南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

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)

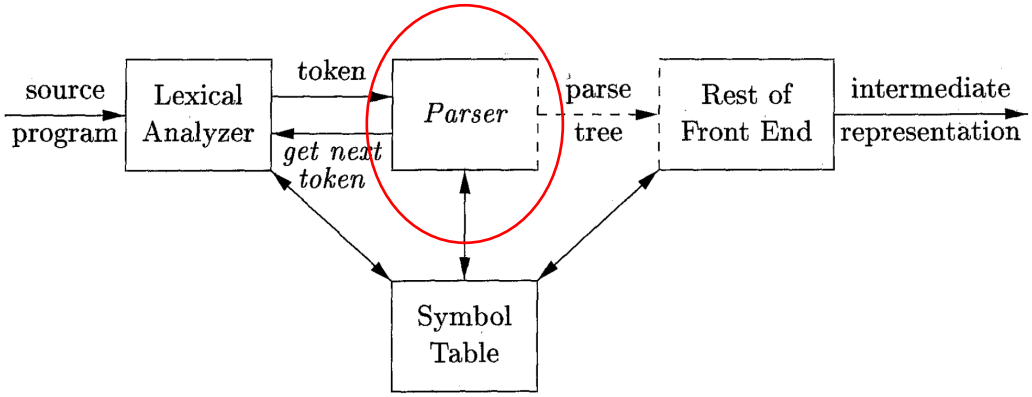
Describing Syntax

- The syntax of programming language constructs can be specified by context-free grammars[1]
  - A grammar gives a precise and easy-to-understand syntactic specification of a programming language, defining its structure
  - For certain grammars, we can automatically construct an efficient parser
  - A properly designed grammar 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
- 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)

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**

→ can be read as:
can be in the form, can be replaced by, can be re-written as, can produce, can generate, can make…

Notational Simplification

expression → *expression* + *term*

expression → *expression* – *term*

expression → *term*

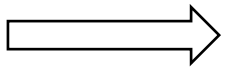
term → *term* * *factor*

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
- 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)

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

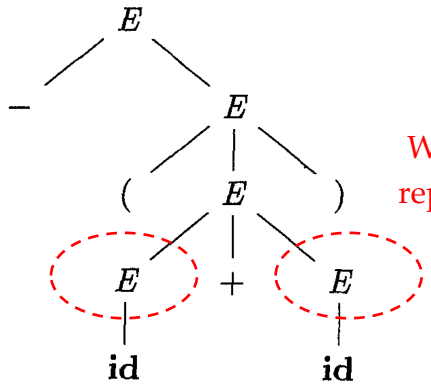
- In **rightmost derivations (最右推导)**, the rightmost nonterminal is always chosen to be replaced
  - $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\mathbf{id}) \underset{rm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$

Parse Trees (语法分析树)

- A ***parse tree*** is a graphical representation of a derivation that filters out the order 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 a 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: $E \rightarrow - E \mid E + E \mid E * E \mid (E) \mid \mathbf{id}$

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



Which one is replaced first?

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

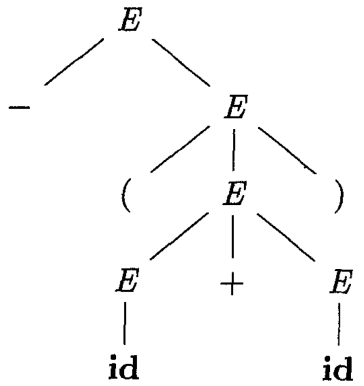
Parse Trees (语法分析树) Cont.

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

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

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



Both derivations correspond to the parse tree.

Outline

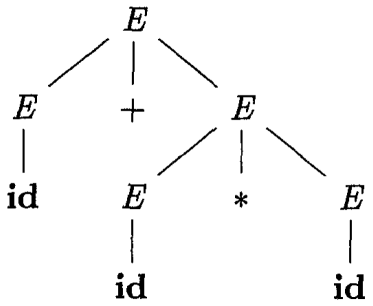
- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

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

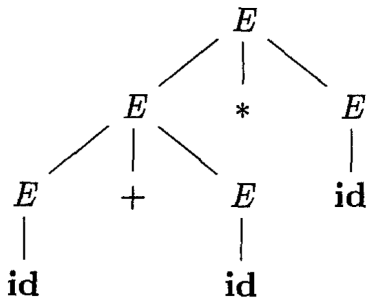
Ambiguity (二义性)

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

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



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



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

CFG vs. Regular Expressions

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

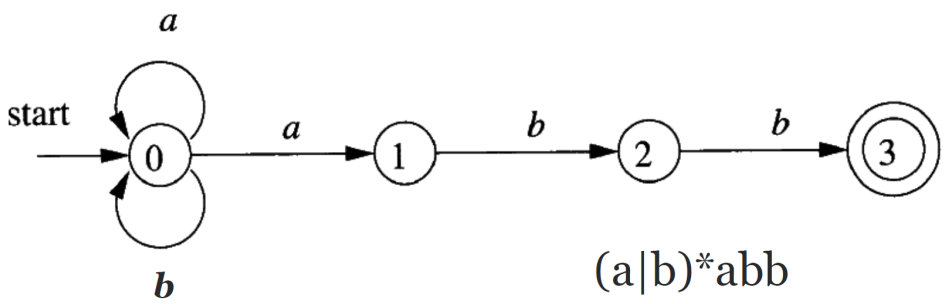
  1. Every language that can be described by a regular expression can also be described by a grammar (i.e., every regular language is also a context-free language)
  2. Some context-free languages cannot be described using regular expressions

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

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



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)

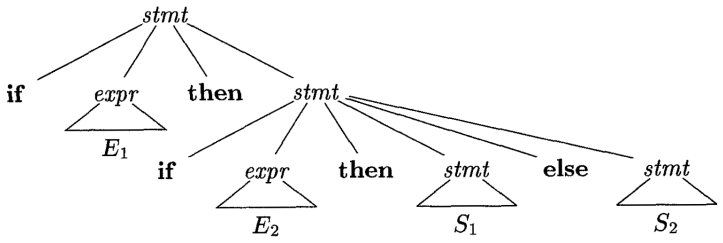
Grammar Design

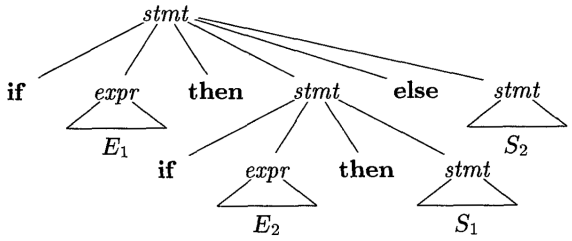
- CFGs are capable of describing most, but not all, of the syntax of programming languages
  - "Identifiers should be declared before use" cannot be described by a CFG
  - Subsequent phases must analyze the output of the parser to ensure compliance with such rules
- Before parsing, we typically apply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

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

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`)

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

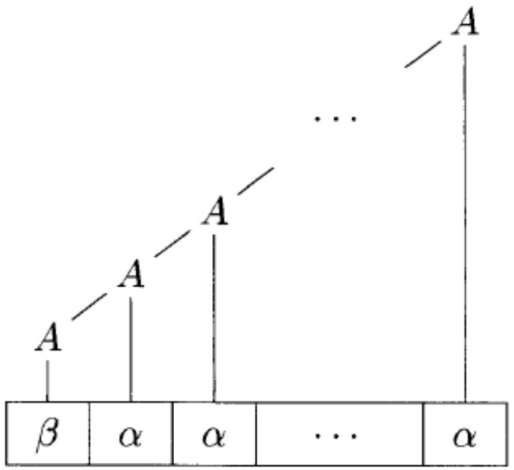
Rewriting grammars to eliminate ambiguity is difficult.
There are no general rules to guide the process.

Eliminating Left Recursion

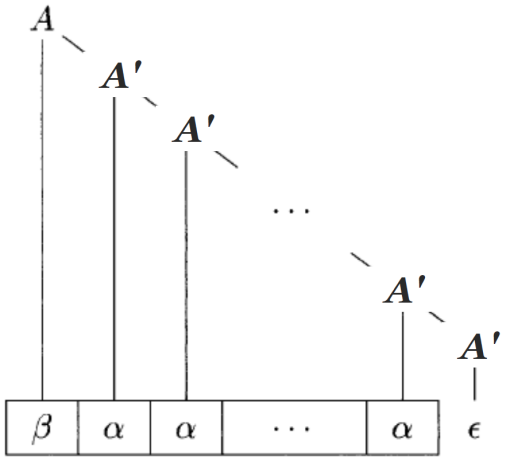
- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
  - Because $S \Longrightarrow Aa \Longrightarrow Sda$
- **Immediate left recursion (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: $A \rightarrow A\alpha \mid \beta$
  - It generates sentences starting with the symbol β followed by zero or more α's



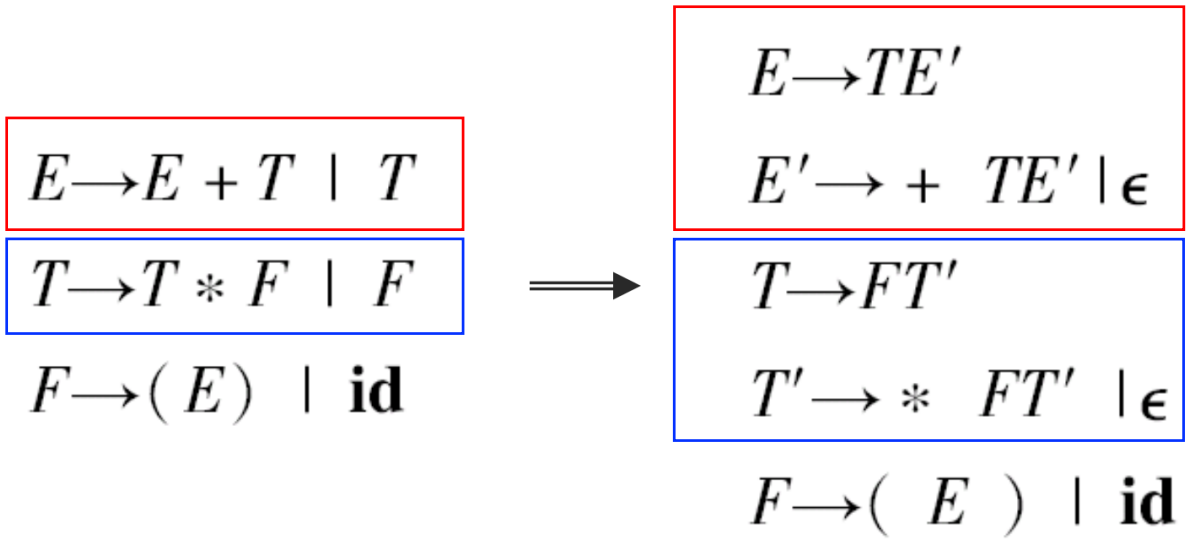
- Replace the grammar by:
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A' \mid \epsilon$
  - It is right recursive now



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


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

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 $A_i \rightarrow A_j\gamma$ by the
            productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
            $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

Example

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

- 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 $A \to Sd$ by $A \to Aad \mid bd$
    - $A \to Aad \mid bd \mid Ac \mid \epsilon$
  - Eliminate immediate left recursion

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

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

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

- 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

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

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 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

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

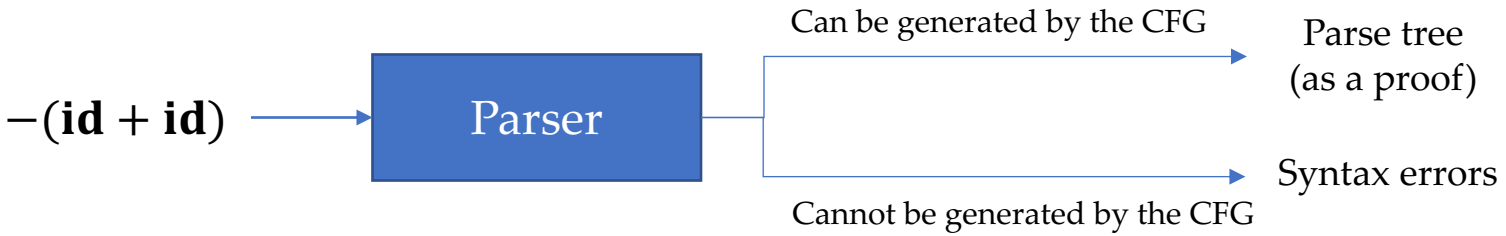
- Repeatedly apply the above transformation until no two alternatives for a nonterminal have a common prefix

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* → – *E* | *E* + *E* | *E* * *E* | (*E*) | **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* (of the parse tree's frontier)
  - **Match:** Match the terminals in the chosen production's body with the input string

* So that the sentential forms may 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**

id + id * id

Is the input string a sentence of the grammar?

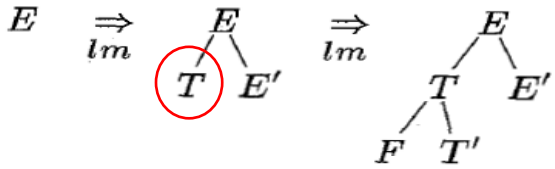
- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string:** **id + id * id**

E

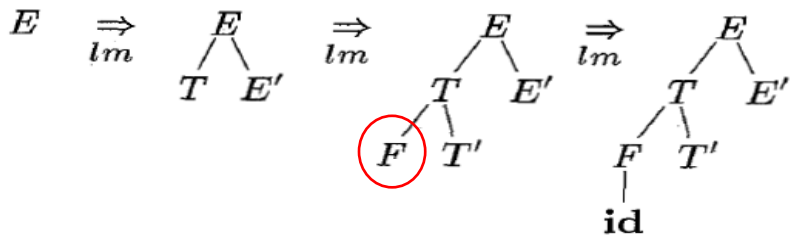
- **Grammar:** $E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$ $T \rightarrow FT'$ $T' \rightarrow * FT' \mid \epsilon$ $F \rightarrow (E) \mid id$
- **Input string:** **id + id * id** **The sentential form after rewrite:** TE'

$$E \underset{lm}{\Rightarrow} \begin{array}{c} E \\ / \quad \backslash \\ T \quad E' \end{array}$$

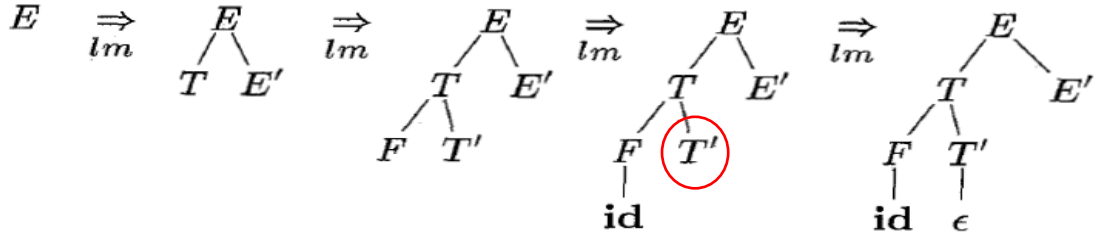
- **Grammar:** $E \to TE'$ $E' \to +TE' \mid \epsilon$ $T \to FT'$ $T' \to * FT' \mid \epsilon$ $F \to (E) \mid id$
- **Input string:** **id + id * id** **The sentential form after rewrite:** $FT'E'$



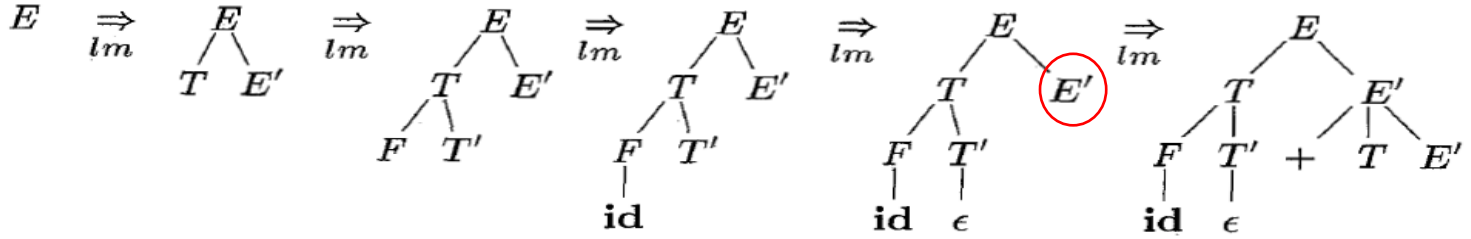
- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id**$T'E'$



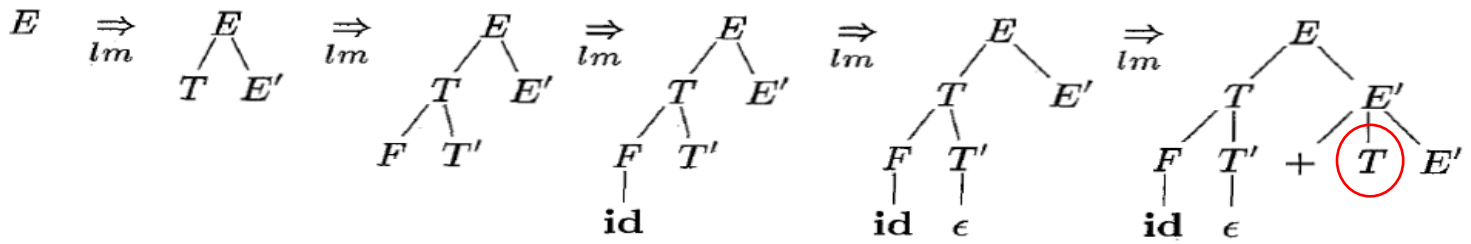
- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: idE'**

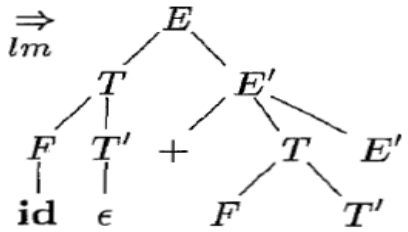


- **Grammar:** $E \to TE'$ $E' \to +TE' \mid \epsilon$ $T \to FT'$ $T' \to * FT' \mid \epsilon$ $F \to (E) \mid id$
- **Input string:** **id** + **id** $*$ **id** **The sentential form after rewrite:** **id** $+TE'$

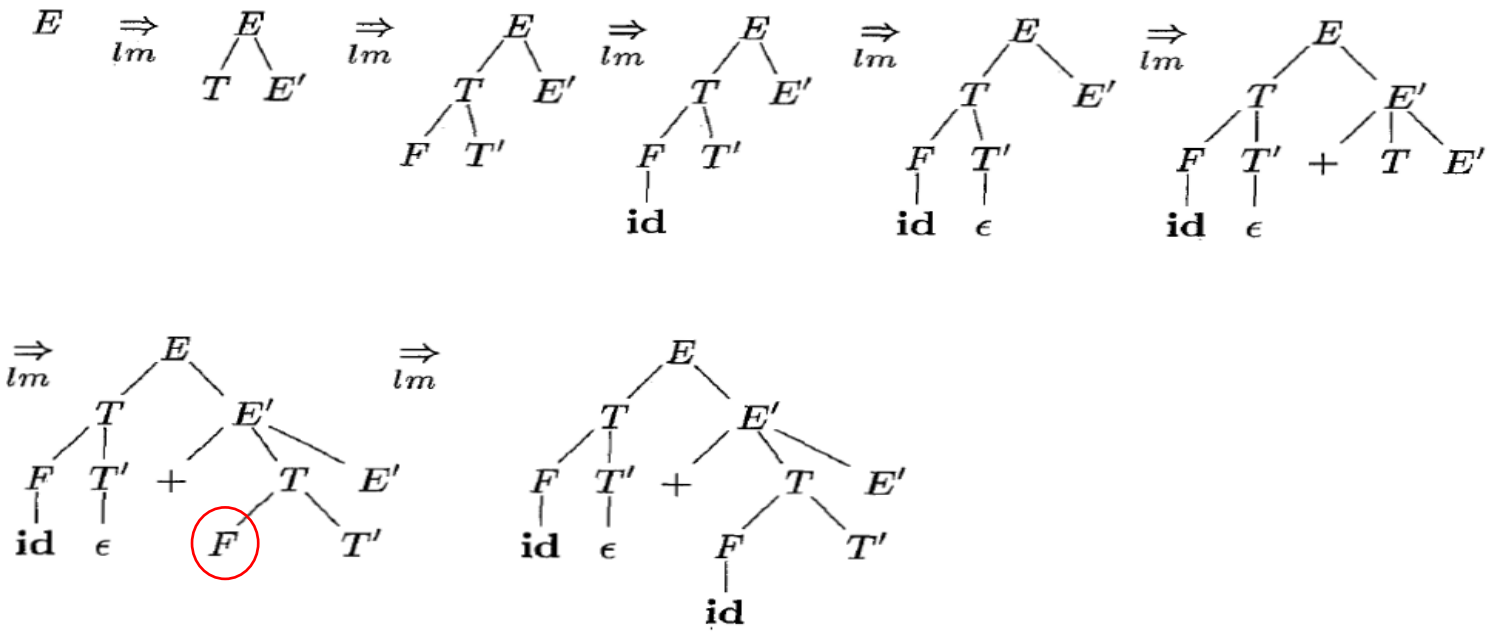


- **Grammar:** $E \to TE'$ $E' \to +TE' \mid \epsilon$ $T \to FT'$ $T' \to * FT' \mid \epsilon$ $F \to (E) \mid id$
- **Input string:** **id** + **id** $*$ **id** **The sentential form after rewrite:** **id** $+FT'E'$

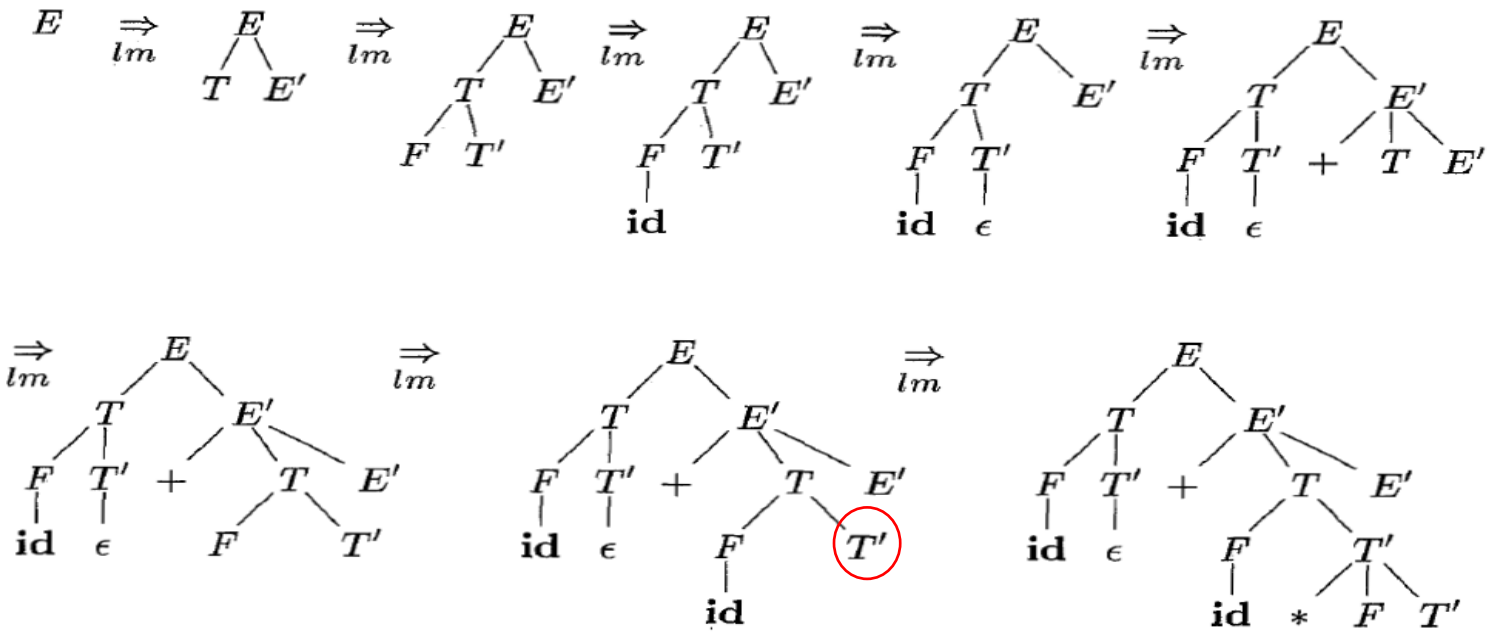




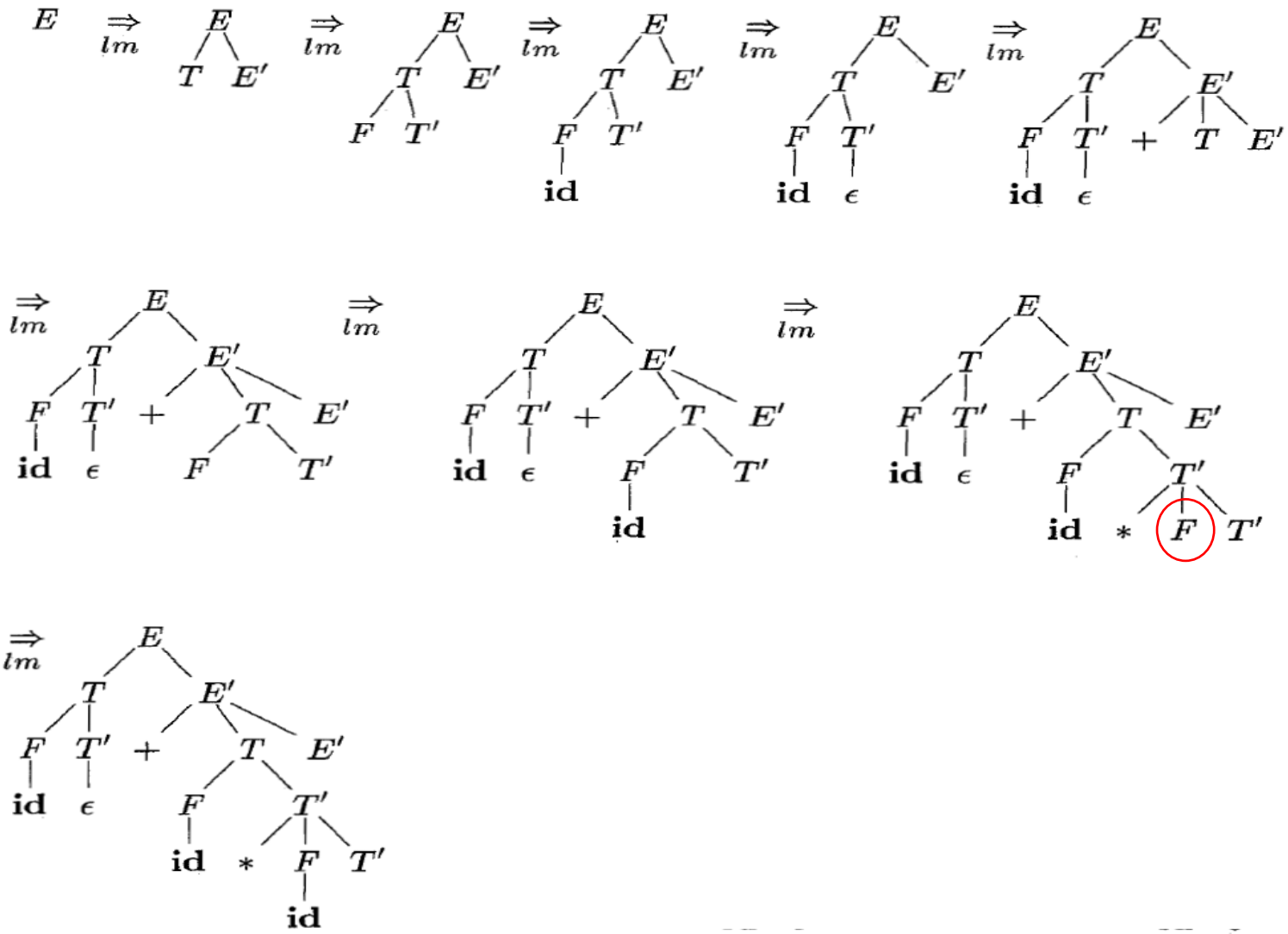
- **Grammar:** $E \rightarrow TE' \quad E' \rightarrow +TE' \mid \epsilon \quad T \rightarrow FT' \quad T' \rightarrow * FT' \mid \epsilon \quad F \rightarrow (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id + id$T'E'$**



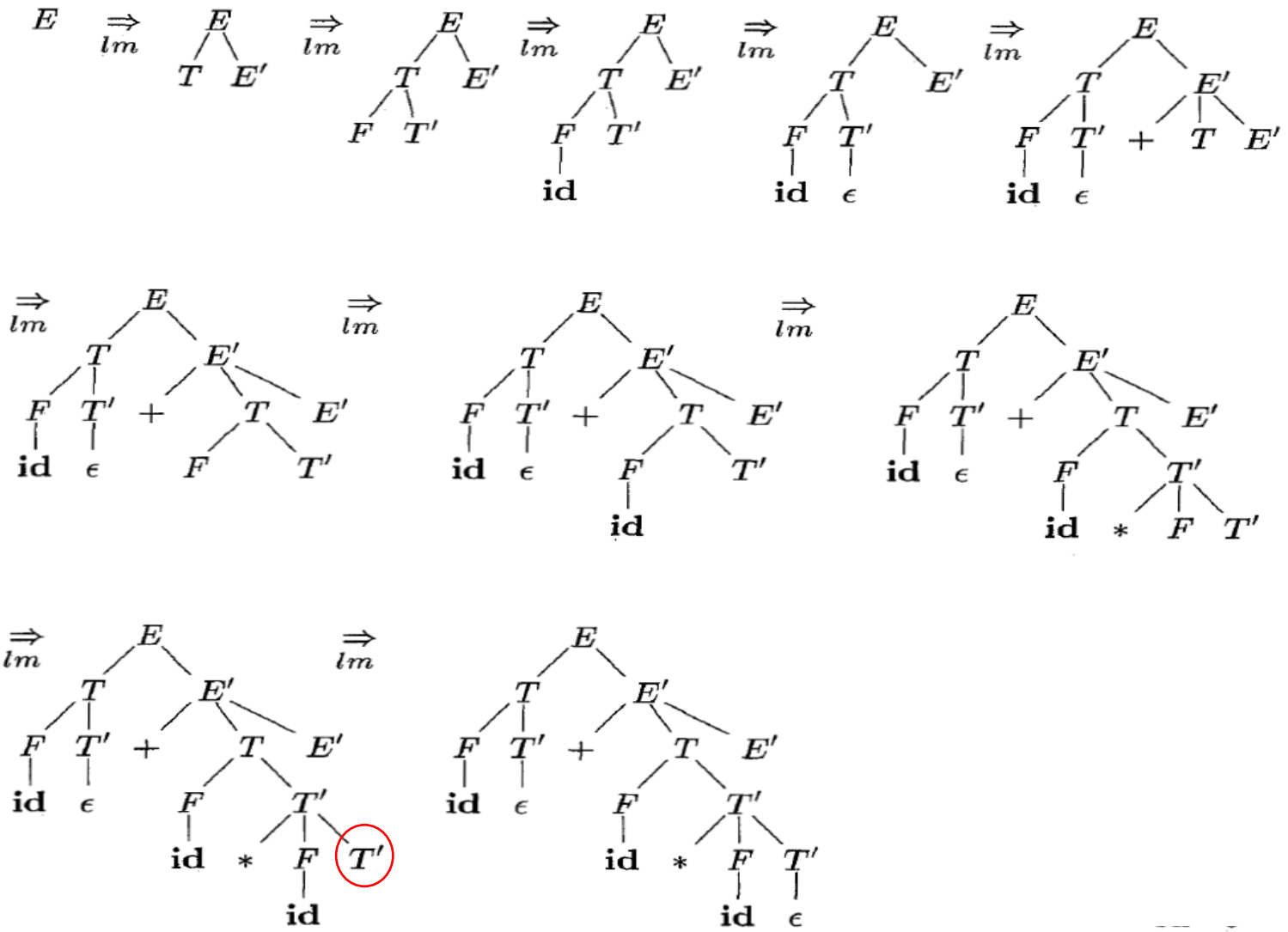
- **Grammar:** $E \to TE'$ $E' \to +TE' \mid \epsilon$ $T \to FT'$ $T' \to * FT' \mid \epsilon$ $F \to (E) \mid id$
- **Input string: id + id * id** **The sentential form after rewrite: id + id * $FT'E'$**



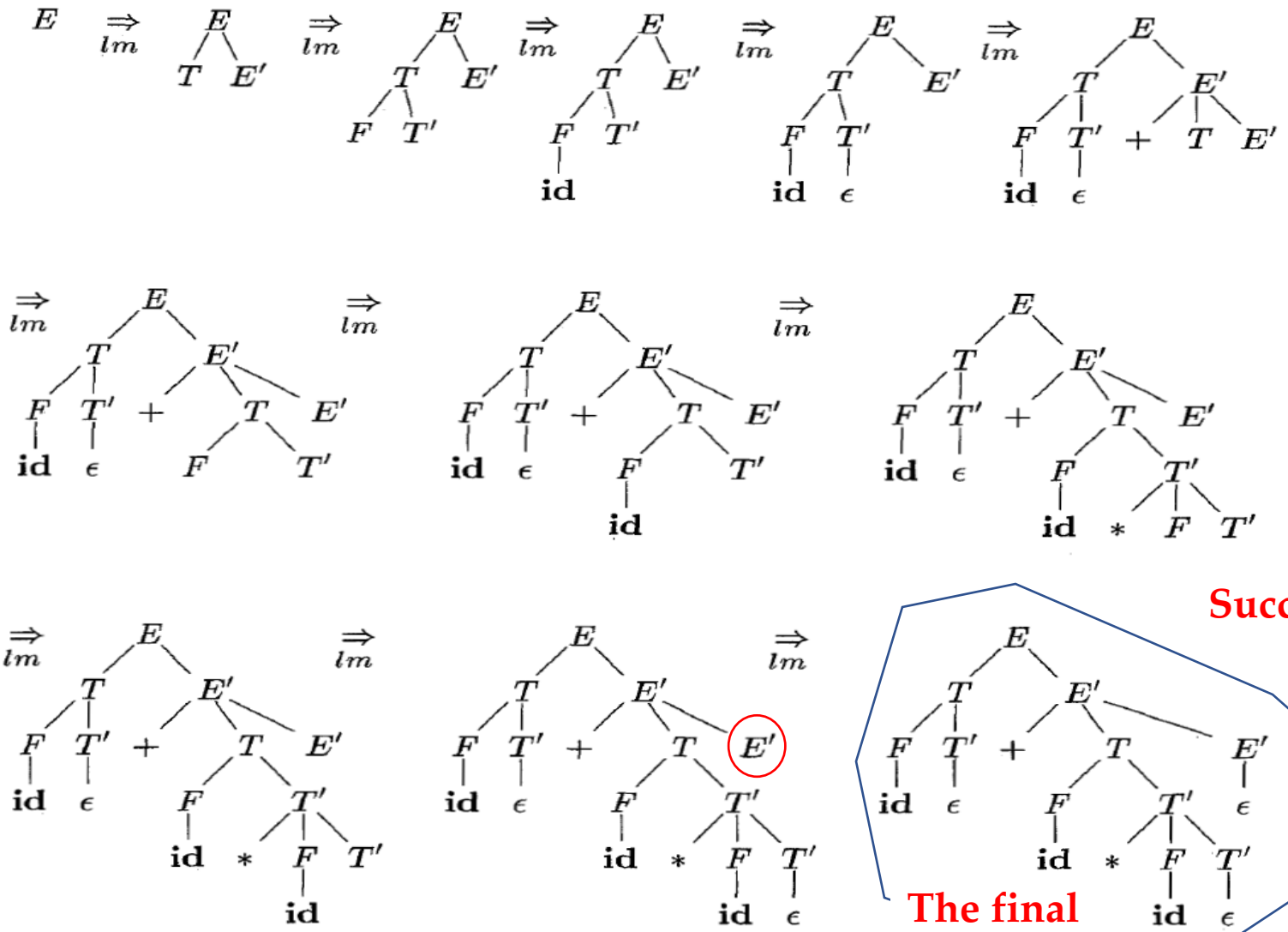
- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to *FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string:** id + id * id **The sentential form after rewrite:** id + id * id $T'E'$



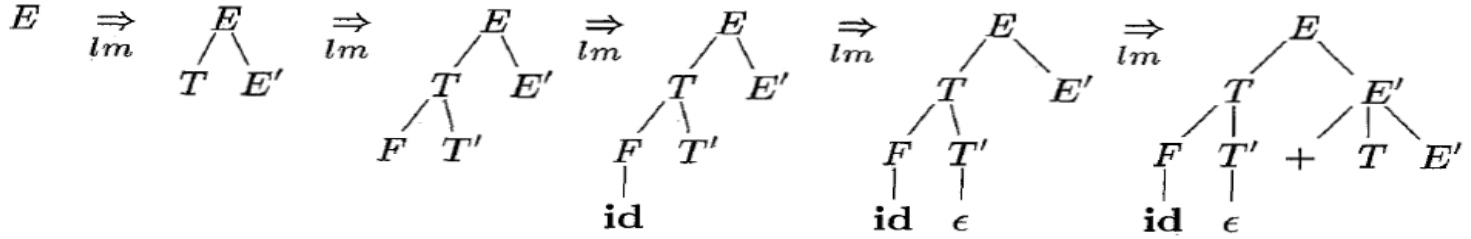
- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to *FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string:** id + id * id **The sentential form after rewrite:** id + id * id E'



- **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: id + id * id**

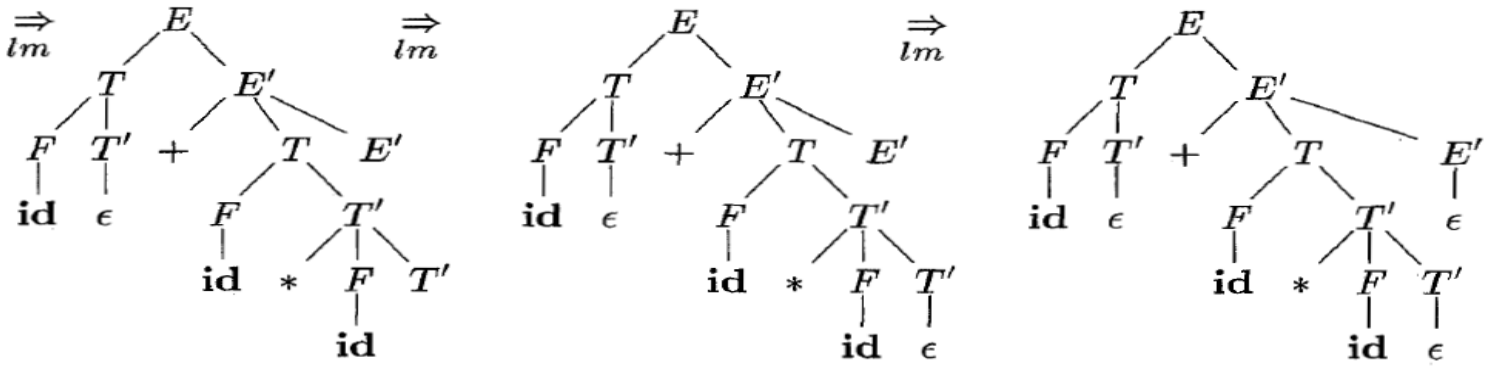


- **Grammar:** $E \to TE' \quad E' \to +TE' \mid \epsilon \quad T \to FT' \quad T' \to * FT' \mid \epsilon \quad F \to (E) \mid id$
- **Input string:** **id + id * id**



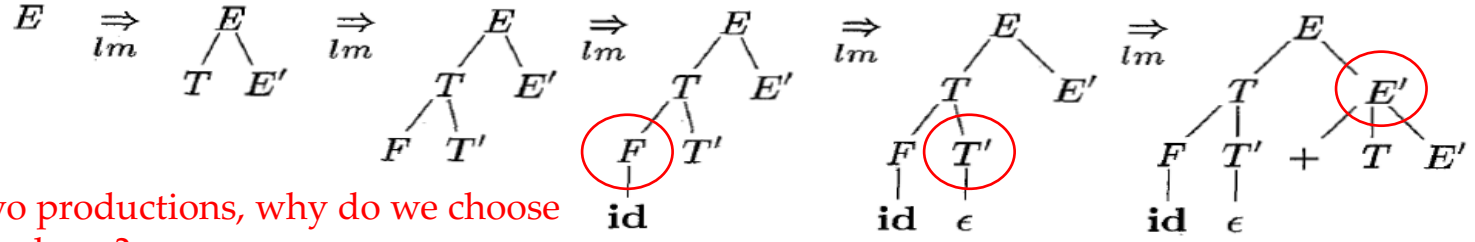
We can make two observations from the example:

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

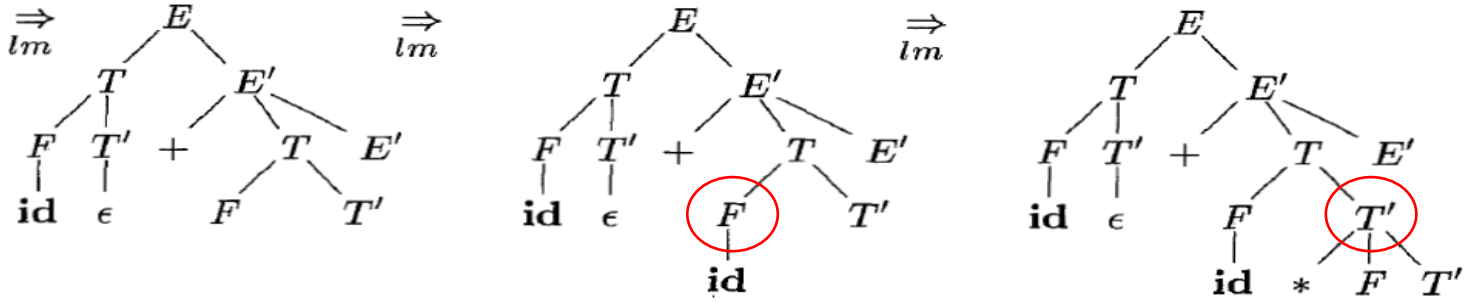


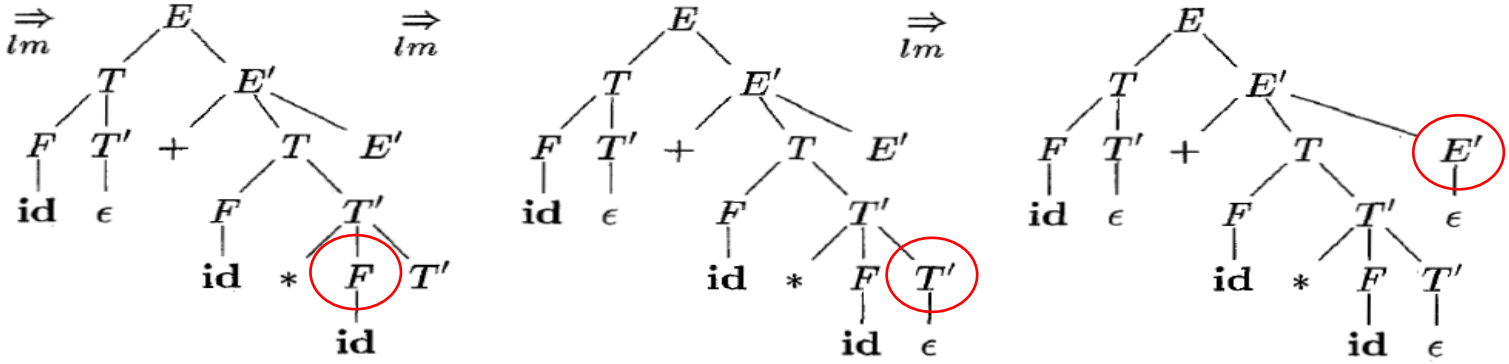
Key decision in top-down parsing: Which production to apply at each step?

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



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

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

id * id

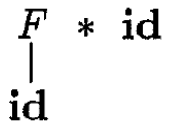
Initially, the tree only contains leaf nodes

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 $F \rightarrow \mathbf{id}$



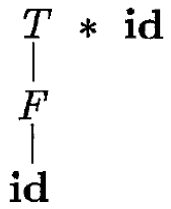
Tree "grows" when reduction happens

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 $T \rightarrow F$



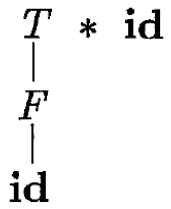
Tree "grows" when reduction happens

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



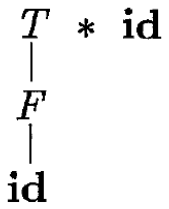
Tree does not change when shift happens

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



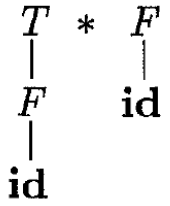
Tree does not change when shift happens

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 $F \rightarrow \mathbf{id}$



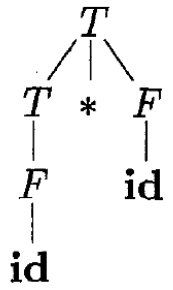
Tree "grows" when reduction happens

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 $T \rightarrow T * F$



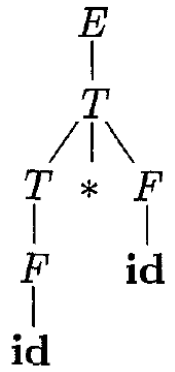
Tree "grows" when reduction happens

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 $\mathbf{id}_1 * \mathbf{id}_2$

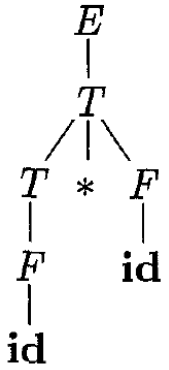
STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow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

Success!!!

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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



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 $F \rightarrow \mathbf{id}$
\$ 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 $E \rightarrow T$
\$ E	\$	accept

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

Rightmost derivation:

$E \Rightarrow T$

$\Rightarrow T * F$

$\Rightarrow T *$ id

$\Rightarrow F *$ id

$\Rightarrow$ id $*$ 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 $\mathbf{id}_1 * \mathbf{id}_2$

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

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 $E \rightarrow T$
\$ E	\$	accept

Key decisions:

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

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
  - Recursive-descent parsing
  - Non-recursive predictive parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

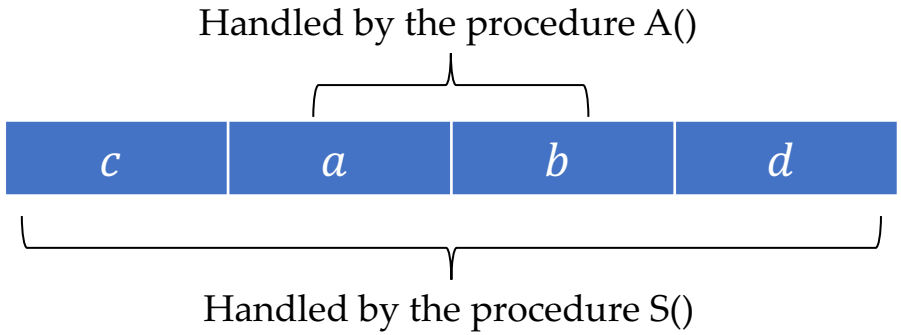
Recursive-Descent Parsing (递归下降的语法分析)

- A recursive-descent parsing program has a set of procedures, one for each nonterminal
  - The procedure for a nonterminal deals with a substring of the input
- Execution begins with the procedure for the start symbol
  - Announce success if the procedure scans the entire input (the start symbol derives the whole input via applying a series of productions)

CFG:

$S \rightarrow cAd$
$A \rightarrow ab$

Input string:



A Typical Procedure for A Nonterminal

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Predict

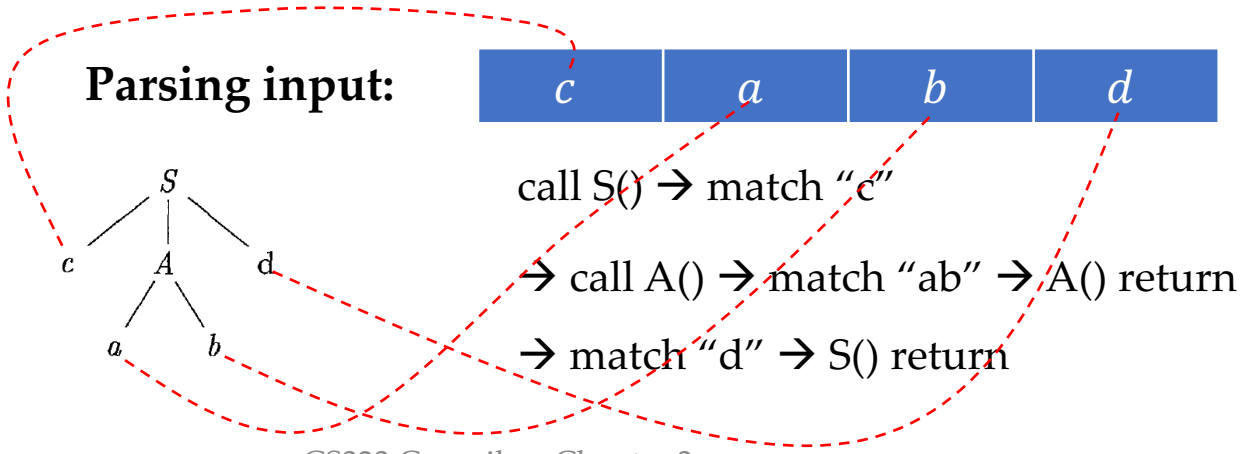
Match

CFG:

$S \rightarrow cAd$

$A \rightarrow ab$

Parsing input:



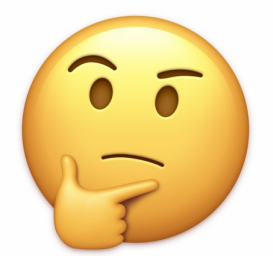
Backtracking (回溯)

```
     void A() {
1)       Choose an A-production, A → X1 X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

If there is a failure at line 7, does this mean that there must be syntax errors?

Backtracking (回溯)

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```



The failure might be caused by a wrong choice of A-production at line 1 !!!

Backtracking (回溯)

- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
     void A() {
1)       Choose an A-production, A → X1 X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

Instead of exploring one *A*-production, we must try each possible production in some order.

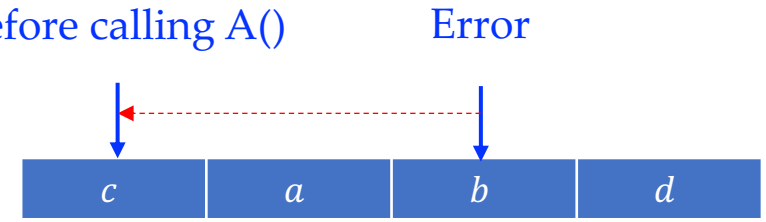
Backtracking (回溯)

- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
     void A() {
1)       Choose an A-production, A → X1 X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

When there is a failure at line 7, return to line 1 and try another A-production.

Backtracking (回溯)



- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

In order to try another A-production, we must reset the input pointer that points to the next symbol to scan (failed trials consume symbols)

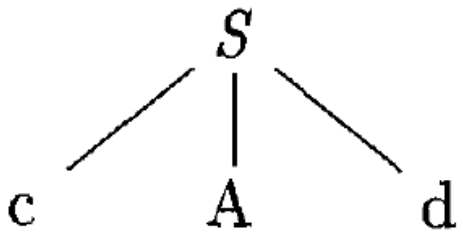
Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$ One more production for A
- Input string: cad

Backtracking Example

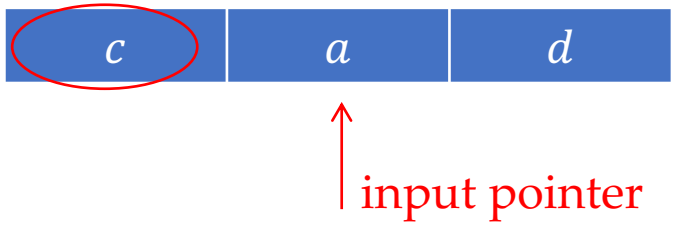
- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

– S has only one production, apply it

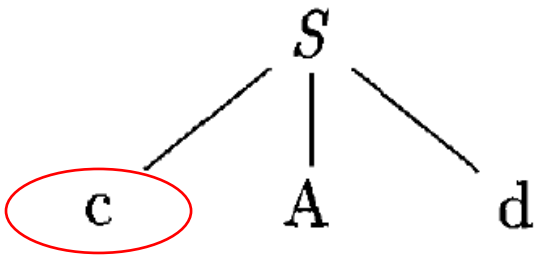


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

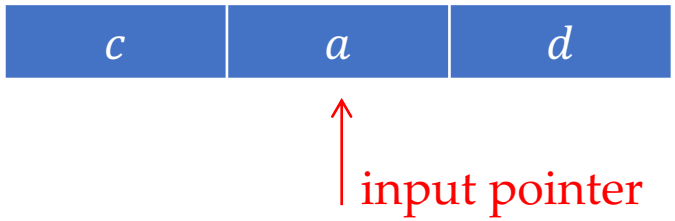


- The leftmost leaf matches c in input
- Advance input pointer

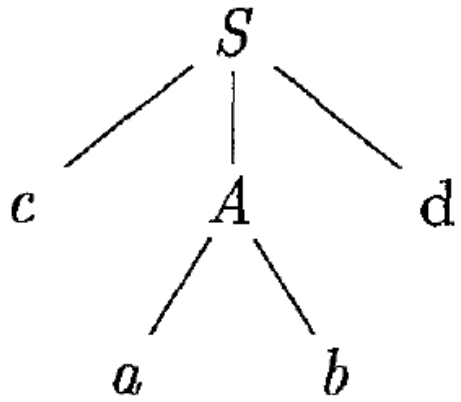


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

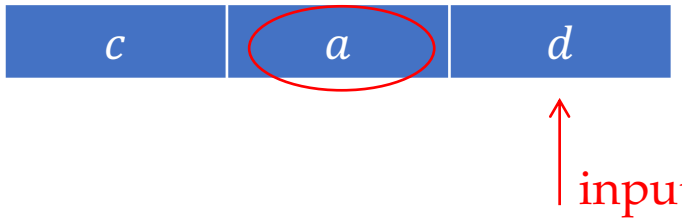


– Expand A using the first production

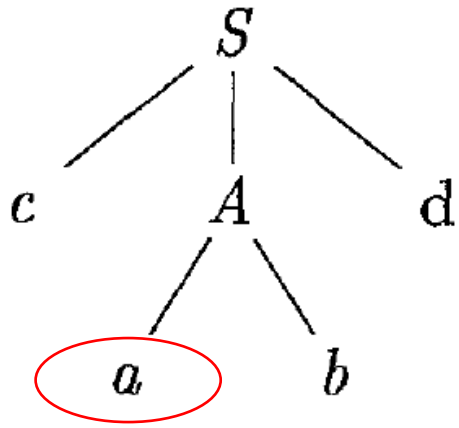


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

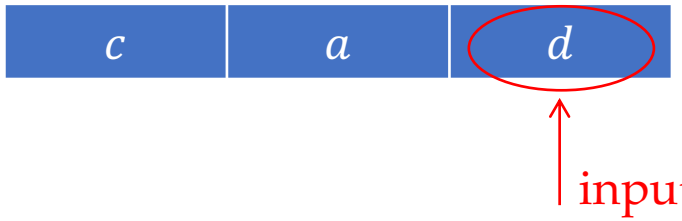


- Leftmost leaf matches a in input
- Advance input pointer

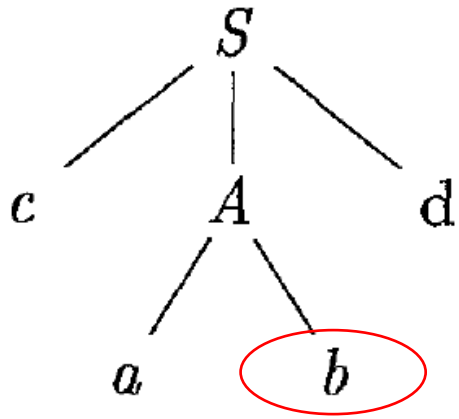


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

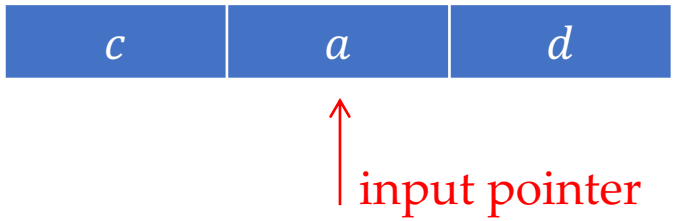


- Symbol mismatch
- Go back to try another A-production

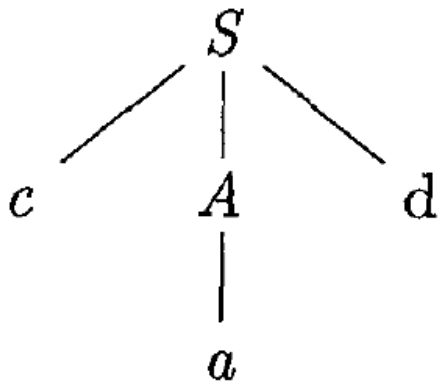


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

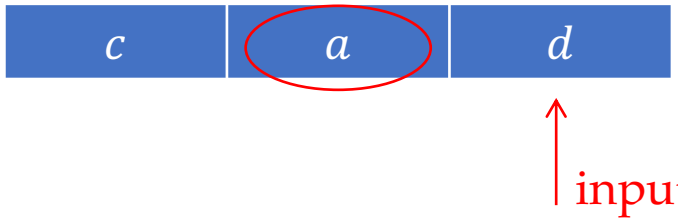


- Reset input pointer
- Expand A using its second production

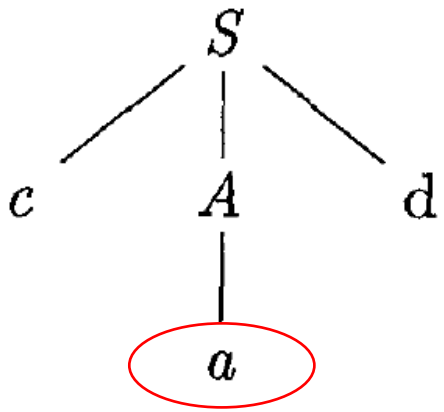


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad



- Leftmost leaf matches a in input
- Advance input pointer

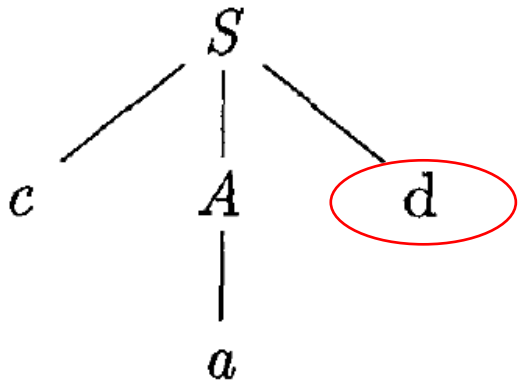


Backtracking Example

- Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- Input string: cad

- The last leaf node matches d in input
- Announce success!

Scanned entire input



The Problem of Left Recursion

Suppose there is only one A-production, $A \rightarrow A\alpha$ ...

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Recursively rewriting A without matching any terminals

If there is left recursion in a CFG, a recursive-descent parser may go into an infinite loop! Revise the CFG before parsing!!!

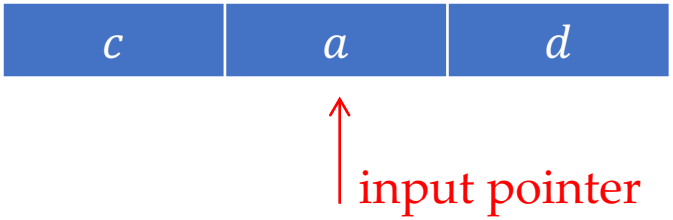
Can We Avoid Backtracking?

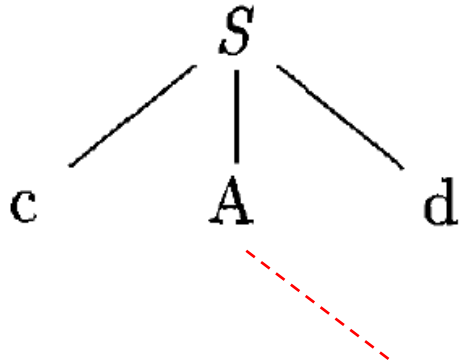
```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Key problem: At line 1, we make ***random choices*** (brute force search)

Can We Avoid Backtracking?

- Grammar: $S \rightarrow cAd \quad A \rightarrow c \mid a$
- Input string: cad





When rewriting A, is it a good idea to choose $A \rightarrow c$?

No! If we look ahead, the next char in the input is a.
$A \rightarrow c$ is obviously a bad choice!!!

Looking Ahead Helps!

- Suppose the input string is $\boldsymbol{x\alpha}\ldots$
- Suppose the current sentential form is $\boldsymbol{xA\beta}$
  - $\boldsymbol{A}$ is a non-terminal; $\boldsymbol{\beta}$ may contain both terminals and non-terminals

If we know the following fact for the productions $\boldsymbol{A \rightarrow \alpha \mid \gamma}$:

- $a \in FIRST(\alpha)$: α derives strings that begin with a
- $a \notin FIRST(\gamma)$: γ derives strings that do not begin with a

*$FIRST(\alpha)$ denotes the set of beginning terminals of strings derived from α

After matching x, which production should we choose to rewrite A? $\boldsymbol{A \rightarrow \alpha}$

Computing FIRST

- **FIRST(X)**, where X is a grammar symbol
  - If X is a terminal, then FIRST(X) = $\{X\}$
  - If X is a nonterminal and $X \rightarrow \epsilon$, then add ϵ to FIRST(X)
  - If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ ($k \geq 1$) is a production
    - If for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1), …, FIRST(Y_{i-1}), then add a to FIRST(X)
    - If ϵ is in all of FIRST(Y_1), …, FIRST(Y_k), then add ϵ to FIRST(X)

Computing FIRST Cont.

- **FIRST($X_1X_2 \ldots X_n$)**, where $X_1X_2 \ldots X_n$ is a string of grammar symbols
  - Add all non-ϵ symbols of FIRST(X_1) to FIRST($X_1X_2 \ldots X_n$)
  - If ϵ is in FIRST(X_1), add non-ϵ symbols of FIRST(X_2) to FIRST($X_1X_2 \ldots X_n$)
  - If ϵ is in FIRST(X_1) and FIRST(X_2), add non-ϵ symbols of FIRST(X_3) to FIRST($X_1X_2 \ldots X_n$)
  - …
  - If ϵ is in FIRST(X_i) for all i, add ϵ to FIRST($X_1X_2 \ldots X_n$)

FIRST Example

- **Grammar**
  - $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \mathbf{id}$
- **FIRST sets**
  - $\text{FIRST}(F) = \{(, \mathbf{id}\}$
  - $\text{FIRST}(T) = \text{FIRST}(F) = \{(, \mathbf{id}\}$
  - $\text{FIRST}(E) = \text{FIRST}(T) = \{(, \mathbf{id}\}$
  - $\text{FIRST}(E') = \{+, \epsilon\}$ $\quad \text{FIRST}(T') = \{*, \epsilon\}$
  - $\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \mathbf{id}\}$
  - …

Strings derived from F or T must start with (or **id**

Looking Ahead Helps Cont.

- Suppose the input string is $\boldsymbol{xa}\ldots$
- Suppose the current sentential form is $\boldsymbol{xA\beta}$
  - $\boldsymbol{A}$ is a non-terminal; $\boldsymbol{\beta}$ may contain both terminals and non-terminals

If we know that for the production $\boldsymbol{A \rightarrow \alpha}$, $\epsilon \in FIRST(\alpha)$, can we choose the production to rewrite A?

*$\epsilon \in FIRST(\alpha)$ means that rewriting A to α may result in an empty string (recall when we add ϵ to the *FIRST* set)

Yes, only if β can derive strings beginning with a, that is, A can be followed by a in some sentential forms (i.e., $\boldsymbol{a \in FOLLOW(A)}$)

Computing FOLLOW

- **Computing FOLLOW set for all nonterminals**
  - Add \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker
  - Apply the rules below, until all FOLLOW sets do not change
    1. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B)
    2. If there is a production $A \rightarrow \alpha B$ (or $A \rightarrow \alpha B \beta$ and FIRST(β) contains ϵ) then everything in FOLLOW(A) is in FOLLOW(B)

By definition, ϵ will not appear in any FOLLOW set

FOLLOW Example

- **Grammar**
  - $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \mathbf{id}$
- **FOLLOW sets**
  - $\text{FOLLOW}(E) = \{\$,)\}$
  - $\text{FOLLOW}(E') = \{\$,)\}$
  - $\text{FOLLOW}(T) = \{+, \$,)\}$
  - $\text{FOLLOW}(T') = \{+, \$,)\}$
  - $\text{FOLLOW}(F) = \{*, +, \$,)\}$

- \$ is always in $\text{FOLLOW}(E)$
- Everything in FIRST()) except ϵ is in $\text{FOLLOW}(E)$

FOLLOW Example

- **Grammar**
  - $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \mathbf{id}$
- **FOLLOW sets**
  - $\text{FOLLOW}(E) = \{\$,)\}$
  - $\text{FOLLOW}(E') = \{\$,)\}$
  - $\text{FOLLOW}(T) = \{+, \$,)\}$
  - $\text{FOLLOW}(T') = \{+, \$,)\}$
  - $\text{FOLLOW}(F) = \{*, +, \$,)\}$

- Everything in $\text{FIRST}(E')$ except ϵ is in $\text{FOLLOW}(T)$
- Since $E' \rightarrow \epsilon$, everything in $\text{FOLLOW}(E)$ and $\text{FOLLOW}(E')$ is in $\text{FOLLOW}(T)$

A Quick Summary

Why Do We Compute FIRST & FOLLOW?

- For a production $head \rightarrow body$, when we are trying to rewrite $head$, if we know $FIRST(body)$, that is, what terminals can strings derived from body start with, we can decide whether to choose $head \rightarrow body$ by looking at the next input symbol.
  - If the next input symbol is in $FIRST(body)$, the production may be a good choice.

- For a production $head \rightarrow \epsilon$ (or $head$ can derive ϵ in some steps), when we are trying to rewrite $head$, if we know $FOLLOW(head)$, that is, what terminals can follow $head$ in various sentential forms, we can decide whether to choose $head \rightarrow \epsilon$ by looking at the next input symbol.
  - If the next input symbol is in $FOLLOW(head)$, the production may be a good choice.

LL(1) Grammars

- Recursive-descent parsers needing no backtracking can be constructed for a class of grammars called **LL(1)**
  - 1st L: scanning the input from left to right
  - 2nd L: producing a leftmost derivation (top-down parsing)
  - 1: using one input symbol of lookahead at each step to make parsing decision

LL(1) Grammars Cont.

A grammar G is LL(1) if and only if for any two distinct productions $A \rightarrow \alpha \mid \beta$, the following conditions hold:

1. There is no terminal a such that α and β derive strings beginning with a
2. At most one of α and β can derive the empty string
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A) and vice versa

*** The three conditions essentially rule out the possibility of applying both productions so that there is a unique choice of production at each "predict" step by looking at the next input symbol**

More formally:

1. $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$ (conditions 1-2 above)
2. If $\epsilon \in \text{FIRST}(\beta)$, then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$ and vice versa

LL(1) Grammars Cont.

- For LL(1) grammars, during recursive-descent parsing, the proper production to apply for a nonterminal can be selected by looking only at the current input symbol

Grammar: *stmt* → **if**(**expr**) *stmt* **else** *stmt* | **while**(**expr**) *stmt* | **a**

① ② ③

Parsing steps for input: **if**(**expr**) **while**(**expr**) **a else a**

1. Rewrite the start symbol *stmt* with ①: **if**(**expr**) *stmt* **else** *stmt*
2. Rewrite the leftmost *stmt* with ②: **if**(**expr**) **while**(**expr**) *stmt* **else** *stmt*
3. Rewrite the leftmost *stmt* with ③: **if**(**expr**) **while**(**expr**) **a else** *stmt*
4. Rewrite the leftmost *stmt* with ③: **if**(**expr**) **while**(**expr**) **a else a**

Parsing Table (预测分析表)

- We can build parsing tables for recursive-descent parsers (**LL parsers**)
- A predictive **parsing table** is a two-dimensional array that determines which production the parser should choose when it sees a nonterminal A and a symbol a on its input stream
- The parsing table of an LL(1) parser has no entries with multiple productions

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Parsing Table Construction

The following algorithm can be applied to any CFG

- **Input:** Grammar G **Output:** Parsing table M
- **Method:**
  - For each production $A \rightarrow \alpha$ of G, do the following:
    - For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$
    - If ϵ is in FIRST(α), then for each terminal b (including the right endmarker \$) in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$
  - Set all empty entries in the table to **error**

Fill the table entries so that when rewriting A, we know what production to choose by checking the next input symbol

Parsing Table Construction Example

- **Grammar**
  - $E \rightarrow TE'$ $\quad E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$ $\quad T' \rightarrow * FT' \mid \epsilon$ $\quad F \rightarrow (E) \mid \mathbf{id}$
- **FIRST sets:** E, T, F: {(, **id**} $\quad E'$: {+, ϵ} $\quad T'$: {*, ϵ}
- **FOLLOW sets:** E, E': {$,)} $\quad T, T'$: {+, $,)} $\quad F$: {*, +, $,)}

NON-TERMINAL	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**			$F \rightarrow (E)$		

For $E \rightarrow TE'$:

FIRST(TE')

= FIRST(T)

= {(, **id**}

Parsing Table Construction Example

- **Grammar**
  - $E \to TE'$ $\quad E' \to +TE' \mid \epsilon$
  - $T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$
- **FIRST sets:** E, T, F: {(, **id**} $\quad E'$: {+, ϵ} $\quad T'$: {*, ϵ}
- **FOLLOW sets:** E, E': {$,)} $\quad T, T'$: {+, $,)} $\quad F$: {*, +, $,)}

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

For $E' \to \epsilon$:

ϵ in FIRST(ϵ)

FOLLOW(E') = {$,)}

Conflicts in Parsing Tables

- **Grammar:** $S \rightarrow \mathbf{i}E\mathbf{t}SS' \mid \mathbf{a}$ $\quad S' \rightarrow \mathbf{e}S \mid \epsilon$ $\quad E \rightarrow \mathbf{b}$
  - FIRST($\mathbf{e}S$) $= \{\mathbf{e}\}$, so we add $S' \rightarrow \mathbf{e}S$ to $M[S', \mathbf{e}]$
  - FOLLOW(S') $= \{\$, \mathbf{e}\}$, so we add $S' \rightarrow \epsilon$ to $M[S', \mathbf{e}]$

NON-TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

- LL(1) grammar is never ambiguous.
- This grammar is not LL(1). The language has no LL(1) grammar !!!

Recursive-Descent Parsing for LL(1) Grammars

```
     void A() {
1)       Choose an A-production, A → X1 X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

Replace line 1 with: Choose A-production according to the parse table

- Assume input symbol is a, then the choice is the production in $M[A, a]$

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
  - Recursive-descent parsing
  - Non-recursive predictive parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Recall Recursive-Descent Parsing

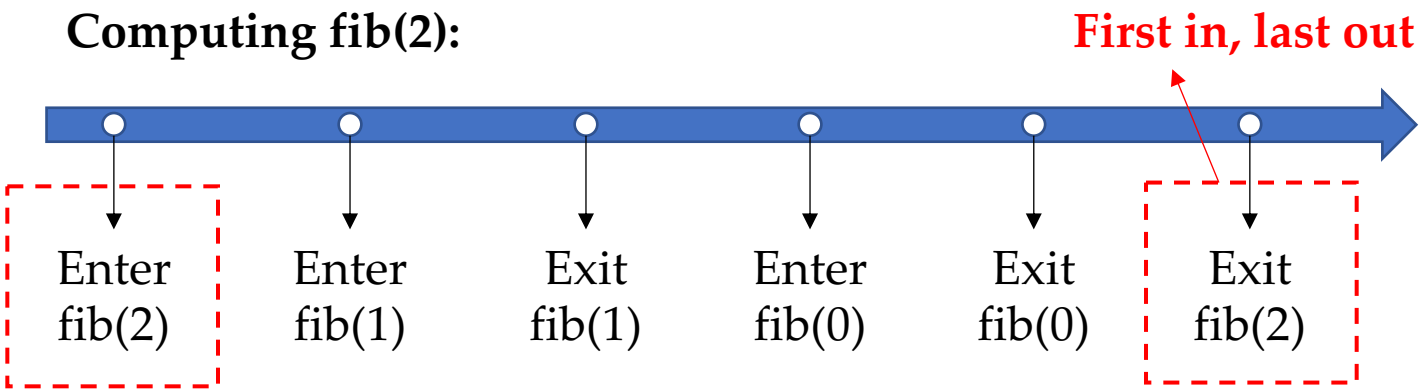
```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Recursive-descent parsing has recursive calls.

Can we design a non-recursive parser?

How Is Recursion Handled?

```
int fib(int n) {
    if(n <= 1) return n;
    else {
        int a = fib(n-1) + fib(n-2);
        return a;
    }
}
```



Non-Recursive Predictive Parsing

- A non-recursive predictive parser can be built by **explicitly maintaining a stack** (not implicitly via recursive calls)
  - **Input buffer** contains the string to be parsed, ending with $
  - **Stack** holds a sequence of grammar symbols with $ at the bottom. Initially, the stack contains only $ and the start symbol *S* on top of $

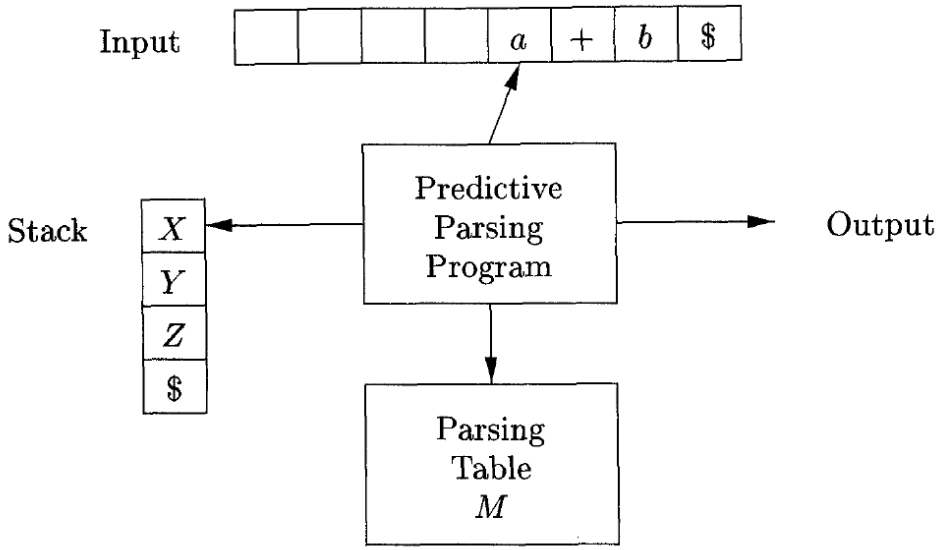
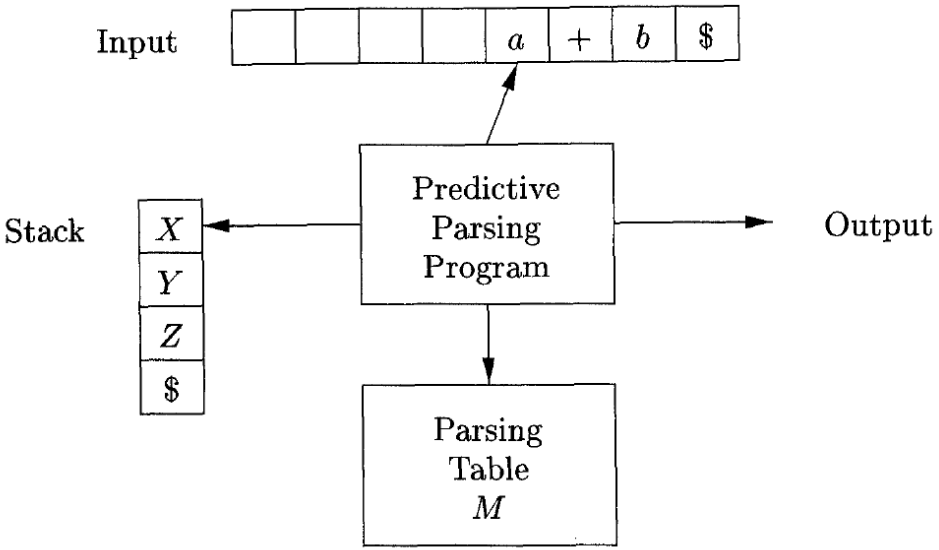


Table-Driven Predictive Parsing

- **Input:** A string ω and a parsing table M for grammar G
- **Output:** If ω is in $L(G)$, a leftmost derivation of ω (input buffer and stack are both empty); otherwise, an error indication



Initially, the input buffer contains $\omega\$$.

The start symbol S of G is on top of the stack, above \$.

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id + id * id$**	

Example

$E \to TE' \quad E' \to +TE' \mid \epsilon$

$T \to FT' \quad T' \to *FT' \mid \epsilon \quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id + id * id$**	
	$TE'\$$	**id + id * id$**	output $E \to TE'$
	$FT'E'\$$	**id + id * id$**	output $T \to FT'$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id + id * id$**	
	$TE'\$$	**id + id * id$**	output $E \to TE'$
	$FT'E'\$$	**id + id * id$**	output $T \to FT'$
	id $T'E'\$$	**id + id * id$**	output $F \to \mathbf{id}$
id	$T'E'\$$	**+ id * id$**	match **id**

Example

$E \to TE' \quad E' \to +TE' \mid \epsilon$

$T \to FT' \quad T' \to *FT' \mid \epsilon \quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id + id * id$**	
	$TE'\$$	**id + id * id$**	output $E \to TE'$
	$FT'E'\$$	**id + id * id$**	output $T \to FT'$
	id $T'E'\$$	**id + id * id$**	output $F \to \mathbf{id}$
id	$T'E'\$$	**+ id * id$**	match **id**
id	$E'\$$	**+ id * id$**	output $T' \to \epsilon$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id + id * id$**	
	$TE'\$$	**id + id * id$**	output $E \to TE'$
	$FT'E'\$$	**id + id * id$**	output $T \to FT'$
	id $T'E'\$$	**id + id * id$**	output $F \to$ **id**
id	$T'E'\$$	**+ id * id$**	match **id**
id	$E'\$$	**+ id * id$**	output $T' \to \epsilon$
id	$+\ TE'\$$	**+ id * id$**	output $E' \to +\ TE'$

Example

$E \to TE'$ $\quad E' \to +TE' \mid \epsilon$

$T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

MATCHED	STACK	INPUT	ACTION
	$E\$$	id + id * id$	
	$TE'\$$	id + id * id$	output $E \to TE'$
	$FT'E'\$$	id + id * id$	output $T \to FT'$
	id $T'E'\$$	id + id * id$	output $F \to$ id
id	$T'E'\$$	+ id * id$	match id
id	$E'\$$	+ id * id$	output $T' \to \epsilon$
id	$+ TE'\$$	+ id * id$	output $E' \to + TE'$
id +	$TE'\$$	id * id$	match +
...	...	...	...
id + id * id	$\$$	$\$$	output $E' \to \epsilon$

There are eight more steps before accepting.

The parser announce success when both stack and input are empty.

Example

$E \to TE'$ $E' \to +TE' \mid \epsilon$

$T \to FT'$ $T' \to *FT' \mid \epsilon$ $F \to (E) \mid \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Input:

id + id * id

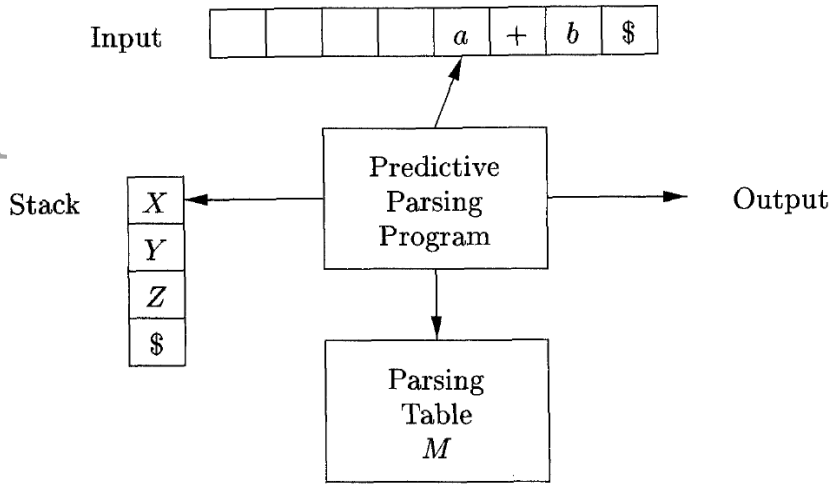
MATCHED	STACK	INPUT	ACTION
	$E\$$	id + id * id$	
	$TE'\$$	id + id * id$	output $E \to TE'$
	$FT'E'\$$	id + id * id$	output $T \to FT'$
	id $T'E'\$$	id + id * id$	output $F \to \mathbf{id}$
id	$T'E'\$$	+ id * id$	match id
id	$E'\$$	+ id * id$	output $T' \to \epsilon$
id	$+ TE'\$$	+ id * id$	output $E' \to + TE'$
id +	$TE'\$$	id * id$	match +
...	...	...	...
id + id * id	$\$$	$	output $E' \to \epsilon$

Leftmost derivation

Matched part + Stack content (from top to bottom) = A left-sentential form

总是最左句型

Parsing Algorithm



1. let a be the first symbol of ω;
2. let X be the top stack symbol;
3. while ($X \neq \$$) { /* stack is not empty */
4. if ($X = a$) pop the stack and let a be the next symbol of ω;
5. else if (X is a terminal) *error*(); /* X can only match a, cannot be another terminal */
6. else if ($M[X, a]$ is an error entry) *error*();
7. else if ($M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$) {
8. output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$;
9. pop the stack;
10. push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top; /* order is critical */
11. }
12. let X be the top stack symbol;
13. }

Outline

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

Shift-Reduce Parsing (Revisit)

- Bottom-up parsing can be seen as a process of "reducing" a string ω to the start symbol of the grammar (a reverse process of derivation)
- Shift-reduce parsing is a general style of bottom-up parsing in which:
  - A **stack** holds grammar symbols
  - An **input buffer** holds the rest of the string to be parsed
  - The stack content (from bottom to top) and the input buffer content form a right-sentential form (assuming no errors)

Shift-Reduce Parsing (Revisit)

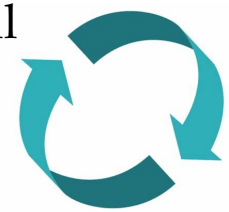
Initial status:

STACK	INPUT
$	$\omega$$

Actions:
Shift
Reduce
Accept
Error

Shift-reduce process:

- The parser **shifts** zero or more input symbols onto the stack, until it is ready to reduce a string β on top of the stack
- **Reduce** β to the head of the appropriate production



The parser repeats the above cycle until it has detected an error or the stack contains the start symbol and input is empty

The Challenge (Revisit)

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
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

Why shifting $*$ instead of reducing T ?

Generally, when to shift/reduce? How to reduce (choosing which production)?

Outline

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

- Simple LR (SLR)
- Canonical LR (CLR)
- Look-ahead LR (LALR)
- Error Recovery (Lab)

LR Parsing (LR语法分析技术)

- **LR(k) parsers:** the most prevalent type of bottom-up parsers
  - L: left-to-right scan of the input
  - R: construct a rightmost derivation in reverse
  - k: use k input symbols of lookahead in making parsing decisions

- LR(0) and LR(1) parsers are of practical interest
  - When $k \geq 2$, the parser becomes too complex to construct (parsing table will be too huge to manage)

Advantages of LR Parsers

- Table-driven (like non-recursive LL parsers) and powerful
  - Although it is too much work to construct an LR parser by hand, there are parser generators to construct parsing tables automatically
  - Comparatively, LL parsers tend to be easier to write by hand, but less powerful (handle fewer grammars)
- LR-parsing is the most general nonbacktracking shift-reduce parsing method known
- LR parsers can be constructed to recognize virtually all programming language constructs for which CFGs can be written
- LR grammars can describe more languages than LL grammars
  - Recall the stringent conditions for a grammar to be LL(1)

When to Shift/Reduce?

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 $E \rightarrow T$
$ E	$	accept

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

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

How does a shift/reduce parser know that *T* on stack top is a bad choice for reduction (the right action is to shift)?

LR(0) Items (LR(0)项)

- An LR parser makes shift-reduce decisions by **maintaining states** to keep track of what have been seen during parsing
- An ***LR(0) item*** (item for short) is a production with a dot at some position of the body, indicating how much we have seen at a given time point in the parsing process
  - $A \rightarrow \cdot XYZ$ $A \rightarrow X \cdot YZ$ $A \rightarrow XY \cdot Z$ $A \rightarrow XYZ \cdot$
  - $A \rightarrow X \cdot YZ$: we have just seen on the input a string derivable from X and we hope to see a string derivable from YZ next
  - The production $A \rightarrow \epsilon$ generates only one item $A \rightarrow \cdot$
- **States:** sets of LR(0) items (LR(0)项集)

Canonical LR(0) Collection

- One collection of states (i.e., sets of LR(0) items), called the ***canonical* LR(0) collection** (LR(0)项集规范族), provides the basis for constructing a DFA to make parsing decisions

- To construct canonical LR(0) collection for a grammar, we need to define:
  - An augmented grammar (增广文法)
  - Two functions: (1) CLOSURE of item sets (项集闭包) and (2) GOTO

Augmented Grammar

- Augmenting a grammar G with start symbol S
  - Introduce a new start symbol S' to take the role of S
  - Add a new production $S' \rightarrow S$
- Obviously, $L(G) = L(G')$
- **Benefit:** With the augmentation, acceptance occurs when and only when the parser is about to reduce by $S' \rightarrow S$
  - Otherwise, acceptance could occur at many points since there may be multiple S-productions

Closure of Item Sets

- If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules
  1. Initially, add every item in I to CLOSURE(I)
  2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\, \gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I)

- **Intuition:** $A \to \alpha \cdot B\beta$ indicates that we hope to see a substring derivable from $B\beta$, which has a prefix derivable from B. Therefore, we add items for all B-productions.

Algorithm for CLOSURE(I)

// the earlier natural language description is already clear enough

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

Example

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

- Augmented grammar
  - $E' \rightarrow E \qquad E \rightarrow E + T \mid T \qquad T \rightarrow T * F \mid F \qquad F \rightarrow (E) \mid \mathbf{id}$

- Computing the closure of the item set $\{[E' \rightarrow \cdot\, E]\}$
  - Initially, $[E' \rightarrow \cdot\, E]$ is in the closure
  - Add $[E \rightarrow \cdot\, E + T]$ and $[E \rightarrow \cdot\, T]$ to the closure
  - Add $[T \rightarrow \cdot\, T * F]$ and $[T \rightarrow \cdot\, F]$ to the closure
  - Add $[F \rightarrow \cdot\, (E)]$ and $[F \rightarrow \cdot\, \mathbf{id}]$ and reach fixed point

- $[E' \rightarrow \cdot\, E]$
- $[E \rightarrow \cdot\, E + T]$
- $[E \rightarrow \cdot\, T]$
- $[T \rightarrow \cdot\, T * F]$
- $[T \rightarrow \cdot\, F]$
- $[F \rightarrow \cdot\, (E)]$
- $[F \rightarrow \cdot\, \mathbf{id}]$

The Function GOTO

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

- **GOTO(I, X)**, where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ where $[A \rightarrow \alpha \cdot X\beta]$ is in I
  - $CLOSURE(\{[A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X\beta] \in I\})$
- **Example:** Computing GOTO$(I, +)$ for $I = \{[E' \rightarrow E \cdot], [E \rightarrow E \cdot +T]\}$
  - There is only one item $[E \rightarrow E \cdot +T]$, in which + follows ·
  - Then compute the CLOSURE$(\{[E \rightarrow E +\cdot T]\})$, which contains:
    - $[E \rightarrow E +\cdot T]$
    - $[T \rightarrow\cdot T * F]$, $[T \rightarrow\cdot F]$
    - $[F \rightarrow\cdot (E)]$, $[F \rightarrow\cdot \mathbf{id}]$

Constructing Canonical LR(0) Collection

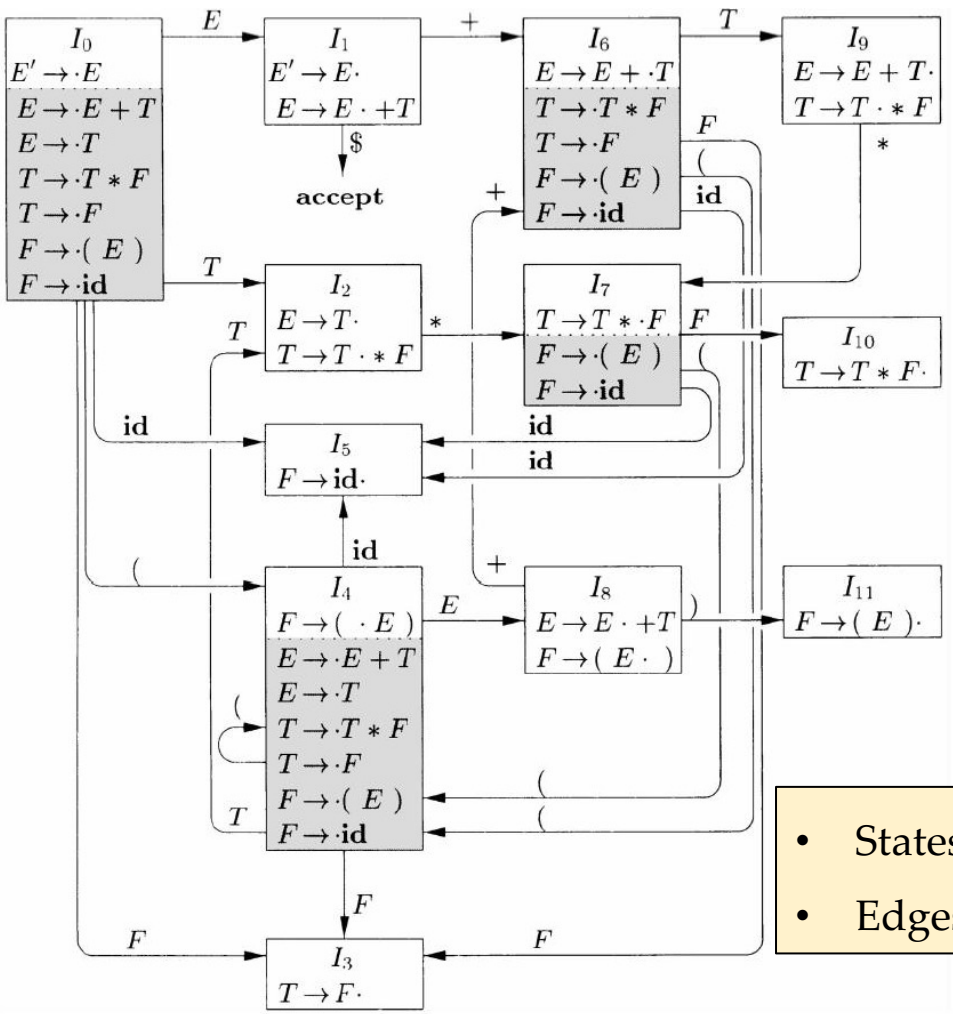
```
void items(G') {
    C = {CLOSURE({[S' → ·S]})};
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Initially, there is just one item set (i.e., the initial state)

Iteratively find all possible GOTO targets (states in the automaton for parsing)

Example

The canonical LR(0) collection for the grammar below is $\{I_0, I_1, \ldots, I_{11}\}$



(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

(6) $F \to \mathbf{id}$

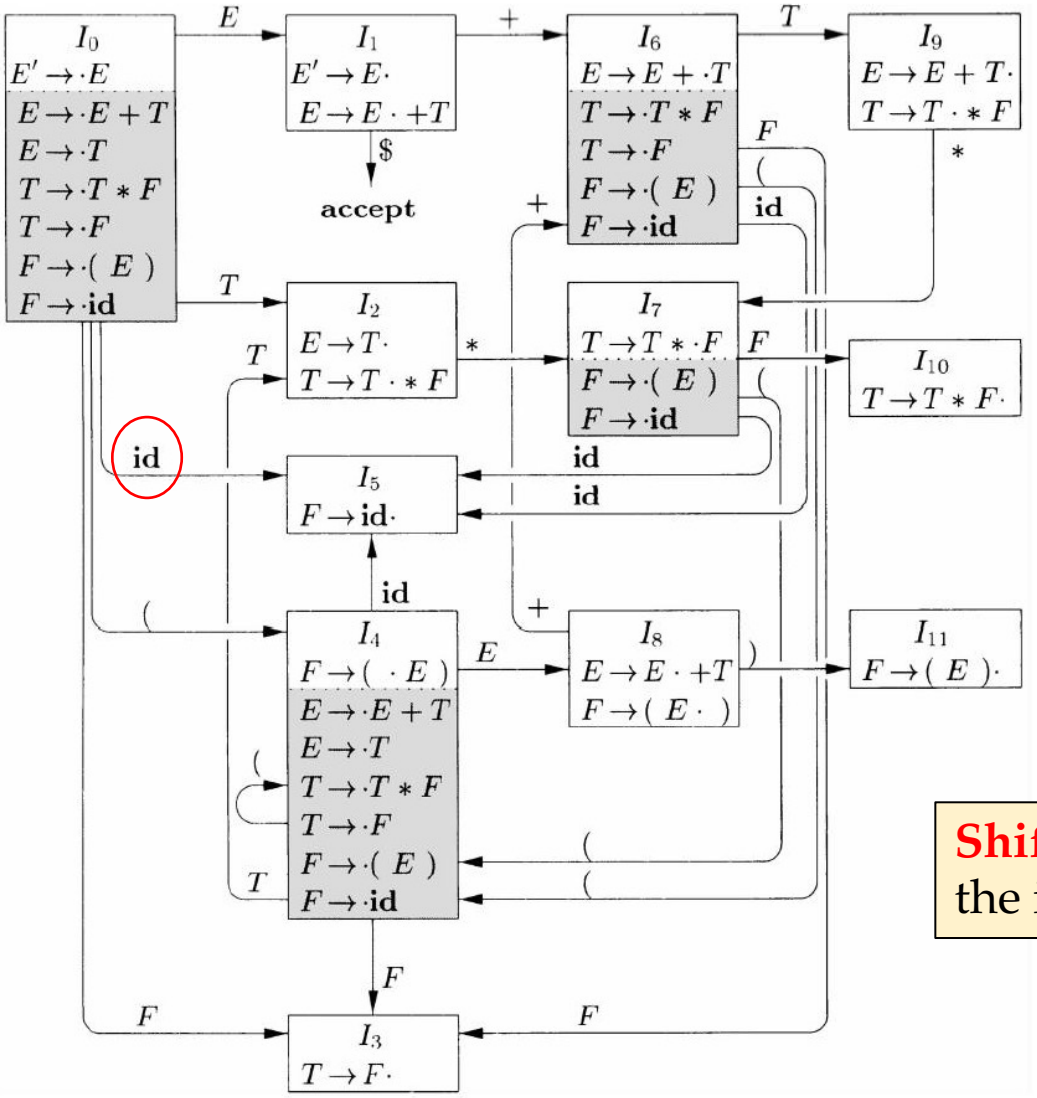
- States are constructed by CLOSURE function
- Edges are constructed by GOTO function

LR(0) Automaton

- The central idea behind "Simple LR", or SLR, is constructing the LR(0) automaton from the grammar
  - The states are the item sets in the canonical LR(0) collection
  - The transitions are given by the GOTO function
  - The start state is $\text{CLOSURE}(\{[S' \rightarrow \cdot\, S]\})$

LR(0) automaton can effectively help make shift-reduce decisions.

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

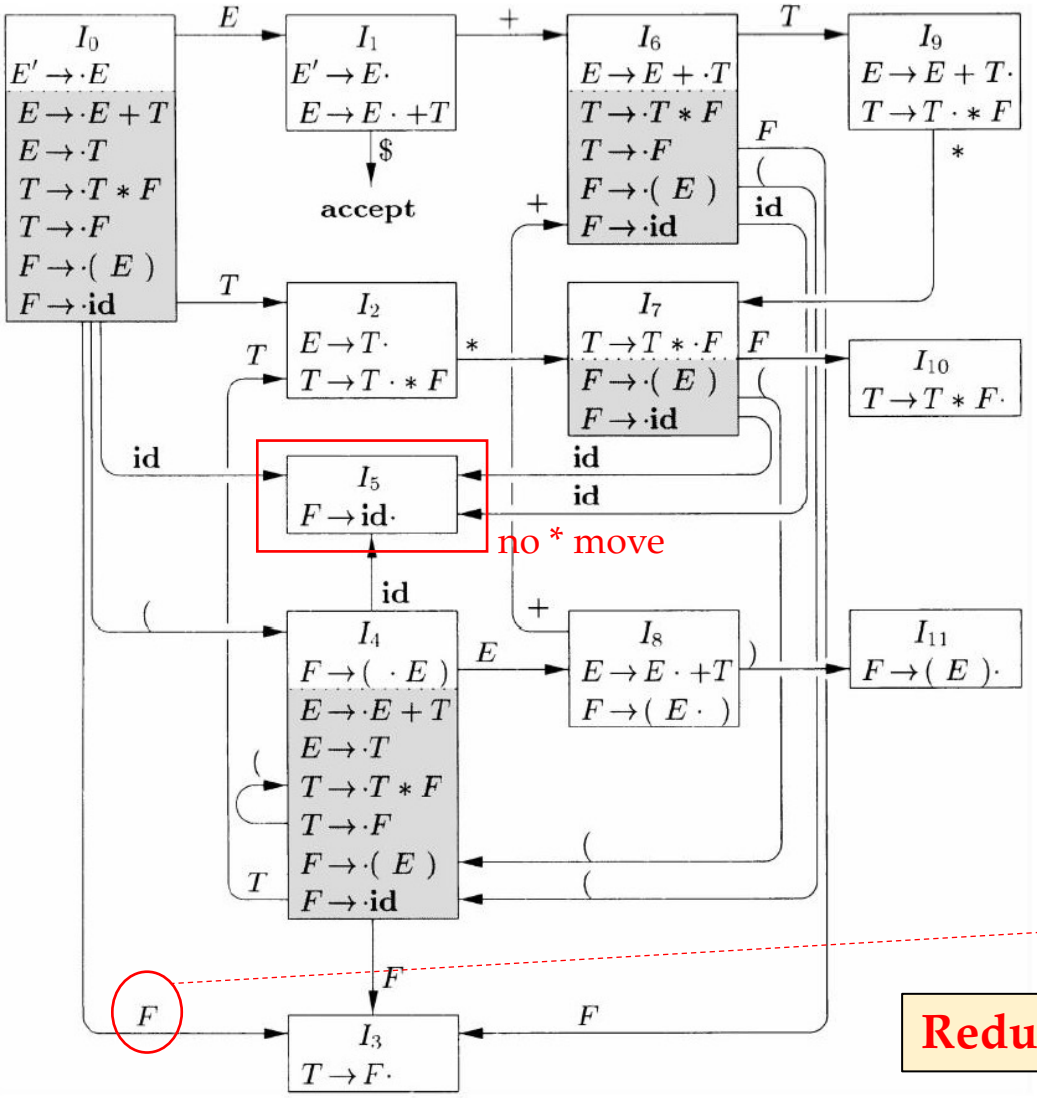
Stack: $ 0 **Input:** id * id $

Grammar Symbols: $

Action: Shift to 5

Shift when the state has a transition on the incoming symbol

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

Stack: $ 0 5 **Input:** * id $

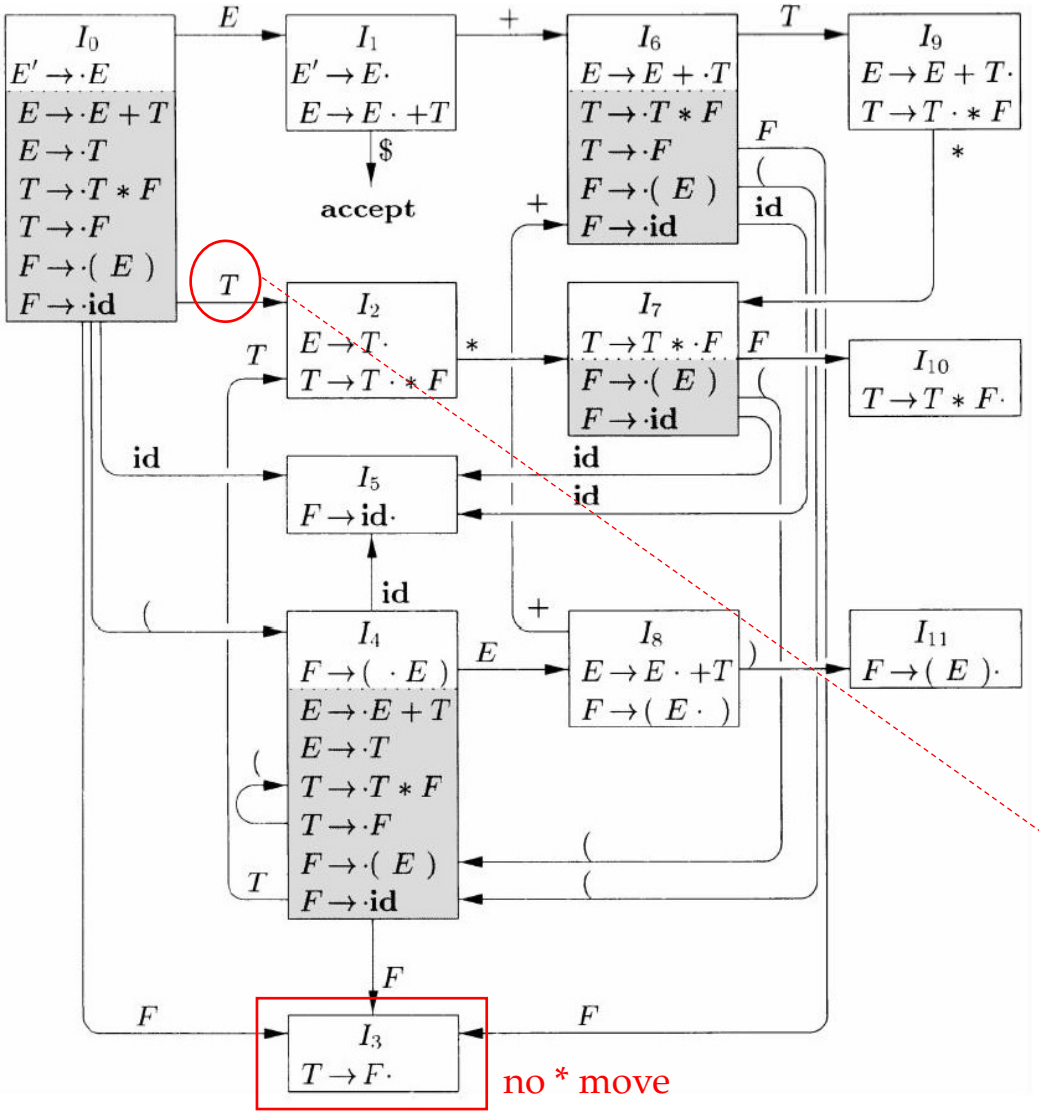
Grammar Symbols: $ id

Action: Reduce by $F \to$ **id**

- Pop state 5 (one symbol corresponds to one state)
- Push state 3

Reduce when there is no further move

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

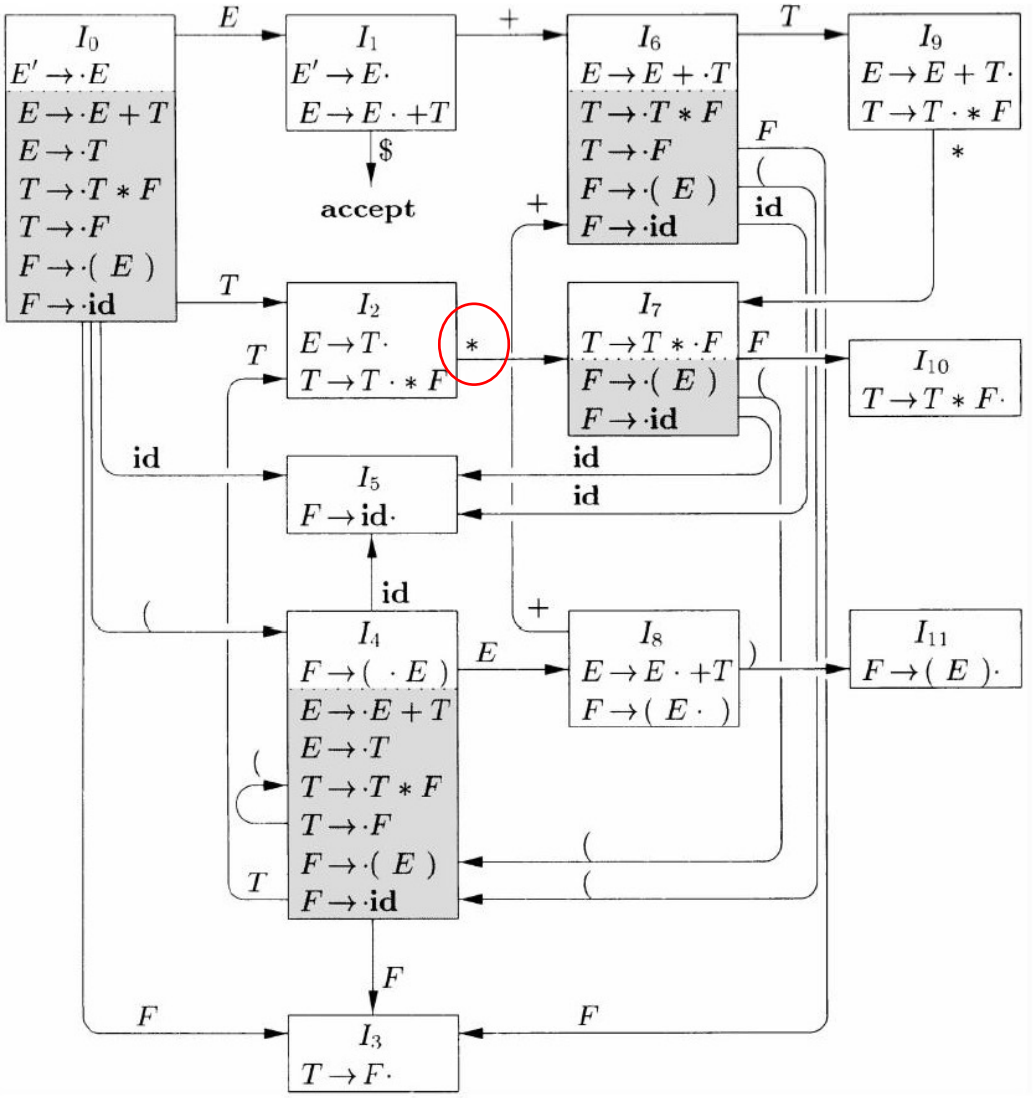
Stack: $ 0 3 **Input:** * id $

Grammar Symbols: $ *F*

Action: Reduce by $T \to F$

- Pop state 3 (one symbol corresponds to one state)
- Push state 2

Example: Parsing id * id



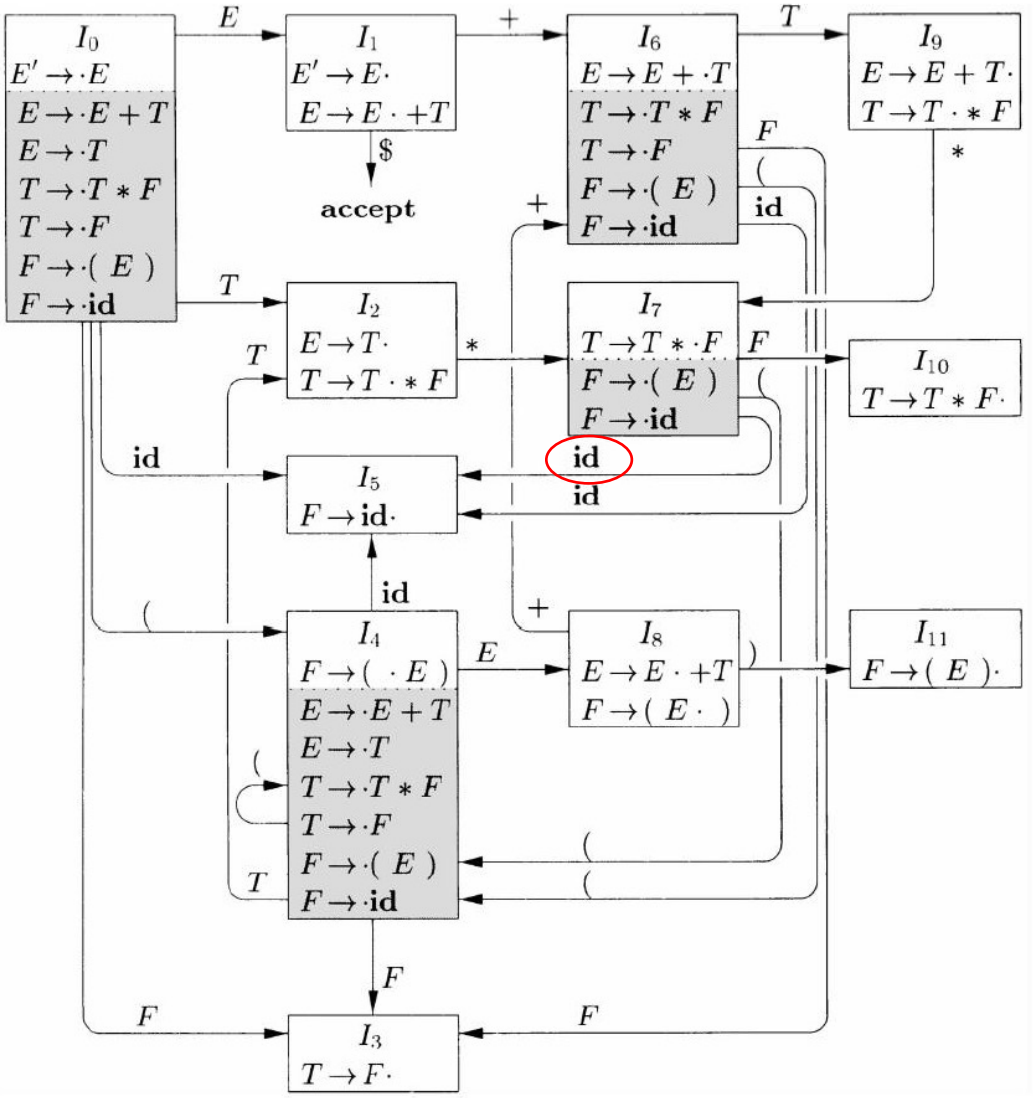
We only keep states in the stack; grammar symbols can be recovered from the states

Stack: $ 0 2 **Input:** * id $

Grammar Symbols: $ *T*

Action: Shift to 7

Example: Parsing id * id



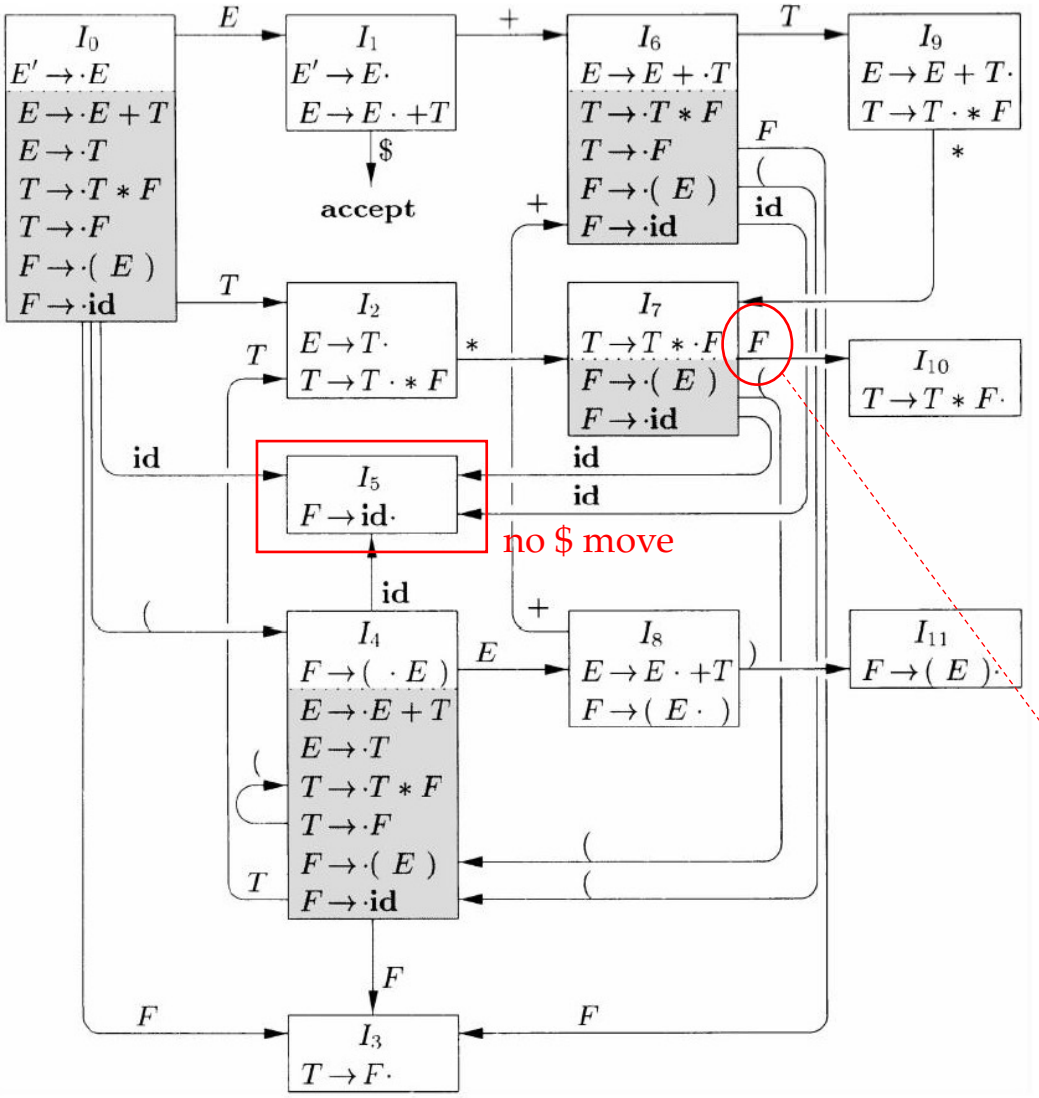
We only keep states in the stack; grammar symbols can be recovered from the states

Stack: $ 0 2 7 **Input:** id $

Grammar Symbols: $ T *

Action: Shift to 5

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

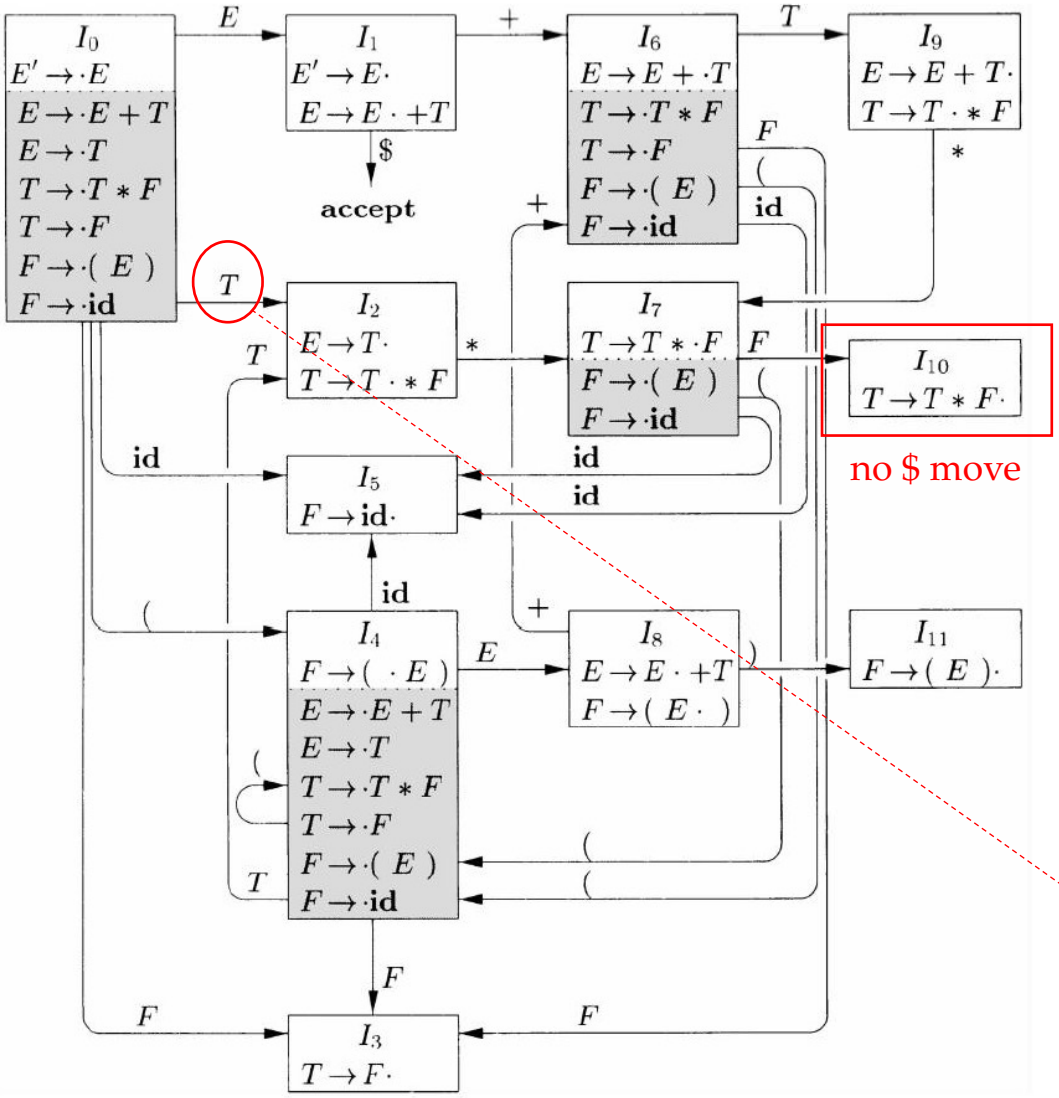
Stack: $ 0 2 7 5 **Input:** $

Grammar Symbols: $ T * id

Action: Reduce by $F \to$ **id**

- Pop state 5 (one symbol corresponds to one state)
- Push state 10

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

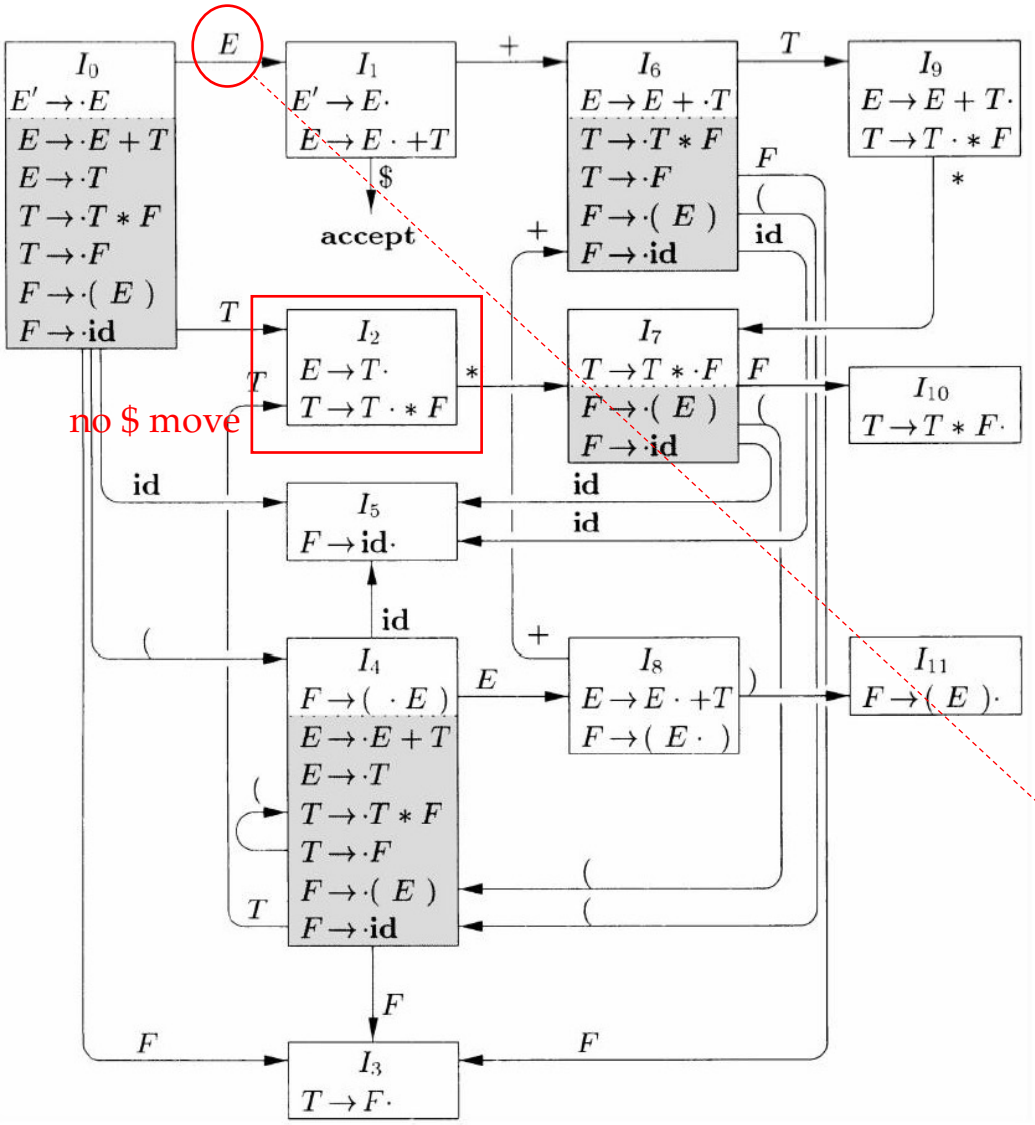
Stack: $ 0 2 7 10 **Input:** $

Grammar Symbols: $ T * F

Action: Reduce by $T \to T * F$

- Pop states 2, 7, 10 (one symbol corresponds to one state)
- Push state 2

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

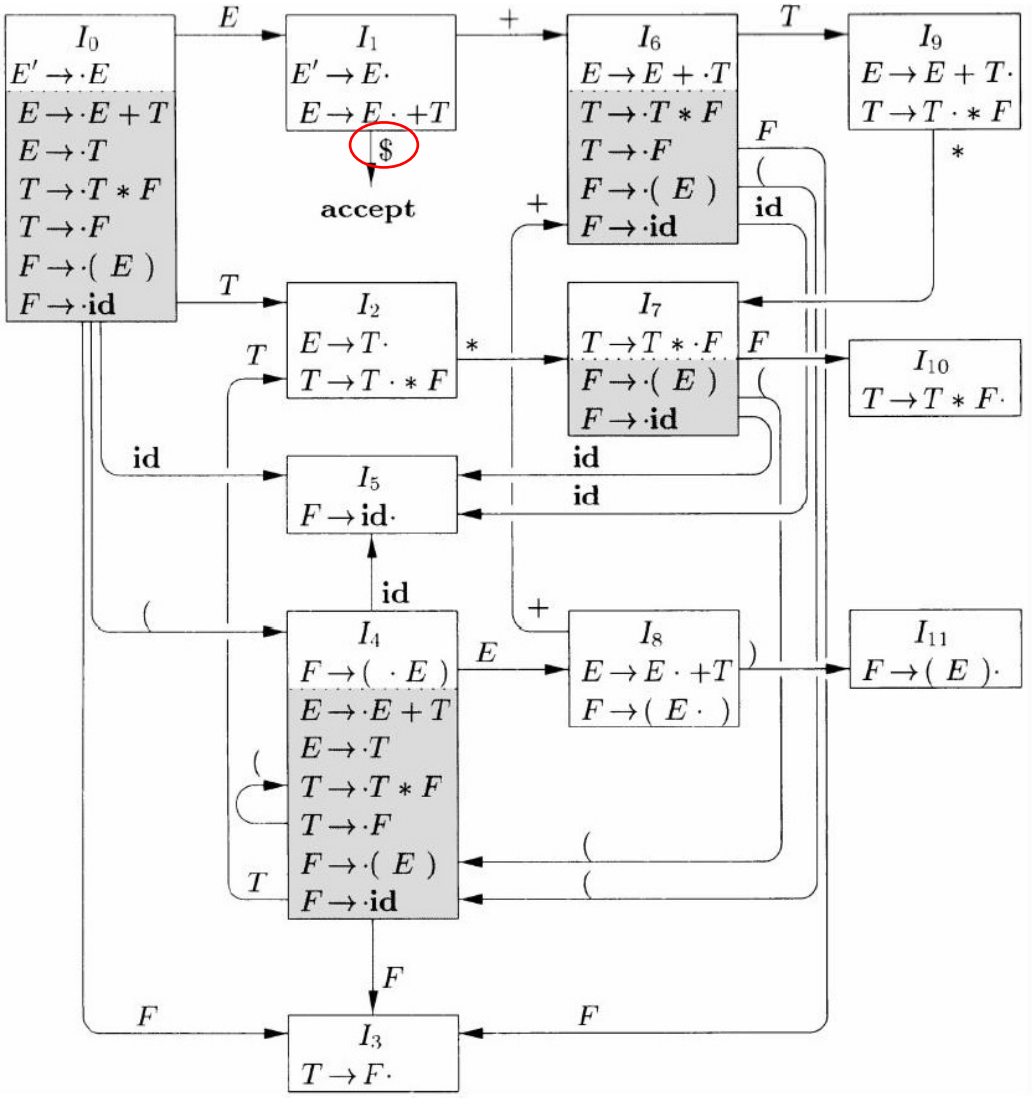
Stack: $ 0 2 **Input:** $

Grammar Symbols: $ T

Action: Reduce by $E \to T$

- Pop states 2 (one symbol corresponds to one state)
- Push state 1

Example: Parsing id * id



We only keep states in the stack; grammar symbols can be recovered from the states

Stack: $ 0 1 **Input:** $

Grammar Symbols: $ *E*

Action: Accept

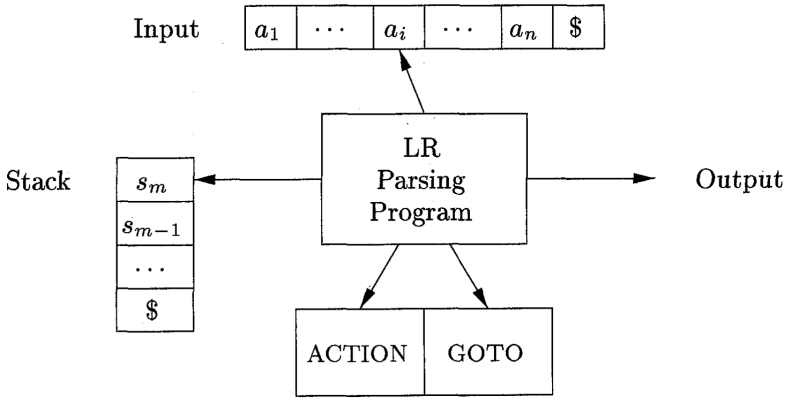
Example: Parsing id * id

- The complete parsing steps

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** * **id** \$	shift to 5
(2)	0 5	\$ **id**	* **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	\$ F	* **id** \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	* **id** \$	shift to 7
(5)	0 2 7	\$ T *	**id** \$	shift to 5
(6)	0 2 7 5	\$ T * **id**	\$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	\$ T * F	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ T	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ E	\$	accept

LR Parser Structure

- An LR parser consists of an input, an output, a stack, a driver program, and a parsing table (ACTION + GOTO)
- The driver program is the same for all LR parsers; only the parsing table changes from one parser to another (depending on the parsing algorithm)
- The stack holds a sequence of states
  - In SLR, the stack holds states from the LR(0) automaton
- The parser decides the next action based on (1) the state at the top of the stack and (2) the next terminal read from the input buffer

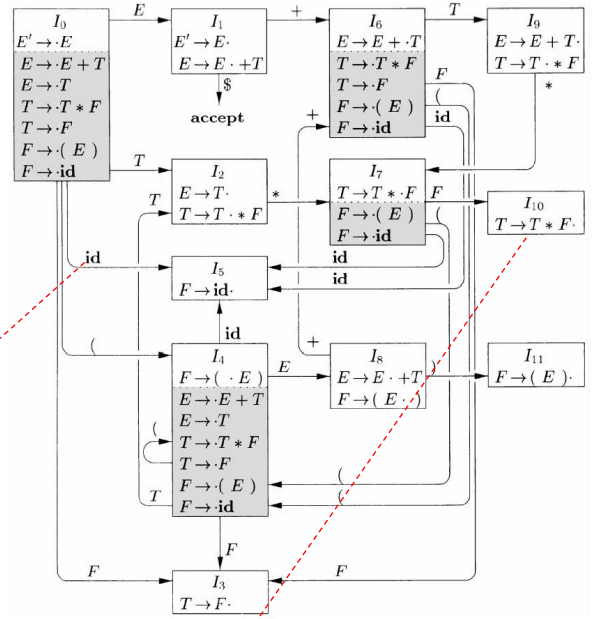


Parsing Table: ACTION + GOTO

- The **ACTION** function takes two arguments: (1) a state i and (2) a terminal a (or \$)
- **ACTION[i, a]** can have one of the four types of values:
  - **Shift j:** shift input a to the stack, and uses state j to represent a
  - **Reduce $A \rightarrow \beta$:** reduce β on the top of the stack to non-terminal A
  - **Accept:** The parser accepts the input and finishes parsing
  - **Error:** syntax errors exist
- The **GOTO** function is obtained from the one defined on sets of items: if GOTO(I_i, A) = I_j, then GOTO(i, A) = j

Parsing Table Example

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

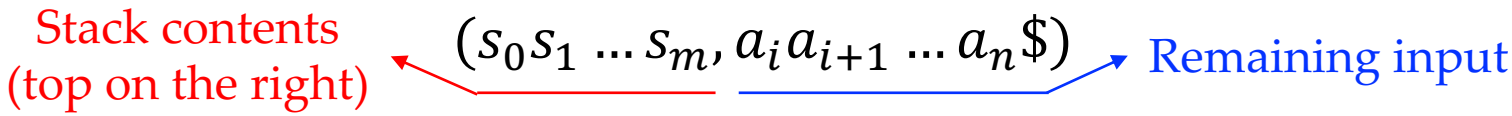


(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to$ id

- *s*5: shift by pushing state 5 *r*3: reduce using production No. 3
- GOTO entries for terminals are not listed, can be checked in ACTION part

LR Parser Configurations (态势)

- "**Configuration**" is notation for representing the complete state of the parser (stack status + input status). A *configuration* is a pair:



- By construction, each state (except s_0) in an LR parser corresponds to a set of items and a grammar symbol (the symbol that leads to the state transition, i.e., the symbol on the incoming edge)
  - Suppose X_i is the grammar symbol for state s_i
  - Then $X_0 X_1 \dots X_m a_i a_{i+1} \dots a_n$ is a right-sentential form (assume no errors)

Behavior of the LR Parser

- For the configuration $(s_0 s_1 \dots s_m, a_i a_{i+1} \dots a_n \$)$, the LR parser checks ACTION$[s_m, a_i]$ in the parsing table to decide the parsing action
  - shift s: shift the next state s onto the stack, entering the configuration $(s_0 s_1 \dots s_m s, a_{i+1} \dots a_n \$)$
  - reduce $A \rightarrow \beta$: execute a reduce move, entering the configuration $(s_0 s_1 \dots s_{m-r} s, a_i a_{i+1} \dots a_n \$)$, where r = the length of β, and $s =$ GOTO(s_{m-r}, A) ➔ pop r states and push the state s onto stack
  - accept: parsing successful
  - error: the parser has found an error and calls an error recovery routine

LR-Parsing Algorithm

- **Input:** The parsing table for a grammar G and an input string ω
- **Output:** If ω is in $L(G)$, the reduction steps of a bottom-up parse for ω; otherwise, an error indication
- **Initial configuration:** $(s_0, \omega\$)$

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Constructing SLR-Parsing Tables

- The SLR-parsing table for a grammar G can be constructed based on the LR(0) item sets and LR(0) automaton

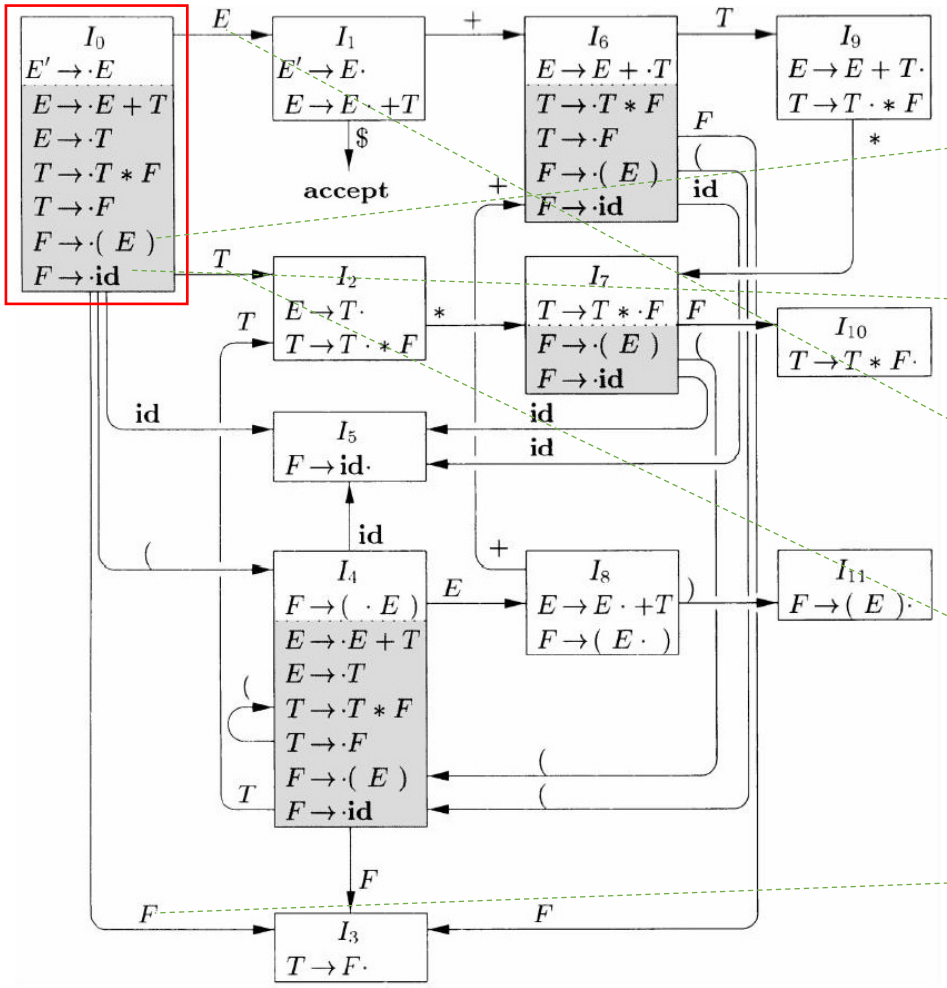
  1. Construct the canonical LR(0) collection $\{I_0, I_1, \ldots, I_n\}$ for the augmented grammar G'
  2. State i is constructed from I_i. ACTION can be determined as follows:
     - If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}[I_i, a] = I_j$, then set $\text{ACTION}[i, a]$ to "shift j" (here a must be a terminal)
     - If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for **all a in FOLLOW(A)**; here A may not be S'
     - If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept"
  3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}(i, A) = j$

Constructing SLR-Parsing Tables

4. All entries not defined in steps 2 and 3 are set to "error"
5. Initial state is the one constructed from the item set containing $[S' \rightarrow \cdot\, S]$

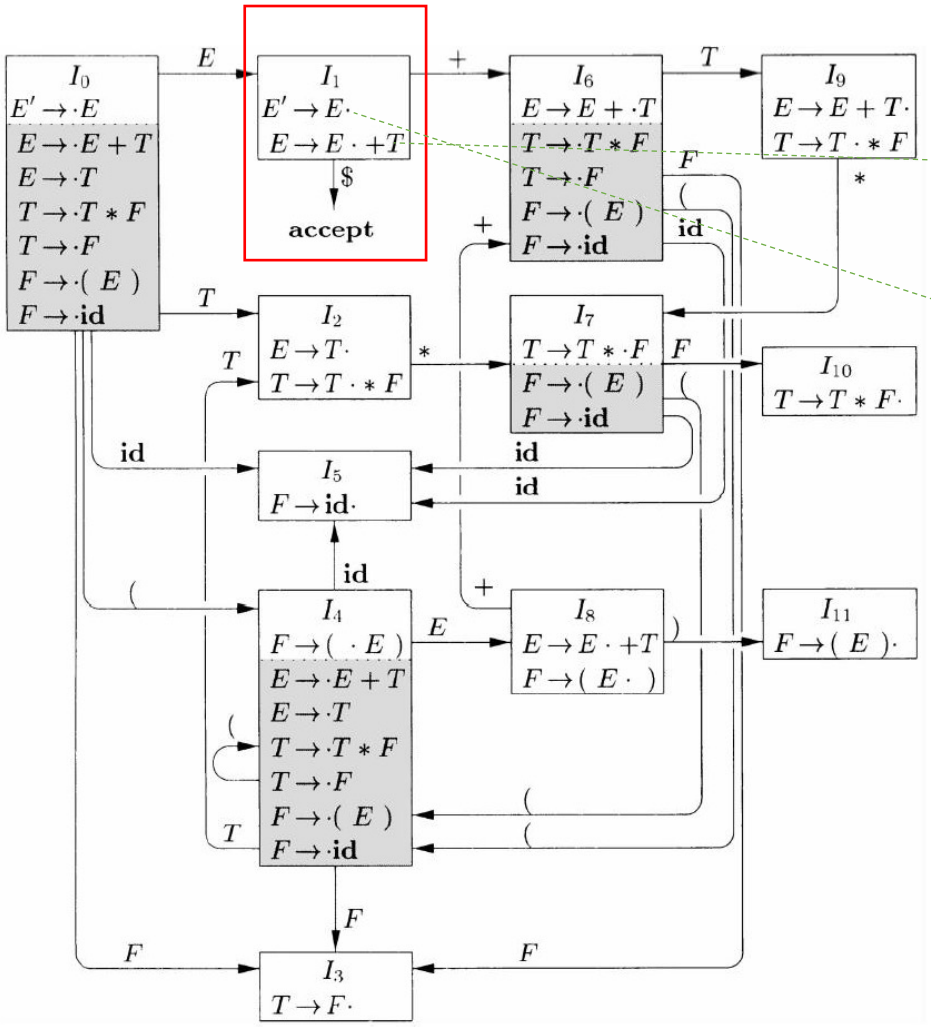
If there is no conflict during the parsing table construction (i.e., multiple actions for a table entry), the grammar is **SLR(1)**

Example



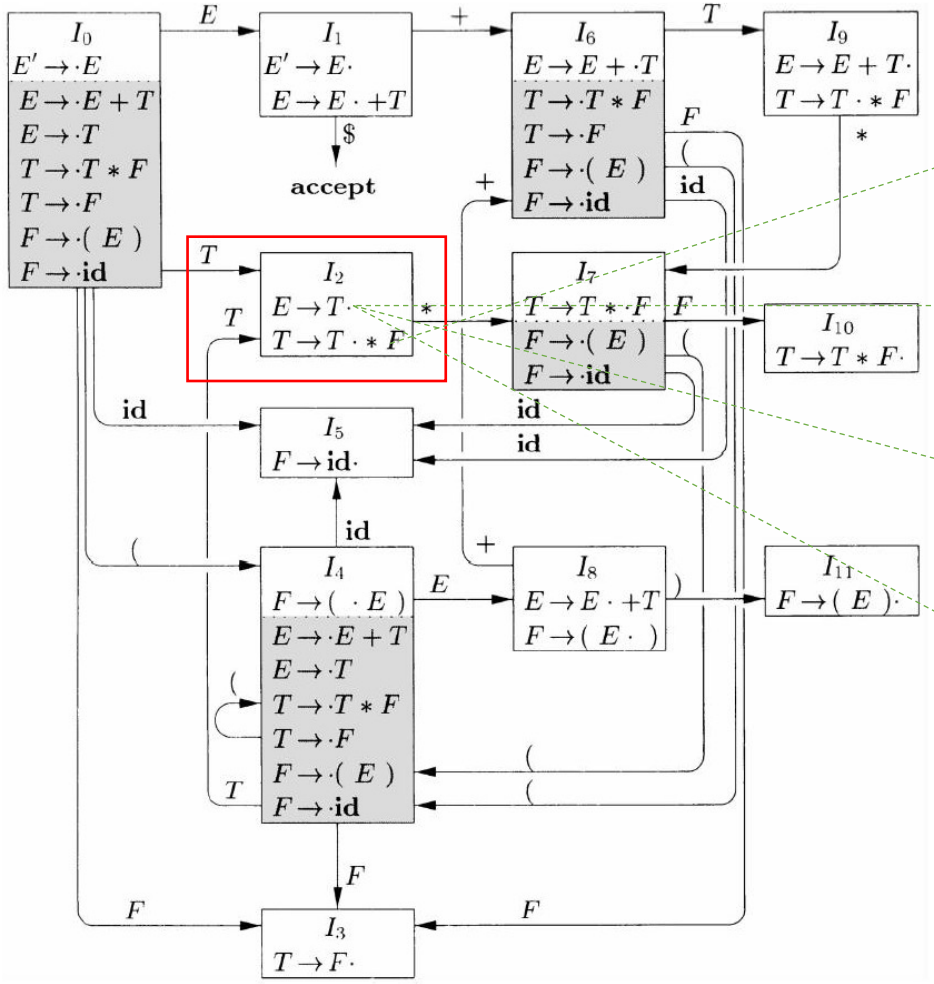
- ACTION$(0, () =$ s4 (shift 4)
- ACTION$(0, \mathbf{id}) =$ s5
- GOTO$[0, E] = 1$
- GOTO$[0, T] = 2$
- GOTO$[0, F] = 3$

Example



- ACTION$(1, +) =$ s6
- ACTION$(1, \$) =$ accept

Example



- ACTION(2,∗) = s7
- ACTION(2, $) = reduce $E \to T$
- ACTION(2, +) = reduce $E \to T$
- ACTION(2,)) = reduce $E \to T$

FOLLOW(E) = {$, +,)}

Non-SLR Grammar

- Grammar
  - $S \rightarrow L = R \mid R$
  - $L \rightarrow * R \mid \mathbf{id}$
  - $R \rightarrow L$
- For item set I_2:
  - According to item #1: ACTION[2, =] is "s6"
  - According to item #2: ACTION[2, =] is "reduce $R \rightarrow L$" (FOLLOW(R) contains =)

I_0: $S' \rightarrow \cdot S$; $S \rightarrow \cdot L = R$; $S \rightarrow \cdot R$; $L \rightarrow \cdot * R$; $L \rightarrow \cdot \mathbf{id}$; $R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$; $R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$; $R \rightarrow \cdot L$; $L \rightarrow \cdot * R$; $L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$; $R \rightarrow \cdot L$; $L \rightarrow \cdot * R$; $L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

This grammar is not ambiguous

CLR and LALR will succeed on a larger collection of grammars, including the above one. However, there exist unambiguous grammars for which every LR parser construction method will encounter conflicts.

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
  - Simple LR (SLR)
  - Canonical LR (CLR)
  - Look-ahead LR (LALR)
  - Error Recovery (Lab)
- Parser Generators (Lab)

Weakness of the SLR Method

- In SLR, the state i calls for reduction by $A \to \alpha$ if (1) the item set I_i contains item $[A \to \alpha \cdot]$ and (2) input symbol a is in FOLLOW(A)

- In some situations, after reduction, the content $\beta\alpha$ on stack top would become βA that cannot be followed by a in any right-sentential form* (i.e., only requiring "a is in FOLLOW(A)" is not enough, βA cannot be followed by a)

* Although SLR algorithm requires a to belong to FOLLOW(A), it is still too casual as the stack content below A is not considered (β is not considered).

Example: Parsing id = id

- $S \rightarrow L = R \mid R$
- $L \rightarrow * R \mid \mathbf{id}$
- $R \rightarrow L$

I_0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1:
$S' \rightarrow S\cdot$

I_2:
$S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3:
$S \rightarrow R\cdot$

I_4:
$L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5:
$L \rightarrow \mathbf{id}\cdot$

I_6:
$S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7:
$L \rightarrow *R\cdot$

I_8:
$R \rightarrow L\cdot$

I_9:
$S \rightarrow L = R\cdot$

Stack	Symbols	Input	Action
$0		id = id	Shift 5
$05	id	= id	Reduce by L→id
$02	L	= id	Suppose reduce by R→L
$03	R	= id	**Error!**

Cannot shift, cannot reduce since FOLLOW(S) = {$}

Problem: SLR reduces too casually

How to know if a reduction is a good move? Utilize the next input symbol to precisely determine whether to call for a reduction.

LR(1) Item

- **Idea:** Carry more information in the state to rule out some invalid reductions (splitting LR(0) states)

- General form of an LR(1) item: $[A \to \alpha \cdot \beta, a]$
  - $A \to \alpha\beta$ is a production and a is a terminal or \$
  - "1" refers to the length of the 2nd component: the *lookahead* (向前看字符)*
  - The lookahead symbol has no effect if β is not ϵ since it only helps determine whether to reduce (a will be inherited during state transitions)
  - An item of the form $[A \to \alpha \cdot, a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is a (the set of such a's is a **subset** of FOLLOW(A))

*: LR(0) items do not have lookahead symbols, and hence they are called LR(0)

Constructing LR(1) Item Sets (1)

- Constructing the collection of LR(1) item sets is essentially the same as constructing the canonical collection of LR(0) item sets. The only differences lie in the CLOSURE and GOTO functions.

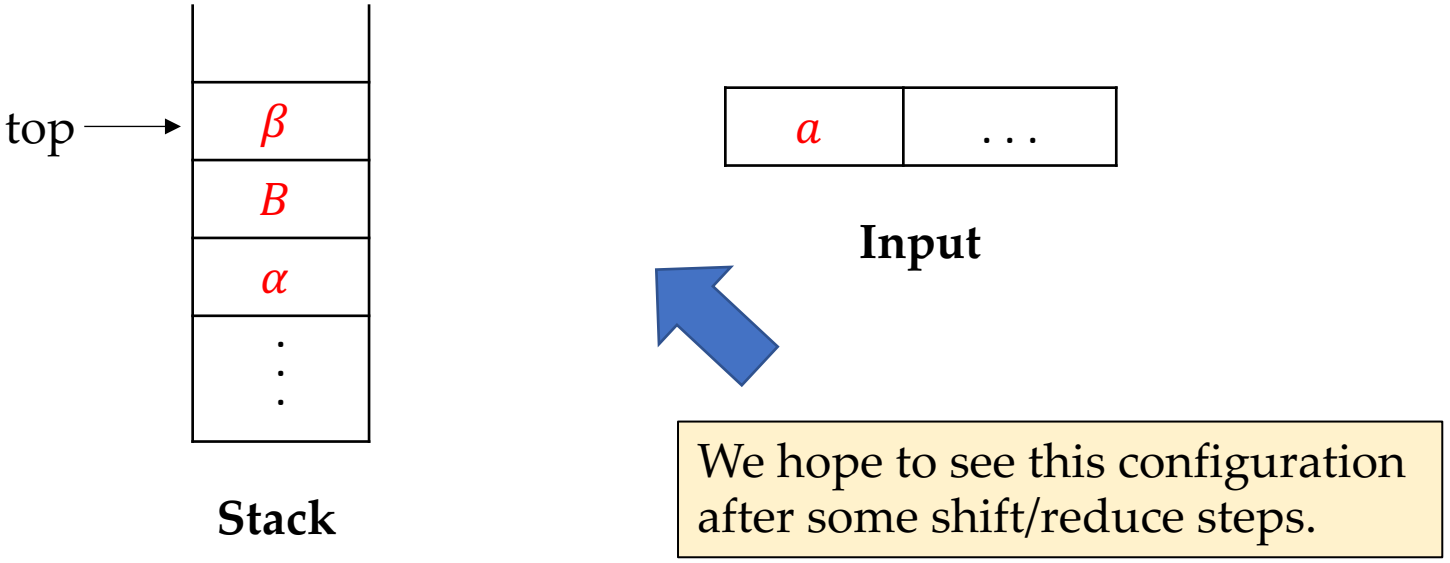
```
SetOfItems CLOSURE(I) {
        repeat
                for ( each item [A → α·Bβ, a] in I )
                        for ( each production B → γ in G' )
                                for ( each terminal b in FIRST(βa) )
                                        add [B → ·γ, b] to set I;
        until no more items are added to I;
        return I;
}
```

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

It only generates the new item $[B \rightarrow \cdot\, \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ if b is in FIRST(βa)

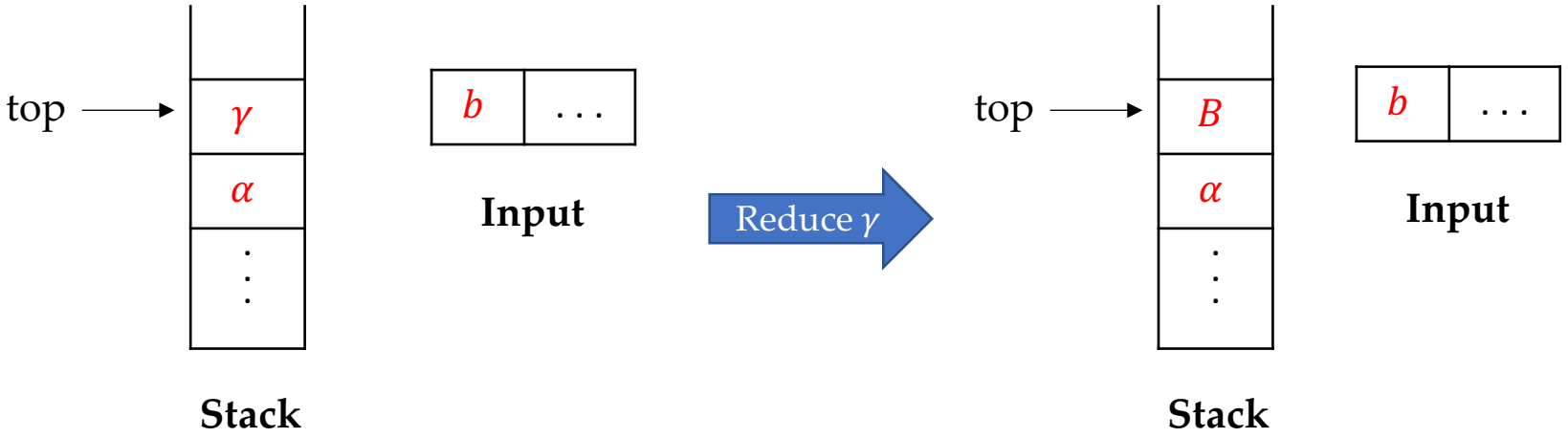
Why b should be in FIRST(βa)?

- The item $[A \to \alpha \cdot B\beta, a]$ will derive $[A \to \alpha B\beta \cdot, a]$, which calls for reduction when the stack top contains $\alpha B\beta$ and the next input symbol is a



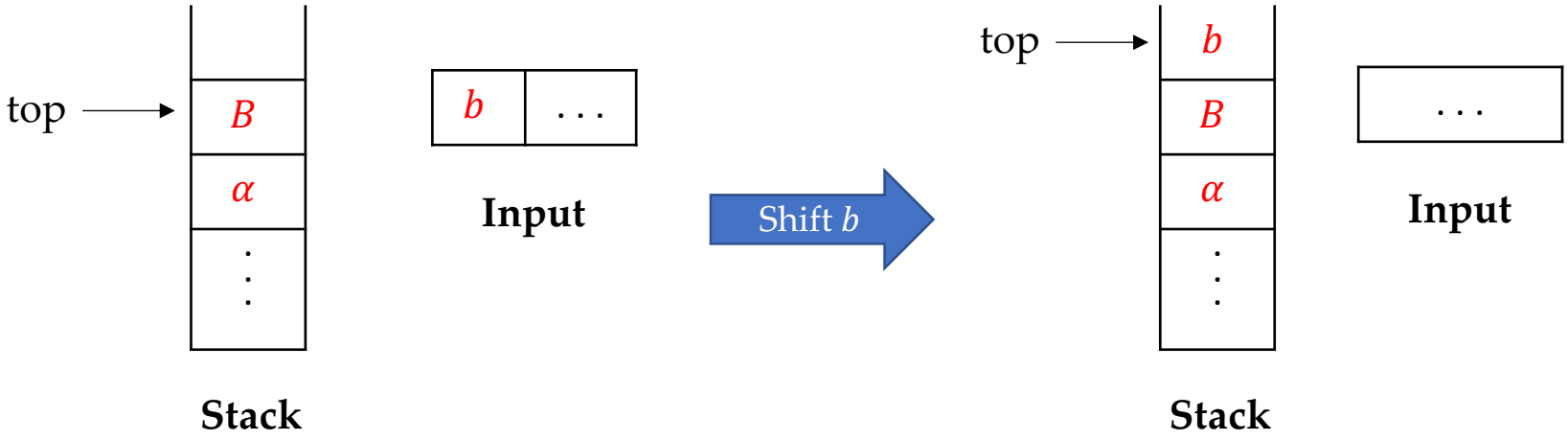
Why b should be in FIRST(βa)?

- When generating the item $[B \rightarrow \cdot\, \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$, suppose we allow that b is not in $FIRST(\beta a)$
- We add the item $[B \rightarrow \cdot\, \gamma, b]$ because we hope that at certain time point during parsing, when we see γ on stack top and b as the next input symbol, we can first reduce γ to B so that in some later step the stack top would contain $\alpha B \beta$ (then we can further reduce it to A)



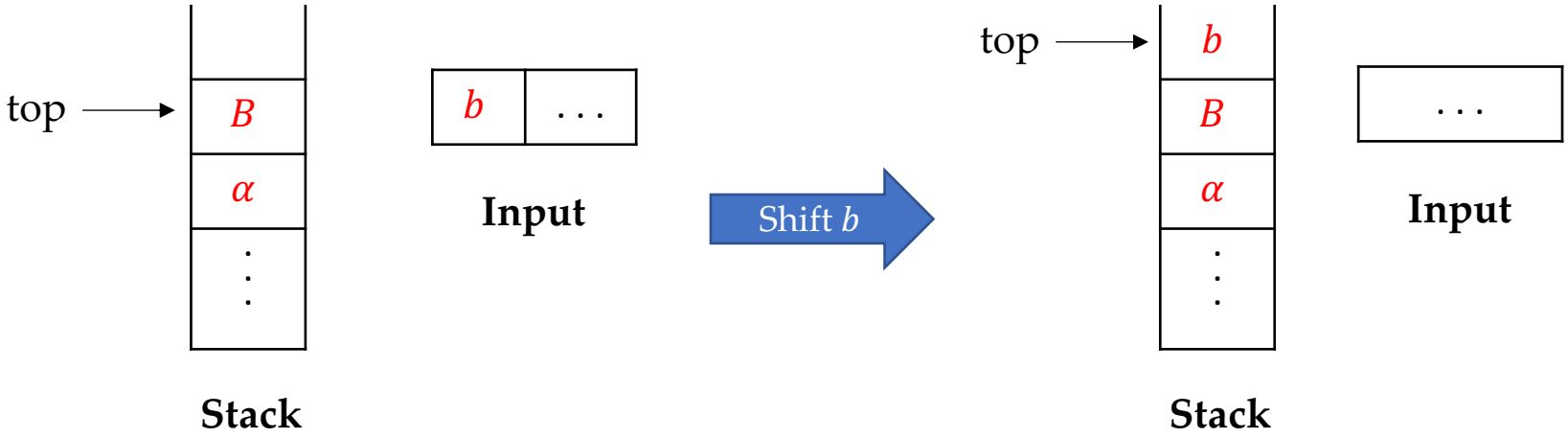
Why b should be in FIRST(βa)?

- If we reduce γ to B, the next action would be "shift b to the stack"
  - Because the production $A \rightarrow \alpha B \beta$ tells us that we are ready for reduction only when we see $\alpha B \beta$ on stack top (i.e., "the next action is shift" is guaranteed by design, as we want to eventually see $\alpha B \beta$ on stack top)



Why b should be in FIRST(βa)?

- Since b is not in $FIRST(\beta a)$, the stack top will never become the form $\alpha B\beta$, which means we will never be able to reduce $\alpha B\beta$ to A
- Then why should we generate $[B \rightarrow \cdot\, \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ in the first palce???



Constructing LR(1) Item Sets (2)

- Constructing the collection of LR(1) item sets is essentially the same as constructing the canonical collection of LR(0) item sets. The only differences lie in the CLOSURE and GOTO functions.

```
SetOfItems GOTO(I, X) {
        initialize J to be the empty set;
        for ( each item [A → α·Xβ, a] in I )
                add item [A → αX·β, a] to set J;
        return CLOSURE(J);
}
```

GOTO(I, X) in LR(0) item sets:

The closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ where $[A \rightarrow \alpha \cdot X\beta]$ is in I.

The lookahead symbols are passed to new items from existing items

Constructing LR(1) Item Sets (3)

```
void items(G') {
        C = {CLOSURE({[S' → ·S]})};
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C on a round;
}
```

```
void items(G') {
        initialize C to {CLOSURE({[S' → ·S, $]})};
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C;
}
```

Constructing the collection of LR(0) item sets

Constructing the collection of LR(1) item sets

LR(1) Item Sets Example

It only generates the new item $[B \rightarrow \cdot\, \gamma, b]$ from $[A \rightarrow \alpha \cdot B\beta, a]$ if b is in FIRST(βa)

- Augmented grammar:
  - $S' \rightarrow S \qquad S \rightarrow CC \qquad C \rightarrow cC \mid d$
- Constructing I_0 item set and GOTO function:
  - $I_0 = \text{CLOSURE}([S' \rightarrow \cdot\, S, \$]) =$
    - $\{[S' \rightarrow \cdot\, S, \$], [S \rightarrow \cdot\, CC, \$], [C \rightarrow \cdot\, cC, c/d], [C \rightarrow \cdot\, d, c/d]\}$

FIRST($) = {$}

FIRST(C$) = {c, d}

  - $\text{GOTO}(I_0, S) = \text{CLOSURE}(\{[S' \rightarrow S \cdot, \$]\}) = \{[S' \rightarrow S \cdot, \$]\}$
  - $\text{GOTO}(I_0, C) = \text{CLOSURE}(\{[S \rightarrow C \cdot C, \$]\}) =$
    - $\{[S \rightarrow C \cdot C, \$], [C \rightarrow \cdot\, cC, \$], [C \rightarrow \cdot\, d, \$]\}$

FIRST($) = {$}

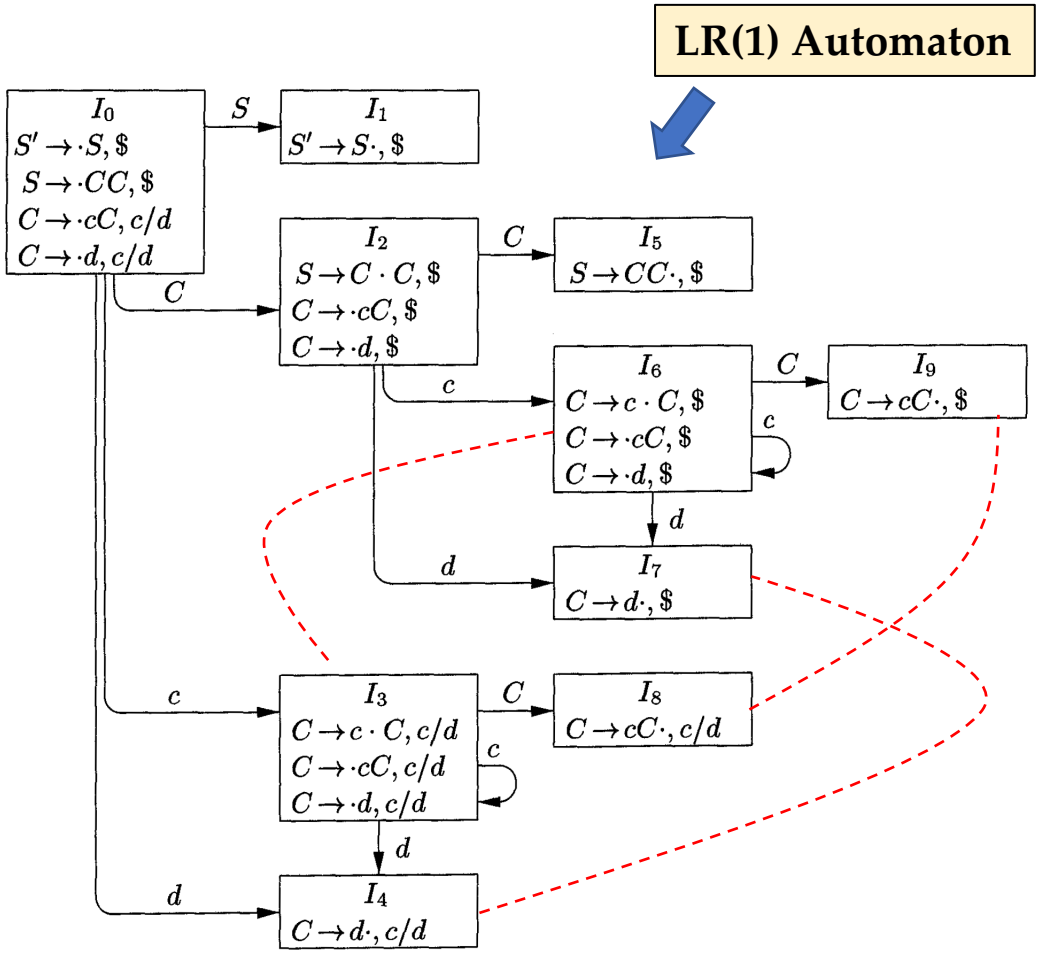
  - $\text{GOTO}(I_0, c) = \text{CLOSURE}(\{[C \rightarrow c \cdot C, c/d\,]\}) =$
    - $\{[C \rightarrow c \cdot C, c/d\,], [C \rightarrow \cdot\, cC, c/d], [C \rightarrow \cdot\, d, c/d]\}$
  - $\text{GOTO}(I_0, d) = \text{CLOSURE}(\{[C \rightarrow d \cdot, c/d\,]\}) = \{[C \rightarrow d \cdot, c/d\,]\}$

The GOTO Graph Example

10 states in total

These states are equivalent if we ignore the lookahead symbols (SLR makes no such distinctions of states):

- I_3 and I_6
- I_4 and I_7
- I_8 and I_9



Constructing Canonical LR(1) Parsing Tables

1. Construct $C' = \{I_0, I_1, \ldots, I_n\}$, the collection of LR(1) item sets for the augmented grammar G'

2. State i of the parser is constructed from I_i. Its parsing action is determined as follows:

   - If $[A \to \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "*shift j*." Here, a must be a terminal.
   - If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set $\text{ACTION}[i, a]$ to "*reduce* $A \to \alpha$"
   - If $[S' \to S \cdot, \$]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "*accept*"

More restrictive than SLR

If any conflicting actions result from the above rules, we say the grammar is not LR(1)

Constructing Canonical LR(1) Parsing Tables

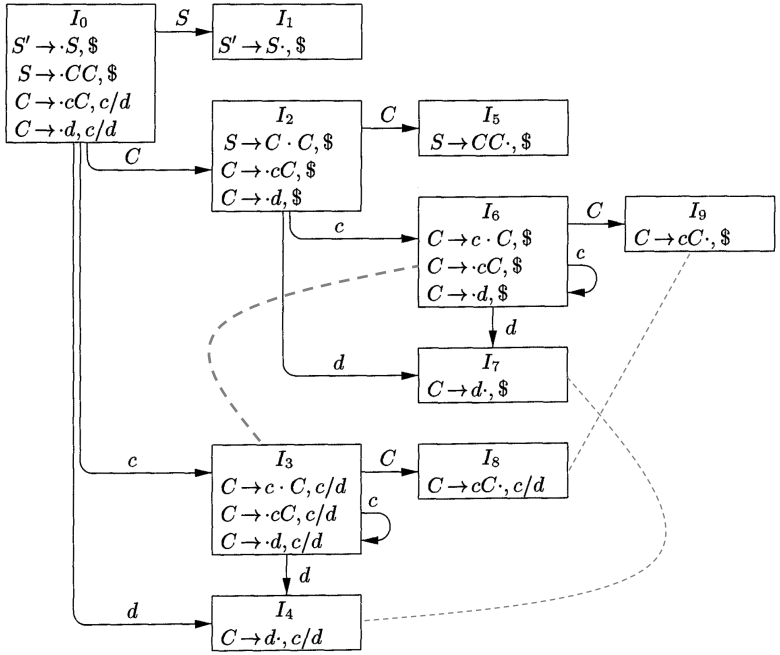
3. The goto transitions for state i are constructed from all nonterminals A using the rule: If $\texttt{GOTO}(I_i, A) = I_j$, then $\texttt{GOTO}(i, A) = j$

4. All entries not defined in steps (2) and (3) are made "error"

5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot\, S, \$]$

LR(1) Parsing Table Example

Grammar: $S' \rightarrow S$ $S \rightarrow CC$ $C \rightarrow cC \mid d$



STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Three pairs of states can be seen as being split from the corresponding LR(0) states:

(3, 6) (4, 7) (8, 9)

Outline

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

- Simple LR (SLR)
- Canonical LR (CLR)
- Look-ahead LR (LALR)
- Error Recovery (Lab)

Lookahead LR (LALR) Method

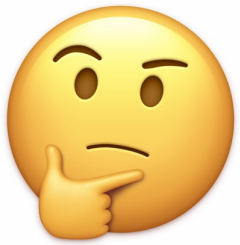
- SLR(1) is not powerful enough to handle a large collection of grammars (recall the previous unambiguous grammar)
- LR(1) has a huge set of states in the parsing table (states are too fine-grained)
- LALR(1) is often used in practice
  - Keeps the lookahead symbols in the items
  - Its number of states is the same as that of SLR(1)
  - Can deal with most common syntactic constructs of modern programming languages

Merging States in LR(1) Parsing Tables

Grammar:
$S' \rightarrow S \quad S \rightarrow CC \quad C \rightarrow cC \mid d$

- **State 4:**
  - Reduce by $C \rightarrow d$ if the next input symbol is c or d
  - Error if \$
- **State 7:**
  - Reduce by $C \rightarrow d$ if the next input symbol is \$
  - Error if c or d

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		



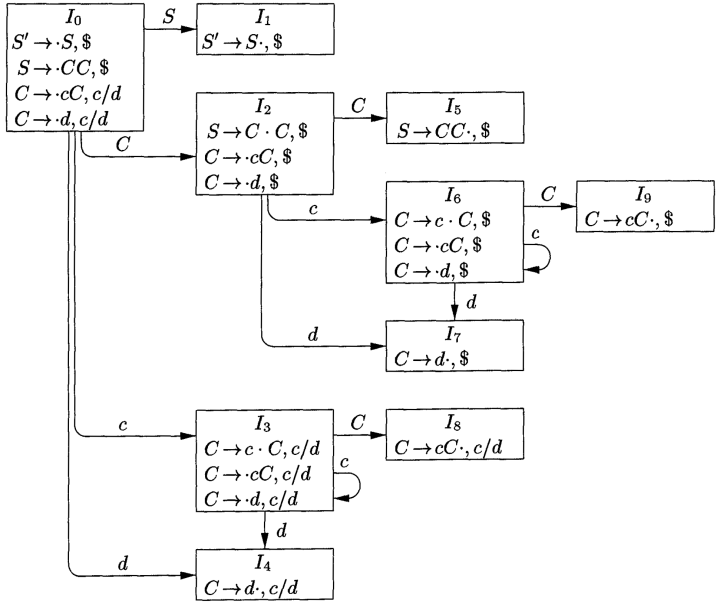
Can we merge states 4 and 7 so that the parser can reduce for all input symbols?

I_{47}: $C \rightarrow d \cdot, c/d/\$$

- I_4:$[C \rightarrow d \cdot, c/d]$
- I_7:$[C \rightarrow d \cdot, \$]$

The Basic Idea of LALR

- Look for sets of LR(1) items with the same ***core***
  - The core of an LR(1) item set is the set of the first components
    - The core of I_4 and I_7 is $\{[C \rightarrow d \cdot]\}$
    - The core of I_3 and I_6 is $\{[C \rightarrow c \cdot C], [C \rightarrow \cdot\, cC], [C \rightarrow \cdot\, d]\}$

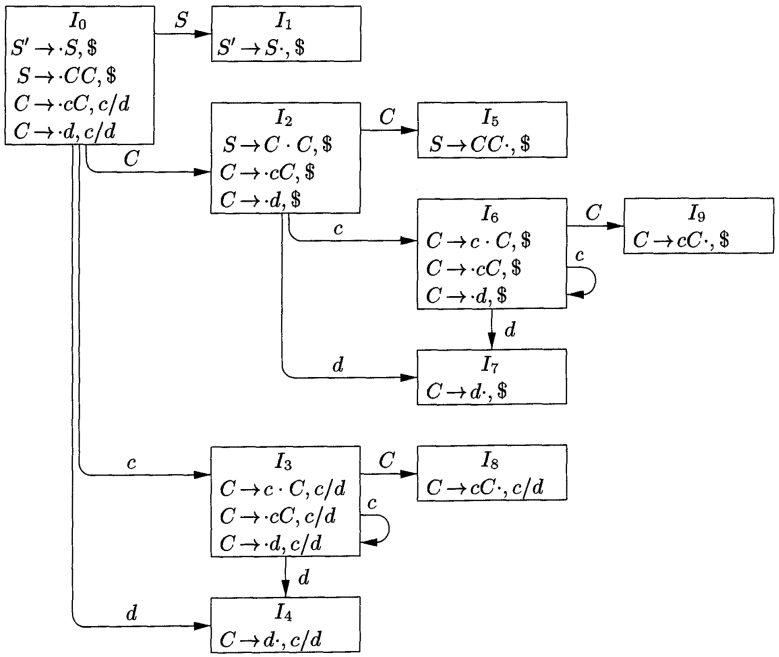


The Basic Idea of LALR Cont.

- Look for sets of LR(1) items with the same ***core***
  - The core of an LR(1) item set is the set of the first components
    - The core of I_4 and I_7 is $\{[C \rightarrow d \cdot]\}$
    - The core of I_3 and I_6 is $\{[C \rightarrow c \cdot C], [C \rightarrow \cdot cC], [C \rightarrow \cdot d]\}$
  - In general, a core is a set of LR(0) items
- We may merge the LR(1) item sets with common cores into one set of items

The Basic Idea of LALR Cont.

- Since the core of GOTO(I, X) depends only on the core of I, the goto targets of merged sets also have the same core and hence can be merged



Consider I_3 and I_6:

- The core $\{[C \to c \cdot C], [C \to \cdot\, cC], [C \to \cdot\, d]\}$ determines state transition targets
- Before merging, GOTO(I_3, C) = I_9, GOTO(I_6, C) = I_8
- After merging, I_3 and I_6 become I_{36}, I_8 and I_9 become I_{89}, and GOTO(I_{36}, C) = I_{89}

Conflicts Caused by State Merging

- Merging states in an LR(1) parsing table may cause conflicts
- Merging does not cause shift/reduce conflicts
  - Suppose after merging there is shift/reduce conflict on lookahead a
    - There is an item $[A \to \alpha \cdot, a]$ in a merged set calling for a reduction by $A \to \alpha$
    - There is another item $[B \to \beta \cdot a\gamma, ?]$ in the set calling for a shift
  - Since the cores of the sets to be merged are the same, there must be a set containing both $[A \to \alpha \cdot, a]$ and $[B \to \beta \cdot a\gamma, ?]$ before merging
  - Then before merging, there is already a shift/reduce conflict on a. According to LR(1) parsing table construction algorithm, the grammar is not LR(1). **Contradiction!!!**
- Merging states may cause reduce/reduce conflicts

Example of Conflicts

- An LR(1) grammar:
  - $S' \rightarrow S \qquad S \rightarrow aAd \mid bBd \mid aBe \mid bAe \qquad A \rightarrow c \qquad B \rightarrow c$
- Language: $\{acd, bcd, ace, bce\}$
- One set of valid LR(1) items
  - $\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$
- Another set of valid LR(1) items
  - $\{[B \rightarrow c \cdot, d], [A \rightarrow c \cdot, e]$
- After merging, the new item set: $\{[A \rightarrow c \cdot, d/e], [B \rightarrow c \cdot, d/e]\}$
  - **Conflict:** reduce c to A or B when the next input symbol is d/e?

Constructing LALR Parsing Table

- Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items
- For each core present among a set of LR(1) items, find all sets having that core, and replace these sets by their union
- Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting collection after merging.
  - The parsing actions for state i are constructed from J_i following the LR(1) parsing table construction algorithm.
  - If there is a conflict, this algorithm fails to produce a parser and the grammar is not LALR(1)

Basic idea: Merging states in LR(1) parsing table; If there is no reduce-reduce conflict, the grammar is LALR(1), otherwise not LALR(1).

Constructing LALR Parsing Table

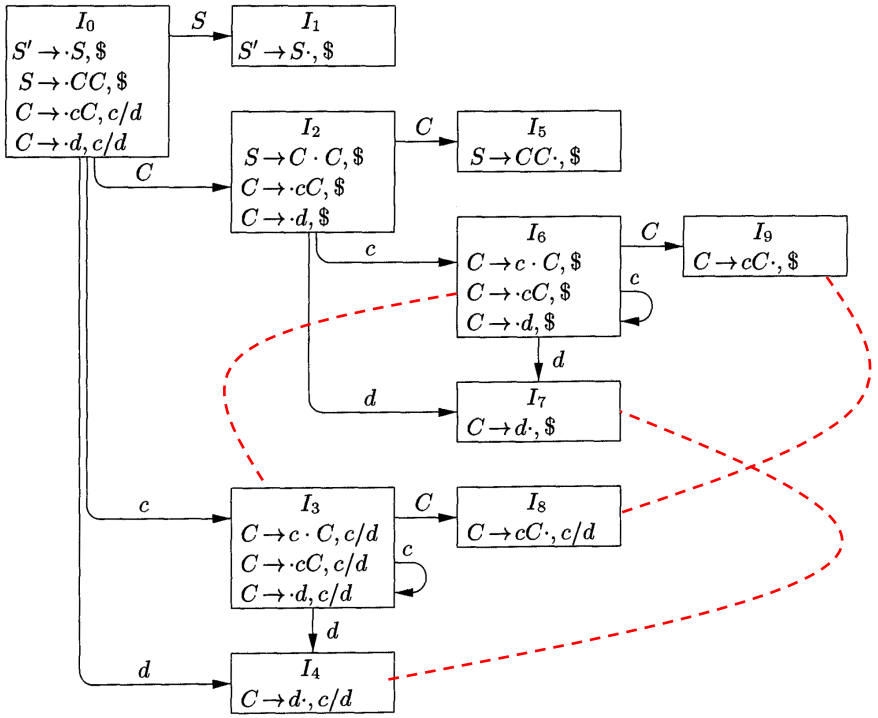
- Construct the GOTO table as follows:
  - If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of $\texttt{GOTO}(I_1, X)$, $\texttt{GOTO}(I_2, X)$, …, $\texttt{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core.
  - Let K be the union of all sets of items having the same core as $\texttt{GOTO}(I_1, X)$
  - $\texttt{GOTO}(J, X) = K$

Check the previous example to understand the above process:

- I_3 and I_6 have the same core; I_{36} is the union of the two LR(1) item sets
- GOTO(I_3, C) = I_8; GOTO(I_6, C) = I_9
- I_8 and I_9 have the same core; I_{89} is the union of the two LR(1) item sets
- GOTO(I_{36}, C) = I_{89}

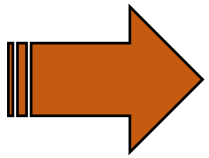
LALR Parsing Table Example

- Merging item sets
  - I_{36}: $[C \to c \cdot C, c/d/\$], [C \to \cdot\, cC, c/d/\$], [C \to \cdot\, d, c/d/\$]$
  - I_{47}: $[C \to d \cdot, c/d/\$]$
  - I_{89}: $[C \to cC \cdot, c/d/\$]$
- $\text{GOTO}(I_{36}, C) = I_{89}$



LALR Parsing Table Example

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		



STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Comparisons Among LR Parsers

- The languages (grammars) that can be handled
  - CLR > LALR > SLR
- # states in the parsing table
  - CLR > LALR = SLR
- Driver programs
  - SLR = CLR = LALR

Outline

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

- Simple LR (SLR)
- Canonical LR (CLR)
- Look-ahead LR (LALR)
- Error Recovery (Lab)

Error Recovery in LR Parsing

- An LR parser should be able to handle errors:
  - Report the precise location of an error
  - Recover from an error and continue with the parsing
- Two typical error recovery strategies
  - Panic-mode recovery (恐慌模式)
  - Phrase-level recovery (短语层次的恢复)

Panic-Mode Recovery

- **Basic idea:** Discard zero or more input symbols until a synchronization token (同步词法单元) is found
- **Rationale:**
  - The parser always looks for a prefix of the input that can be derivable from a non-terminal A
  - When there is an error, it means it is impossible to find such a prefix
  - If errors only occur in the part related to A, we can skip the part by looking for a symbol that can legitimately follow A
    - **Example:** If A is *stmt*, then the synchronization symbol can be a semicolon

Phrase-Level Recovery

- **Basic idea:**
  - Examine each error entry in the parsing table and decide the most likely programmer error that would give rise to the error
  - Modify the top of the stack or first input symbols and issue messages to programmers

STATE	ACTION						GOTO
	id	+	*	(	)	$	*E*
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

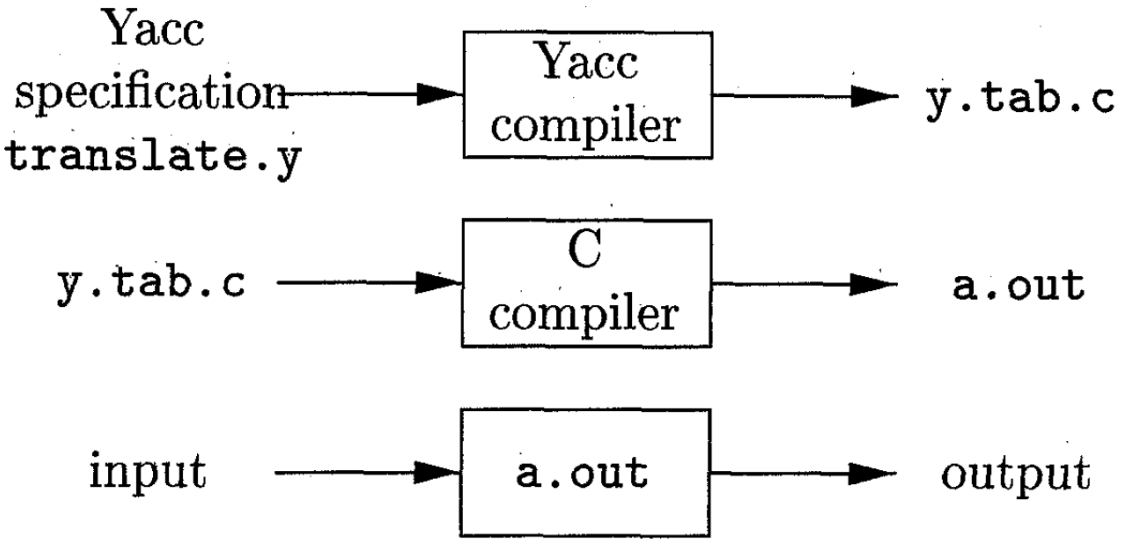
Example phrase-level recovery:

- Remove the right) from the input;
- Issue diagnostic "unbalanced right parenthesis".

Outline

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

The Parser Generator YACC/BISON



Structure of YACC Source Programs

declarations
%%
translation rules
%%
supporting C routines

- **Declarations (声明)**
  - Ordinary C declarations
  - Grammar tokens
- **Translation rules (翻译规则)**
  - Rule = a grammar production + the associated semantic action
- **Supporting C routines (辅助性C语言例程)**
  - Directly copied to `y.tab.c`
  - Can be invoked in the semantic actions
  - `yylex()` must be provided, which returns tokens
  - Other procedures such as error recovery routines may be provided

Translation Rules

$$
\begin{array}{lcll}
\langle \text{head} \rangle & : & \langle \text{body} \rangle_1 & \{\ \langle \text{semantic action} \rangle_1\ \} \\
& | & \langle \text{body} \rangle_2 & \{\ \langle \text{semantic action} \rangle_2\ \} \\
& & \cdots & \\
& | & \langle \text{body} \rangle_n & \{\ \langle \text{semantic action} \rangle_n\ \} \\
& ; & &
\end{array}
$$

- The first head is taken to be the start symbol
- A semantic action is a sequence of C statements
  - `$$` refers to the attribute value associated with the nonterminal of the head
  - `$i` refers to the value associated with *i*th grammar symbol of the body
- A semantic action is performed when we reduce by the associated production
  - We can compute a value for `$$` in terms of the `$i`'s

YACC Source Program Example

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line   : expr '\n'          { printf("%d\n", $1); }
       ;
expr   : expr '+' term      { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor    { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsing cannot be used
- Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols, cannot decide
  - Whether to shift or to reduce (shift/reduce conflicts, 移入/归约冲突)
  - Which of several possible reductions to make (reduce/reduce conflicts, 归约/归约冲突)

Shift/Reduce Conflict Example

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

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

Reduce or shift? What if there is a *stmt* after **else**?

Reduce/Reduce Conflict Example

- Parsing input **id(id, id)**

STACK	INPUT
$id(id	, id$

(1) *stmt* → **id** (*parameter_list*)
(2) *stmt* → *expr* := *expr*
(3) *parameter_list* → *parameter_list* , *parameter*
(4) *parameter_list* → *parameter*
(5) *parameter* → **id**
(6) *expr* → **id** (*expr_list*)
(7) *expr* → **id**
(8) *expr_list* → *expr_list* , *expr*
(9) *expr_list* → *expr*

Reduce by which production?

Conflicts Resolution in YACC

- **Default strategy:**
  - **Shift/reduce conflicts:** always shift
  - **Reduce/reduce conflicts:** reduce with the production listed first
- **Specifying the precedence and associativity of terminals**
  - **Associativity:** `%left`, `%right`, `%nonassoc`
  - **Shift a/reduce $A \rightarrow \alpha$ conflict:** compare the precedence of a and $A \rightarrow \alpha$ (use associativity when precedence is not enough)
    - The declaration order of terminals determine their precedence
    - The precedence of a production is equal to the precedence its rightmost terminal. It can also be specified using `%prec<terminal>`, which defines the precedence of the production to be the same as the terminal

Error Recovery in YACC

- In YACC, error recovery uses a form of error productions
  - General form: $A \rightarrow$ **error** α
  - The users can decide which nonterminals (e.g., those generating expressions, statements, blocks, etc.) will have error productions
- **Example: *stmt* → error ;**
  - When the parser encounters an error, it would skip just beyond the next semicolon and assumes that a statement had been found
  - The semantic action of the error production will be invoked: it would not need manipulate the input, but could simply generate a diagnostic message

Reading Tasks

- Chapter 4 of the dragon book
  - 4.1 Introduction
  - 4.2 Context-Free Grammars
  - 4.3 Writing a Grammar (4.3.1 – 4.3.4)
  - 4.4 Top-Down Parsing (4.4.1 – 4.4.4)
  - 4.5 Bottom-Up Parsing
  - 4.6 Simple LR
  - 4.7 More Powerful LR Parsers (4.7.1 – 4.7.4)
  - 4.8 Using Ambiguous Grammars
  - 4.9 Parser Generators