$	$\mathbf{id}_2 \$$	shift
$\$\, T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\, T * F$	$\$$	reduce by $T \to T * F$
$\$\, T$	$\$$	reduce by $E \to T$
$\$\, E$	$\$$	accept

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

Rightmost derivation:

$E \Rightarrow T$
$\Rightarrow T * F$
$\Rightarrow T * \text{id}$
$\Rightarrow F * \text{id}$
$\Rightarrow \text{id} * \text{id}$

We can make two observations from the example:

- Bottom-up parsing is equivalent to finding a rightmost derivation (in reverse).
- At each step, stack + remaining input is a right-sentential form.

Shift-Reduce Parsing Example

Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$

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

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

Key decisions:

1. When to shift? When to reduce?
2. Which production to apply when reducing (there could be multiple possibilities)?