北京郵電大學
BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Chapter 3: Syntax Analysis

Yanhui Guo

yhguo@bupt.edu.cn

Outline

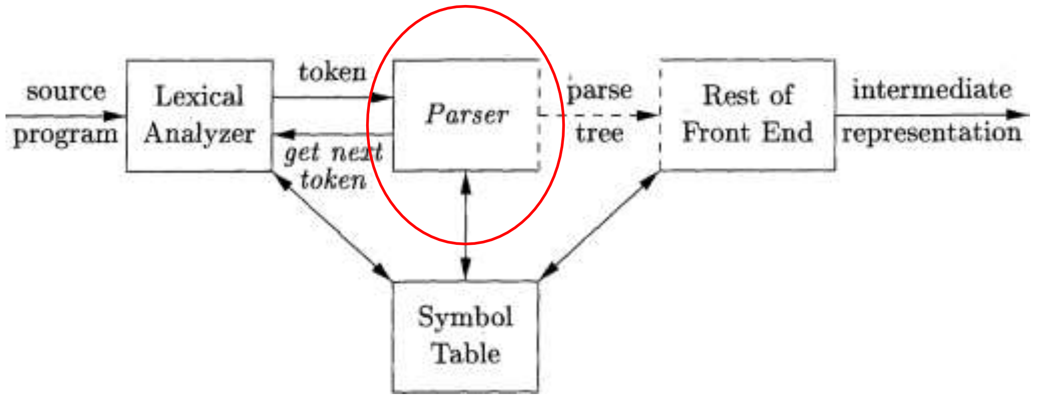
- Introduction
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (to be discussed in lab sessions)

Describing Syntax

- The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation
    - A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language
    - For certain grammars, we can automatically construct an efficient parser
    - A properly designed grammar defines the structure of a language and helps translate source programs into correct object code and detect errors
    - A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks

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

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

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 the manner in which the terminals and nonterminals can be combined to form strings
    - **Format:** `head (left side)` → `body (right side)`
    - The head is a nonterminal; the 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*, *factor*
  - Start symbol: *expression*
  - Productions:
    - *expression* → *expression* + *term*
    - *expression* → *expression* – *term*
    - *expression* → *term*
    - *term* → *term* * *factor*
    - *term* → *term* / *factor*
    - *term* → *factor*
    - *factor* → (*expression*)
    - *factor* → **id**

Notational Simplification

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

⇨

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

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

Outline

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

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

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

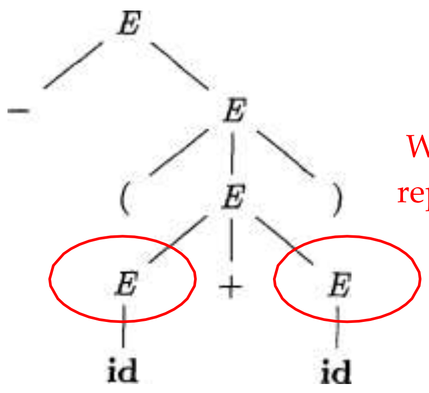
Parse Trees (语法分析树)

- 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 or ϵ
  - Each interior node (内部结点) is labeled by a nonterminal symbol
  - Each interior node 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: $E \rightarrow -\ E \mid E\ +\ E \mid E\ *\ E \mid (\ E\ ) \mid \mathbf{id}$

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

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



Which one is replaced first?

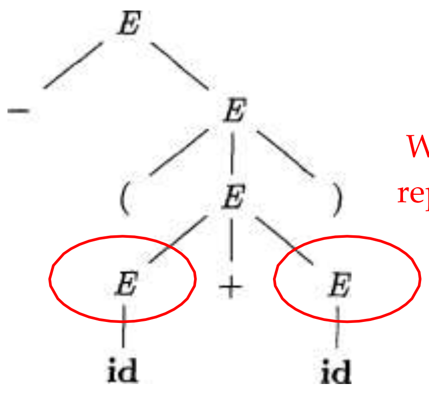
Parse Trees (语法分析树) Cont.

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

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

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

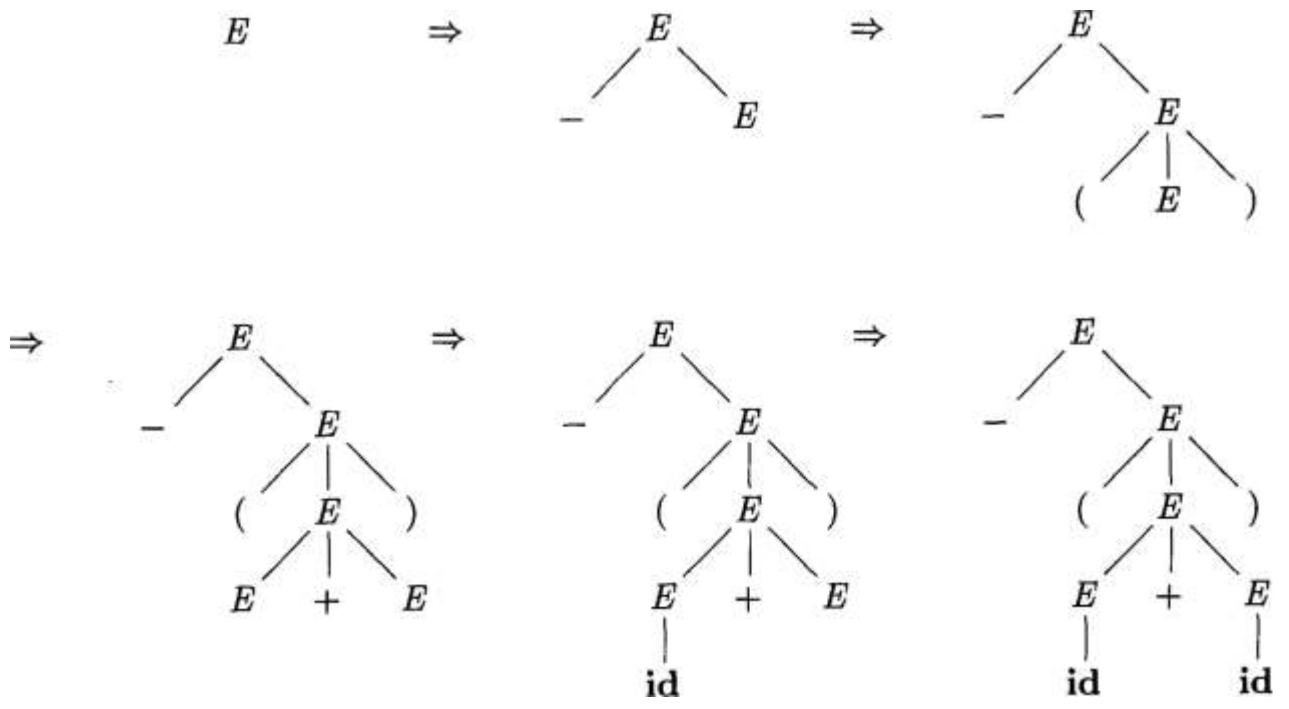
$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\textbf{id}) \underset{rm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$



Which one is replaced first?

Constructing Parse Trees (Example)

Derivation: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$



Outline

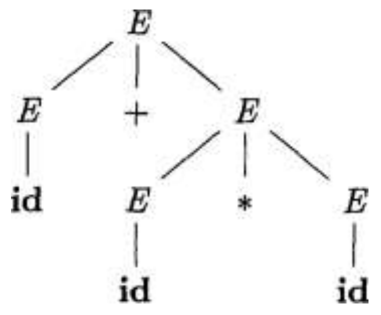
- Introduction
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (to be discussed in lab sessions)

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

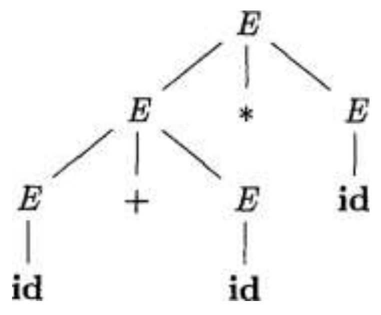
Ambiguity (二义性)

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

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



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



Both are leftmost derivations

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

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

CFG vs. Regular Expressions

- **CFGs are more expressive than regular expressions**
  - Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa
  - Every regular language is a context-free language, but not vice-versa

- 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^+ from s to an accepting state
- Since a^i reaches the state s and there is a path labeled b^+ 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

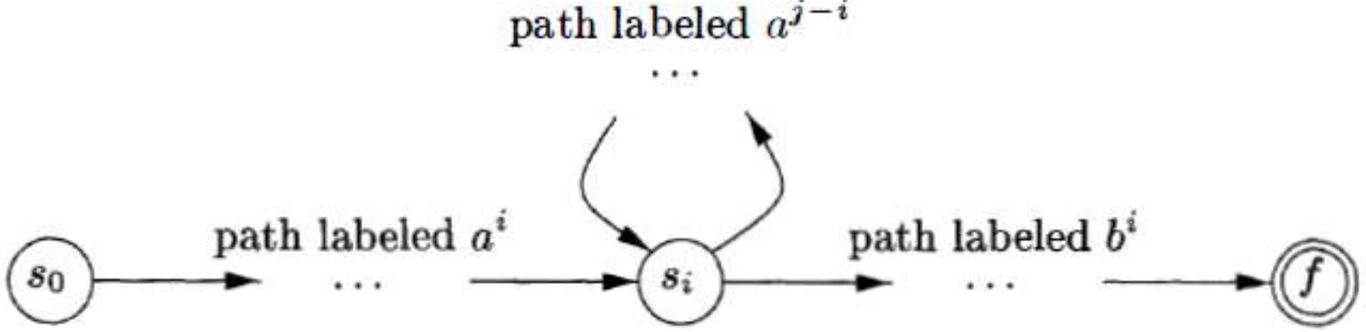


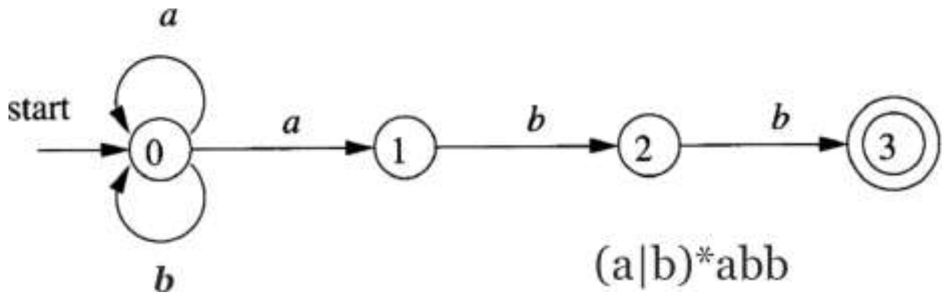
Figure 4.6: DFA D accepting both a^ib^i and a^jb^i.

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 $A_i \rightarrow aA_\mathrm{j}$
  - If state i goes to state j on input ϵ, add the production $A_i \rightarrow A_\mathrm{j}$
  - 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

Outline

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

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

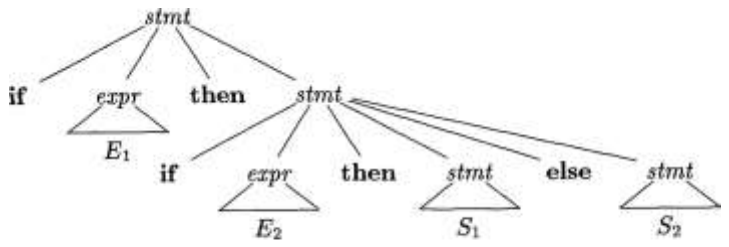
Grammar Design

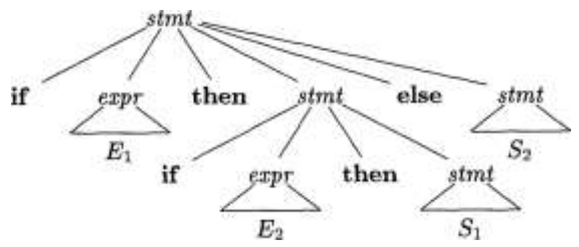
- CFGs are capable of describing most, but not all, of the syntax of programming languages
  - "Identifiers should be declared before use" cannot be described by a CFG
  - Subsequent phases must analyze the output of the parser to ensure compliance with such rules
- Before parsing, we typically apply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

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

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`)

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

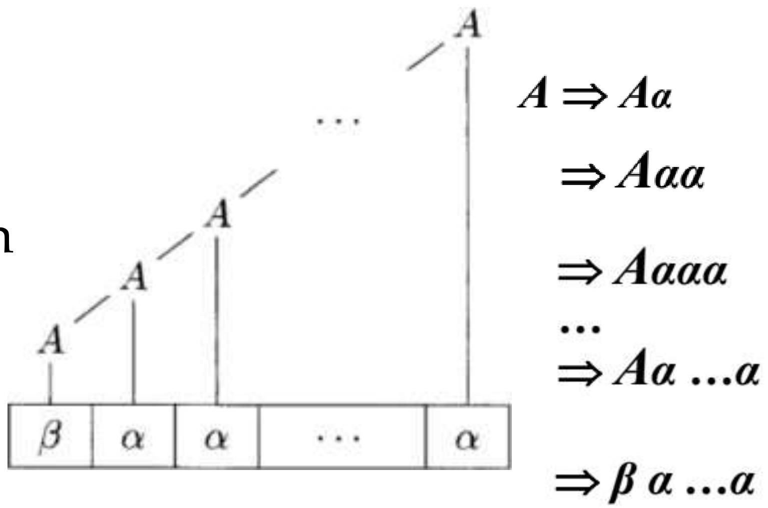
Rewriting grammars to eliminate ambiguity is difficult.
There are no general rules to guide the process.

Eliminating Left Recursion

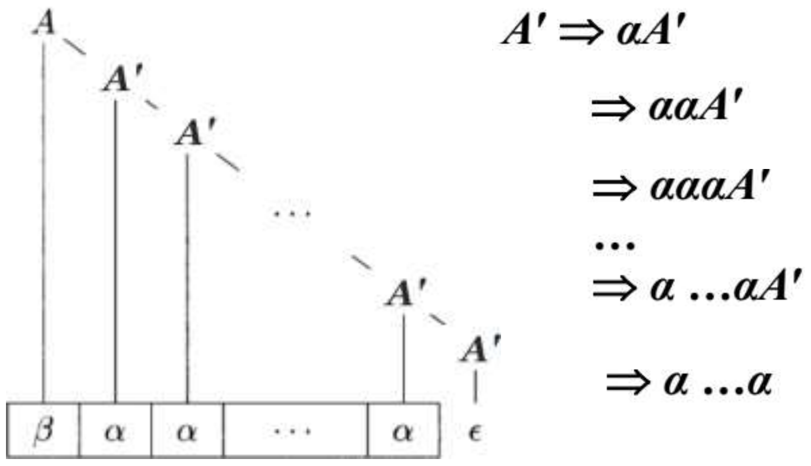
- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \to Aa \mid b$
  - $A \to Ac \mid Sd \mid \epsilon$
  - Because $S \Longrightarrow Aa \Longrightarrow Sda$
- **Immediate left recursion (立即左递归):**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

Eliminating Immediate Left Recursion

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



A
A
A
A
β
α
α
...
α
$A \Rightarrow A\alpha$
$\Rightarrow A\alpha\alpha$
$\Rightarrow A\alpha\alpha\alpha$
...
$\Rightarrow A\alpha \ldots \alpha$
$\Rightarrow \beta\, \alpha \ldots \alpha$



A
A'
A'
A'
A'
β
α
α
...
α
ϵ
$A' \Rightarrow \alpha A'$
$\Rightarrow \alpha\alpha A'$
$\Rightarrow \alpha\alpha\alpha A'$
...
$\Rightarrow \alpha \ldots \alpha A'$
$\Rightarrow \alpha \ldots \alpha$

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

$$E \rightarrow E + \underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$$

$$T \rightarrow T \underbrace{* F}_{\alpha} \mid \underbrace{F}_{\beta}$$

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

$$\Rightarrow$$

$$E \rightarrow T E'$$

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

$$T \rightarrow F T'$$

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

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

$$\boxed{E \rightarrow E + T \mid T}$$

$$\boxed{T \rightarrow T * F \mid F}$$

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

$$\Longrightarrow$$

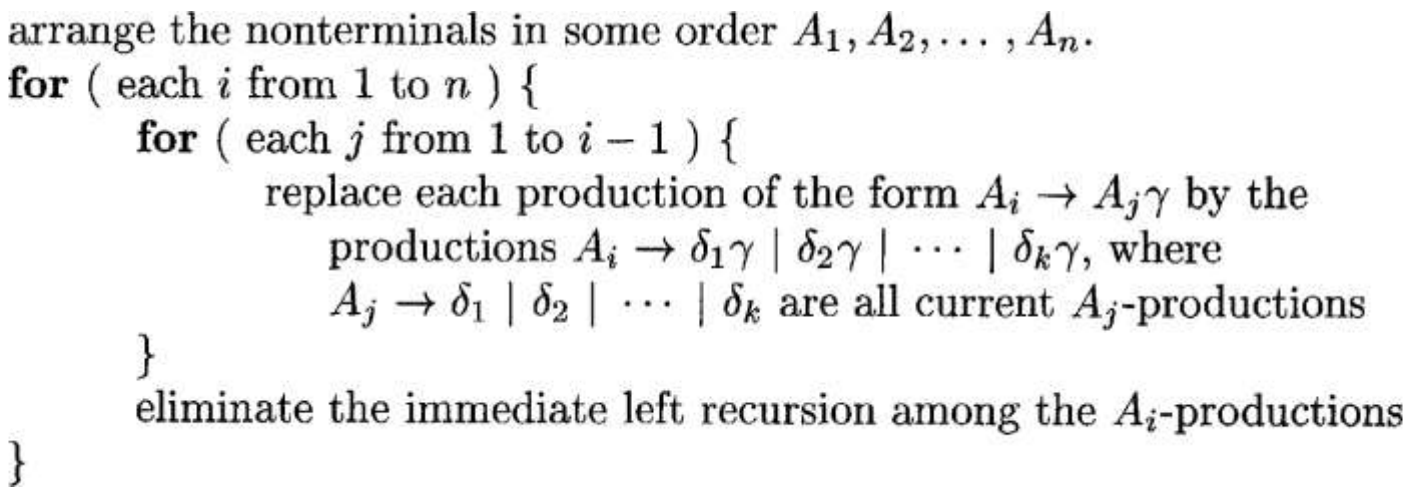
$$\boxed{\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow + \ TE' \mid \epsilon \end{array}}$$

$$\boxed{\begin{array}{l} T \rightarrow FT' \\ T' \rightarrow * \ FT' \mid \epsilon \end{array}}$$

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

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

Example

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

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

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$

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

Left Factoring (提取左公因子)

- If we have the following two productions

$$\begin{aligned} stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ &\mid\ \textbf{if}\ expr\ \textbf{then}\ stmt \end{aligned}$$

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

$$\boxed{\begin{aligned} &A \rightarrow \alpha A' \\ &A' \rightarrow \beta_1 \mid \beta_\& \end{aligned}}$$

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 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_{\&} \mid \ldots \mid \alpha\beta_n \mid \gamma$, 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
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (to be discussed in lab sessions)

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)
  - Equivalent to finding a leftmost derivation for an input string

- **Basic idea (two actions):**
  - **Predict:** At each step of parsing, determine the production to be applied for the leftmost nonterminal
  - **Match:** Match the terminals in the chosen production's body with the input string

Top-Down Parsing Example

- **Grammar**

  $E \rightarrow TE'$

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

  $T \rightarrow FT'$

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

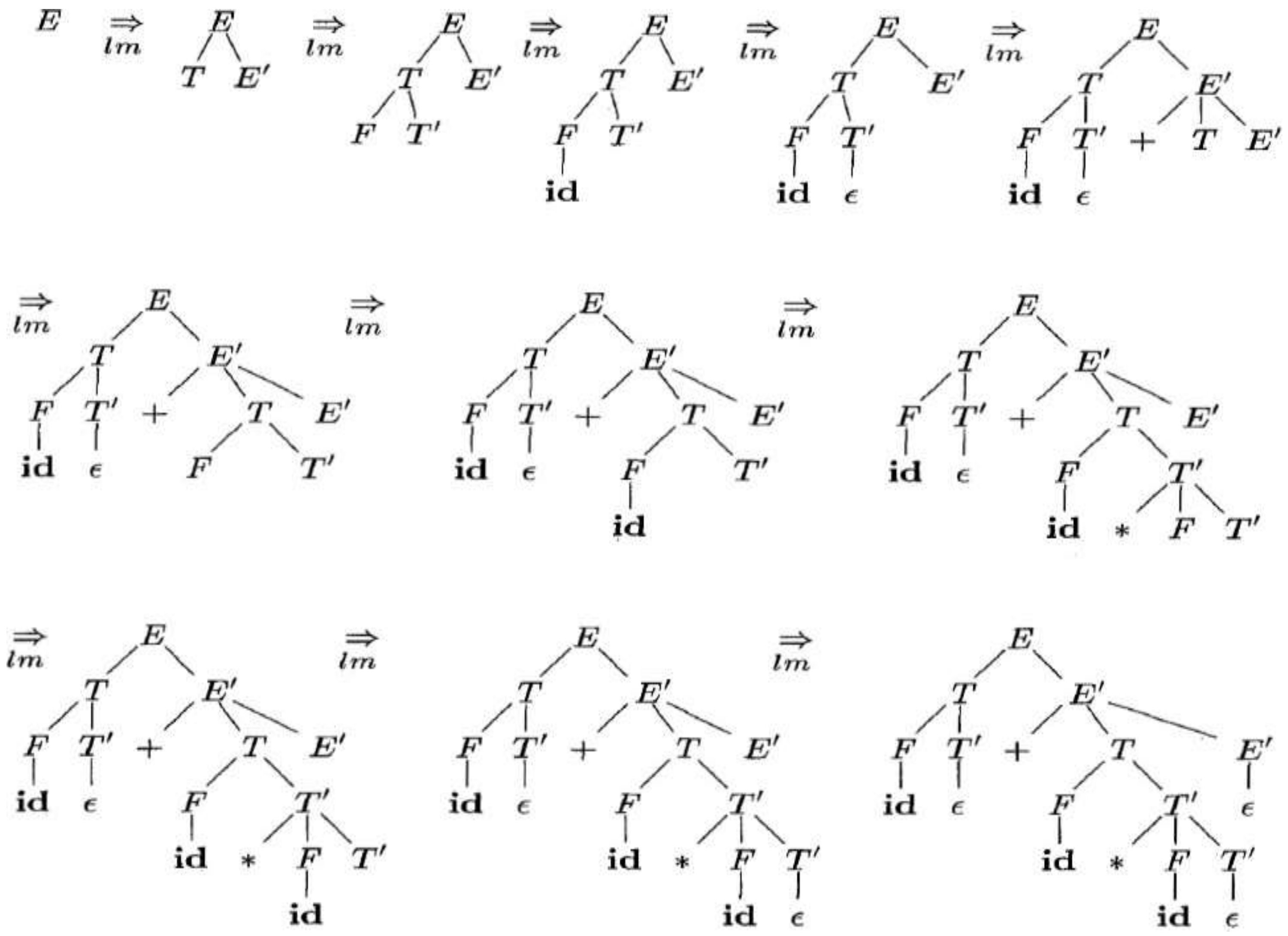
  $F \rightarrow (E) \mid id$

- **Input string**

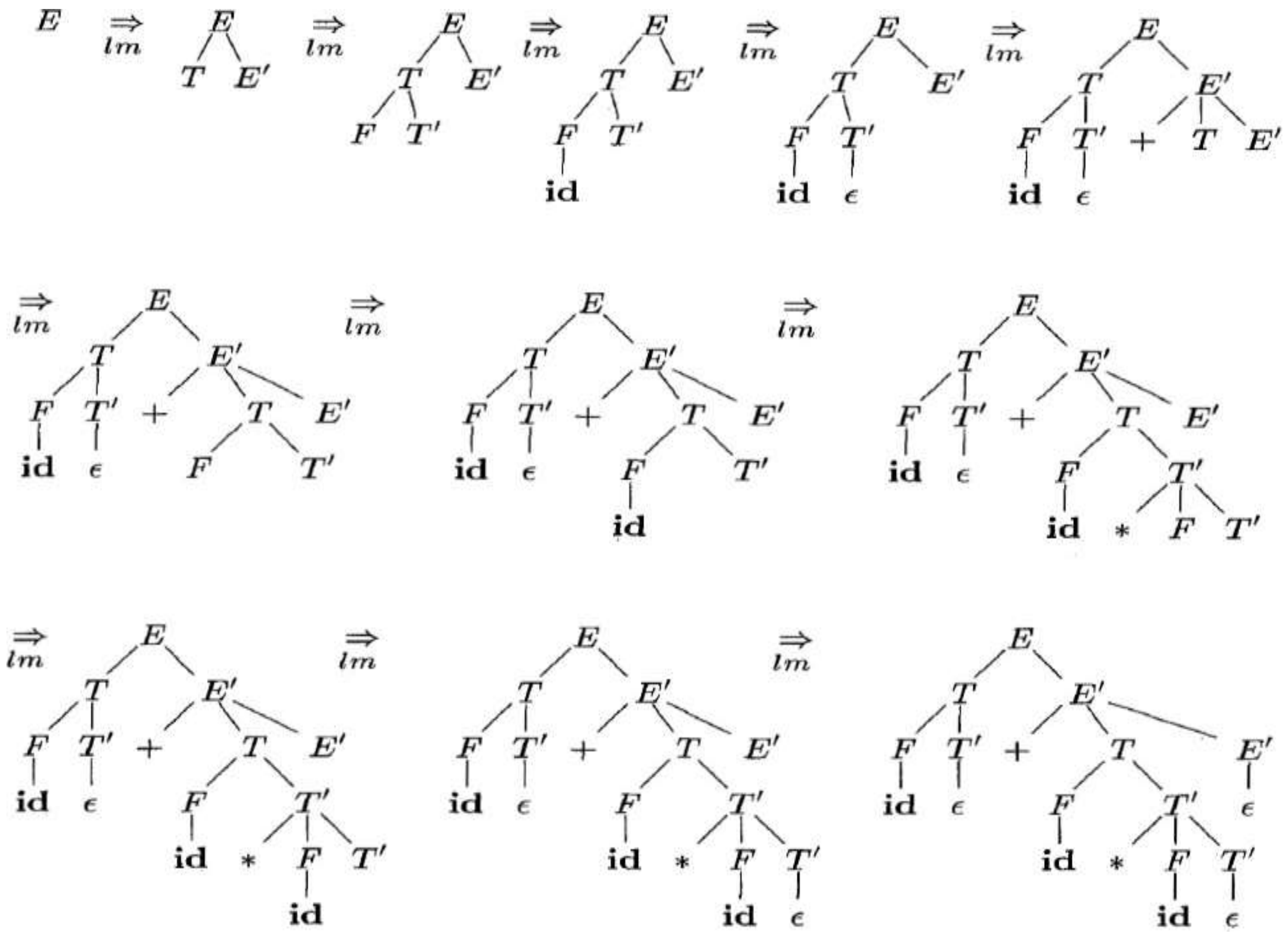
  $\mathbf{id + id * id}$

Is the input string a sentence of the grammar?

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



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



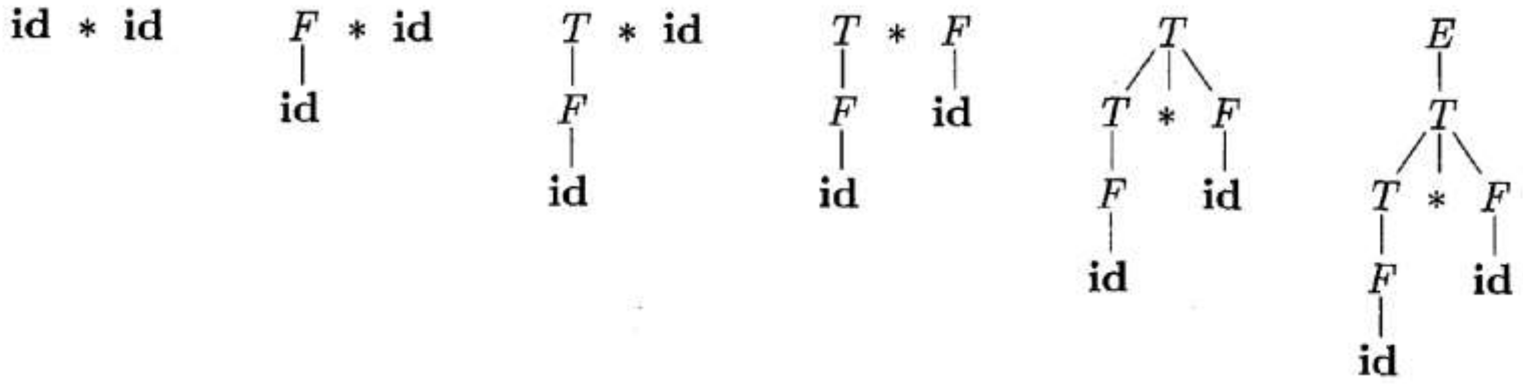
Bottom-Up Parsing

$E \to E + T \mid T$

$T \to T * F \mid F$

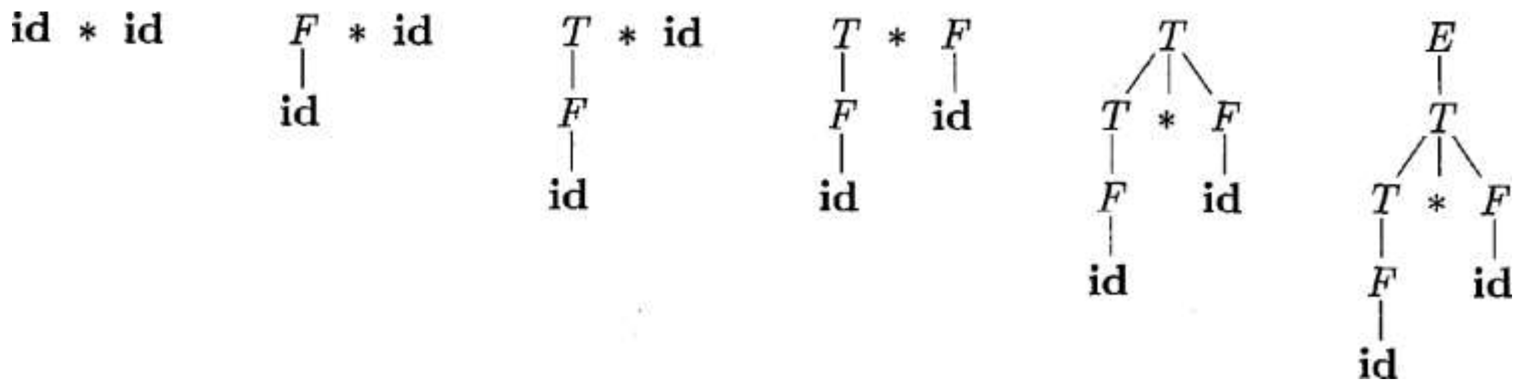
$F \to (E) \mid \mathbf{id}$

- **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)
  - Equivalent to finding a rightmost derivation (in reverse) for an input string
- Shift-reduce parsing (移入-归约分析技术) is a general style of bottom-up parsing (using a stack to hold grammar symbols)



Reductions (归约)

- Bottom-up parsing can be seen as a process of "reducing" a string ω to the start symbol of the grammar
- At each *reduction* step, a specific substring (at the top of the stack) matching the body of a production is replaced by the head of the production (the reverse of a step in 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
$\$\ \mathbf{id}_1$	$* \mathbf{id}_2\ \$$	reduce by $F \to \mathbf{id}$
$\$\ F$	$* \mathbf{id}_2\ \$$	reduce by $T \to F$
$\$\ T$	$* \mathbf{id}_2\ \$$	shift
$\$\ T *$	$\mathbf{id}_2\ \$$	shift
$\$\ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\ T * F$	$\$$	reduce by $T \to T * F$
$\$\ T$	$\$$	reduce by $E \to T$
$\$\ E$	$\$$	accept

$E \to E + T \mid T$

$T \to T * F \mid F$

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

Key decisions:

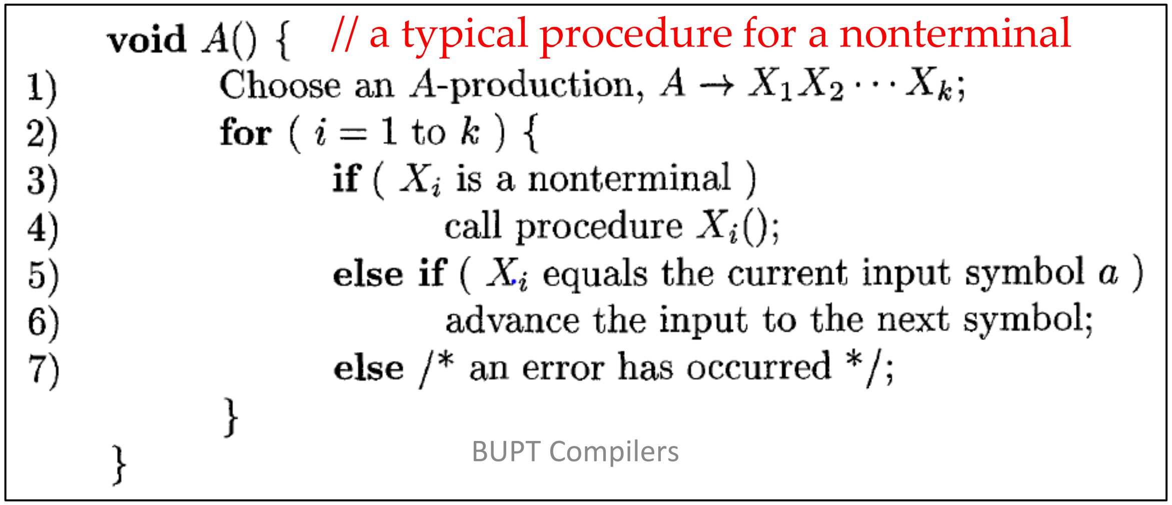
1. When to reduce?
2. Which production to apply?

Outline

- Introduction
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
    - Recursive-descent parsing
    - Non-recursive predictive parsing
- Bottom-Up Parsing
- Parser Generators (to be discussed in lab sessions)

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 scans the structure (a substring of the input) corresponding to the nonterminal
- Execution begins with the procedure for the start symbol
  - Announce success if the procedure scans the entire input string



```
void $A$() {   // a typical procedure for a nonterminal
    Choose an $A$-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ( $i = 1$ to $k$ ) {
        if ( $X_i$ is a nonterminal )
            call procedure $X_i()$;
        else if ( $X_i$ equals the current input symbol $a$ )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Backtracking (回溯)

```
     void $A()$ {
1)       Choose an $A$-production, $A \to X_1 X_2 \cdots X_k$;
2)       for ( $i = 1$ to $k$ ) {
3)           if ( $X_i$ is a nonterminal )
4)               call procedure $X_i()$;
5)           else if ( $X_i$ 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, $A \rightarrow X_1 X_2 \cdots X_k$;
2)     for ( $i = 1$ to $k$ ) {
3)         if ( $X_i$ is a nonterminal )
4)             call procedure $X_i()$;
5)         else if ( $X_i$ 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, $A \rightarrow X_1 X_2 \cdots X_k$;
2)      for ( $i = 1$ to $k$ ) {
3)          if ( $X_i$ is a nonterminal )
4)              call procedure $X_i()$;
5)          else if ( $X_i$ 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)

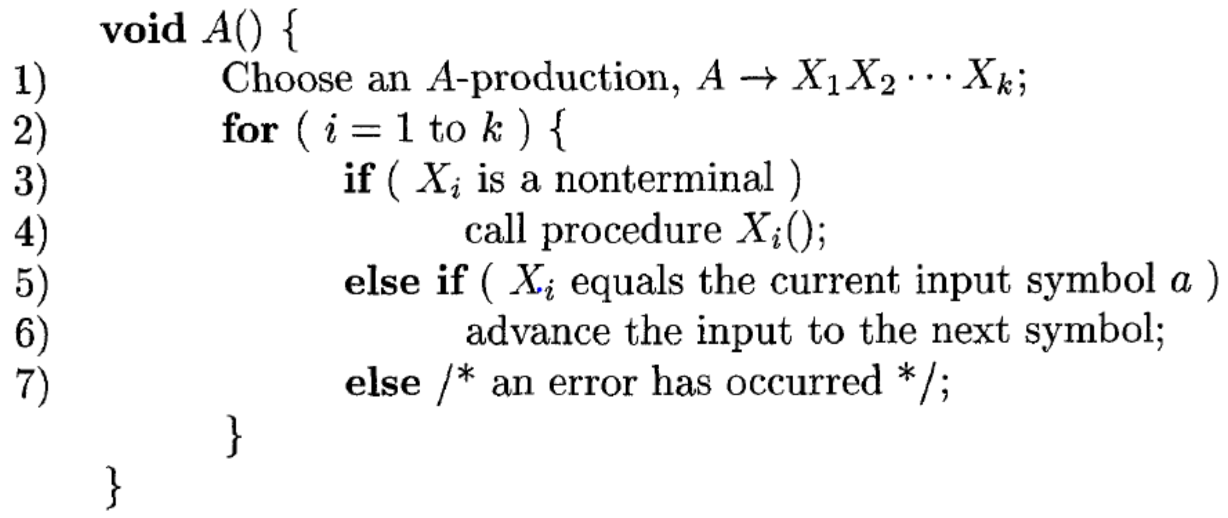
- General recursive-descent parsing may require repeated scans over the input (backtracking)
- To allow backtracking, we need to modify the algorithm

```
        void $A()$ {
1)          Choose an $A$-production, $A \rightarrow X_1 X_2 \cdots X_k$;
2)          for ( $i = 1$ to $k$ ) {
3)              if ( $X_i$ is a nonterminal )
4)                  call procedure $X_i()$;
5)              else if ( $X_i$ 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

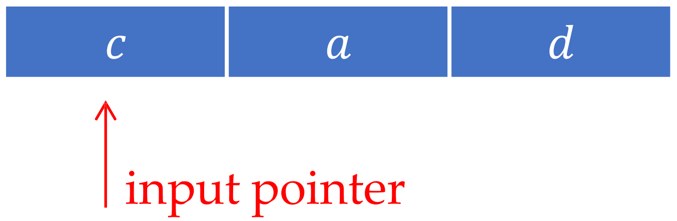


```
void $A()$ {
    Choose an $A$-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ( $i = 1$ to $k$ ) {
        if ( $X_i$ is a nonterminal )
            call procedure $X_i()$;
        else if ( $X_i$ equals the current input symbol $a$ )
            advance the input to the next symbol;
        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)

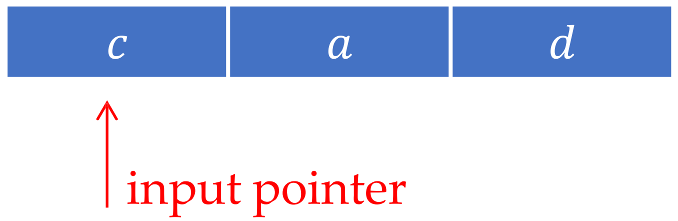
Backtracking Example

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

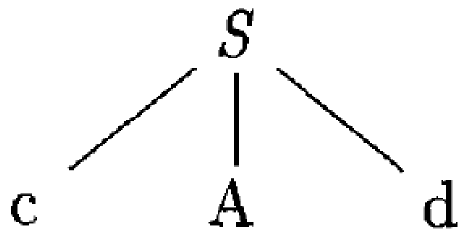


Backtracking Example

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

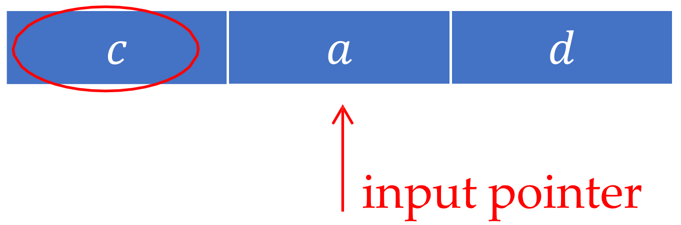


– S has only one production, apply it

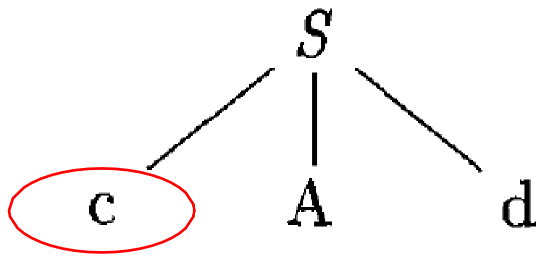


Backtracking Example

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

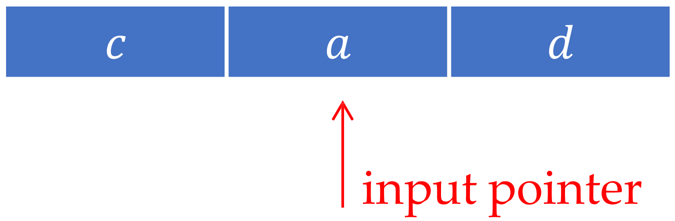


- The leftmost leaf matches c in input
- Advance input pointer

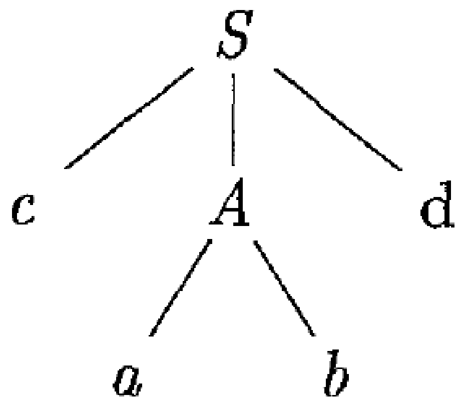


Backtracking Example

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

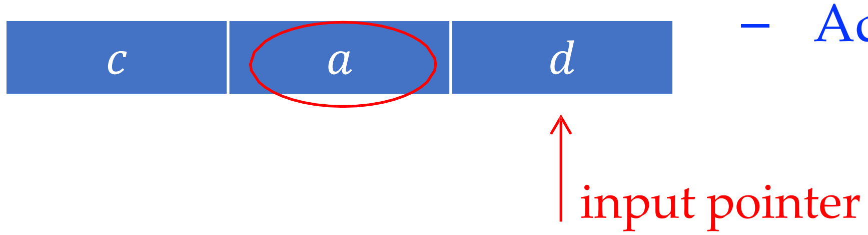


– Expand A using the first production

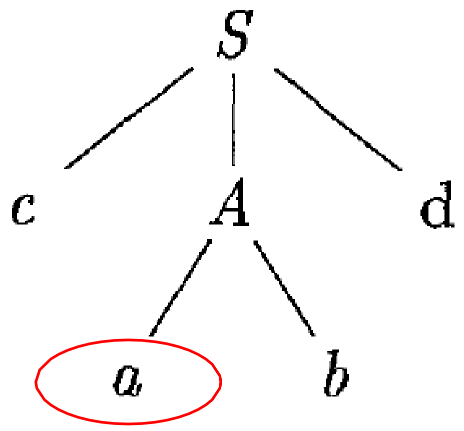


Backtracking Example

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

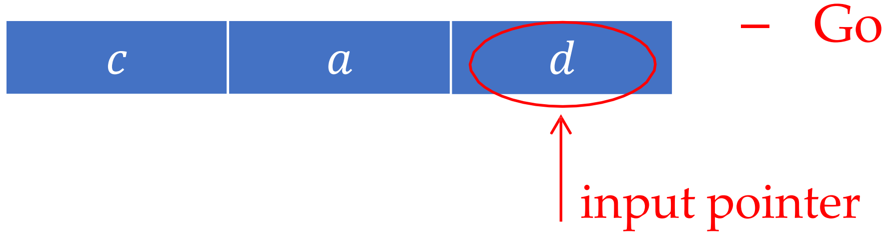


- Leftmost leaf matches a in input
- Advance input pointer

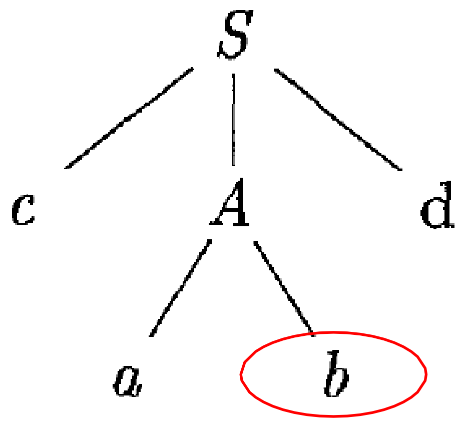


Backtracking Example

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

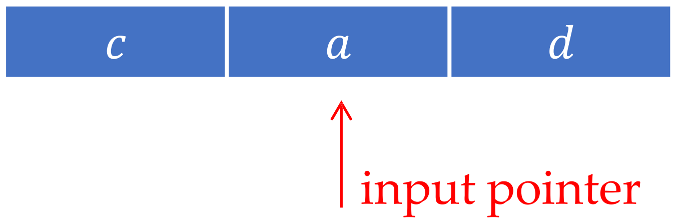


- Symbol mismatch
- Go back to try another *A*-production

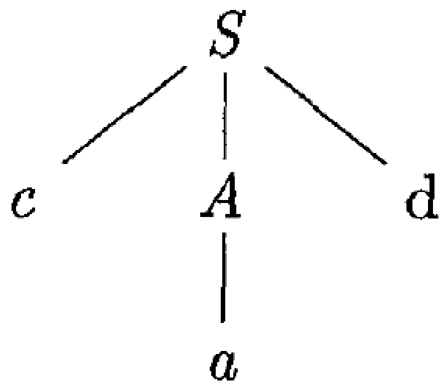


Backtracking Example

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

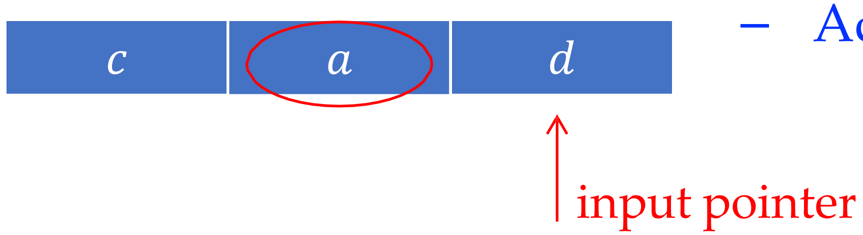


- Reset input pointer
- Expand A using its second production

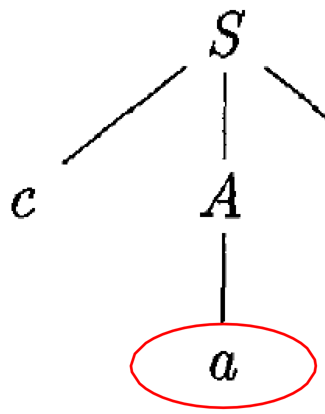


Backtracking Example

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

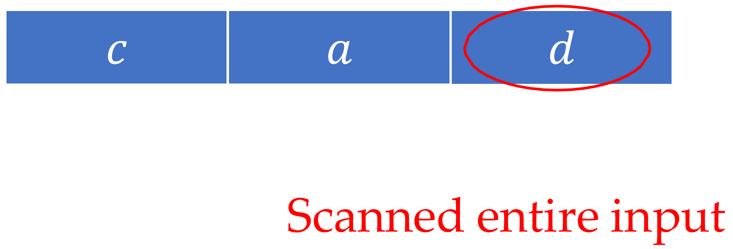


- Leftmost leaf matches a in input
- Advance input pointer

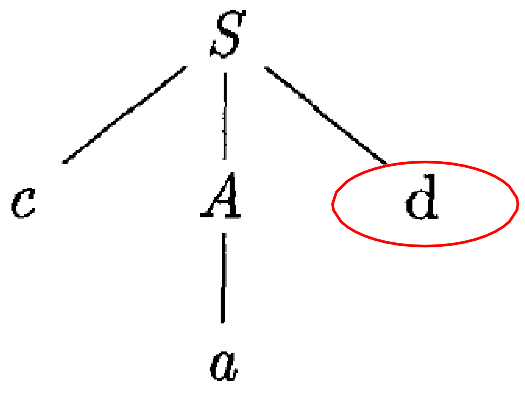


Backtracking Example

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



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



The Problem of Left Recursion

Suppose there is $A \rightarrow \boldsymbol{A}\alpha$ ...

```
    void A() {
1)      Choose an A-production, $\boxed{A \rightarrow X_1 X_2 \cdots X_k}$;
2)      for ( $i = 1$ to $k$ ) {
3)          if ( $X_i$ is a nonterminal )
4)              call procedure $X_i()$;
5)          else if ( $X_i$ equals the current input symbol $a$ )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

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

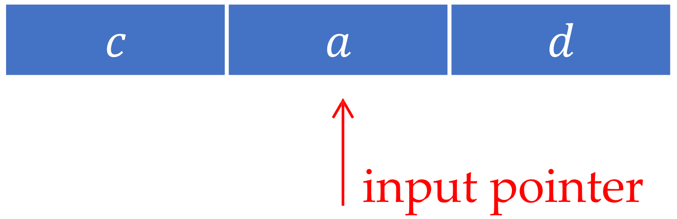
Can We Avoid Backtracking?

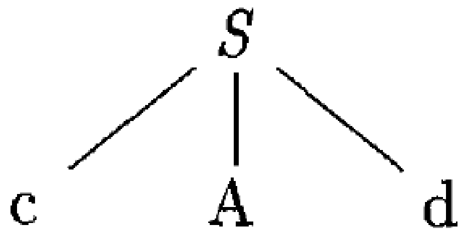
```
    void $A()$ {
1)      Choose an $A$-production, $A \rightarrow X_1 X_2 \cdots X_k$;
2)      for ( $i = 1$ to $k$ ) {
3)          if ( $X_i$ is a nonterminal )
4)              call procedure $X_i()$;
5)          else if ( $X_i$ equals the current input symbol $a$ )
6)              advance the input 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 $\boldsymbol{xa}$
- Suppose the current sentential form is $\boldsymbol{xA}$β
  - $\boldsymbol{A}$ is a non-terminal; β may contain both terminals and non-terminals

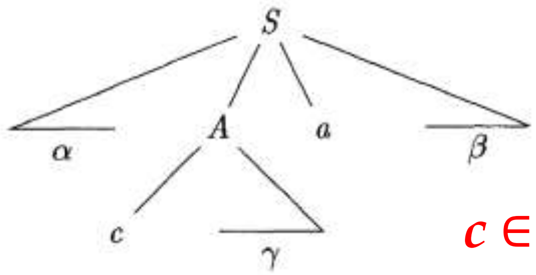
If we know the following fact for the productions $\boldsymbol{A} \rightarrow \alpha |\, \boldsymbol{\gamma}$:

- $a \in FIRST(\alpha)$: 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 terminals that begin strings derived from α

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

FIRST



$c \in$ **FIRST(A)**

- **FIRST(α)**, where α is any string of grammar symbols
  - The set **of terminals** that begin strings derived from α
  - If $\alpha \Rightarrow^* \epsilon$, then ϵ is also in FIRST(α)
- **FIRST function is useful in parsing**
  - $\boldsymbol{A} \rightarrow \alpha |$ β; suppose FIRST(α) and FIRST(β) are disjoint
  - Choose$\boldsymbol{A} \rightarrow \alpha$ if the next input symbol a FIRST(α)
  - Choose$\boldsymbol{A} \rightarrow$ β if the next input symbol a FIRST(β)

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 $X \rightarrow Y_1 Y_2 \ldots Y_k$ $(k \geq 1)$ 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)
  - If X is a nonterminal and $X \rightarrow \epsilon$, then add ϵ to FIRST(X)

Computing FIRST Cont.

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

FIRST Example

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

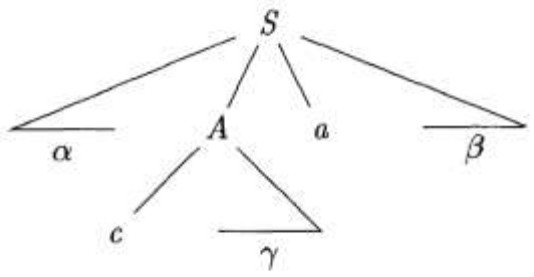
Looking Ahead Helps Cont.

- Suppose the input string is $\boldsymbol{xa}$
- Suppose the current sentential form is $\boldsymbol{xA}$β
  - $\boldsymbol{A}$ is a non-terminal; β 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?

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

FOLLOW



- **FOLLOW(A)**, where A is a nonterminal

a ∈**FOLLOW(A)**

  - The set **of** **terminals** a that can appear immediately to the right of A in some sentential form

- **FOLLOW function is also useful in parsing**
  - Consider a production $\boldsymbol{A} \rightarrow \alpha$; when $\alpha \overset{*}{\Rightarrow} \epsilon$, FOLLOW(A) can help choose the right production
  - Example: if the next input symbol is b , and b ∈ FOLLOW(A), then we can choose $\boldsymbol{A} \rightarrow \alpha$

Computing FOLLOW

- **Computing FOLLOW(A) for all nonterminals A**
  - 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)

ϵ will not appear in any FOLLOW set

FOLLOW Example

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

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

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

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

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
  - 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 $\text{FOLLOW}(A)$ and vice versa

* The three conditions basically rule out the possibility of applying both productions

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

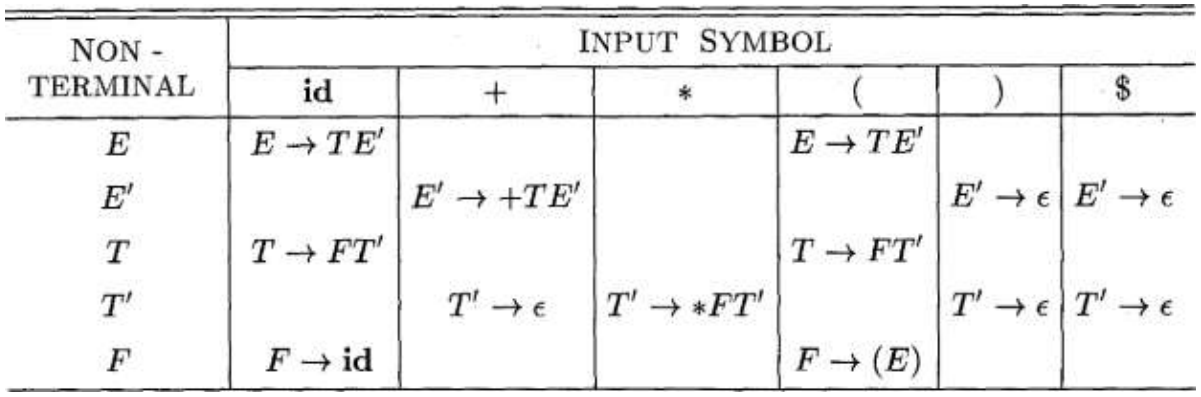
Grammar: $stmt \rightarrow \underbrace{\textbf{if}\,(\textbf{expr})\ stmt\ \textbf{else}\ stmt}_{①} \mid \underbrace{\textbf{while}(\textbf{expr})\ stmt}_{②} \mid \underbrace{\textbf{a}}_{③}$

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)

- Recursive-descent parsers (or **LL parsers**) are table-based 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	$F \to \textbf{id}$			$F \to (E)$		

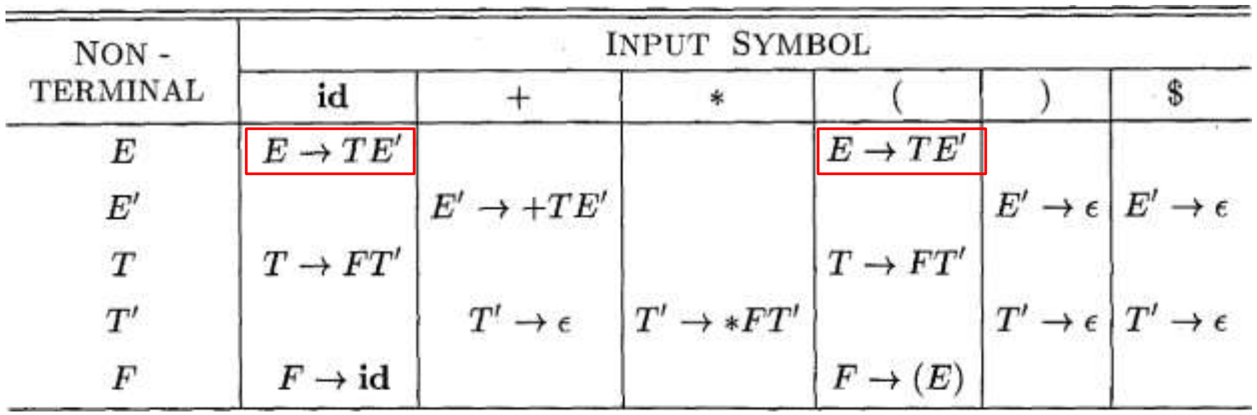
Parsing Table Construction

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 $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
    - If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b (including the right endmarker \$) in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$
  - Set all empty entries in the table to **error**

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: $\{(, \mathbf{id}\}$ $\quad E'$: $\{+, \epsilon\}$ $\quad T'$: $\{*, \epsilon\}$
- **FOLLOW sets:** E, E': $\{\$,)\}$ $\quad T, T'$: $\{+, \$,)\}$ $\quad F$: $\{*, +, \$,)\}$



NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$\boxed{E \to TE'}$			$\boxed{E \to TE'}$		
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	$F \to \mathbf{id}$			$F \to (E)$		

For $E \to TE'$:

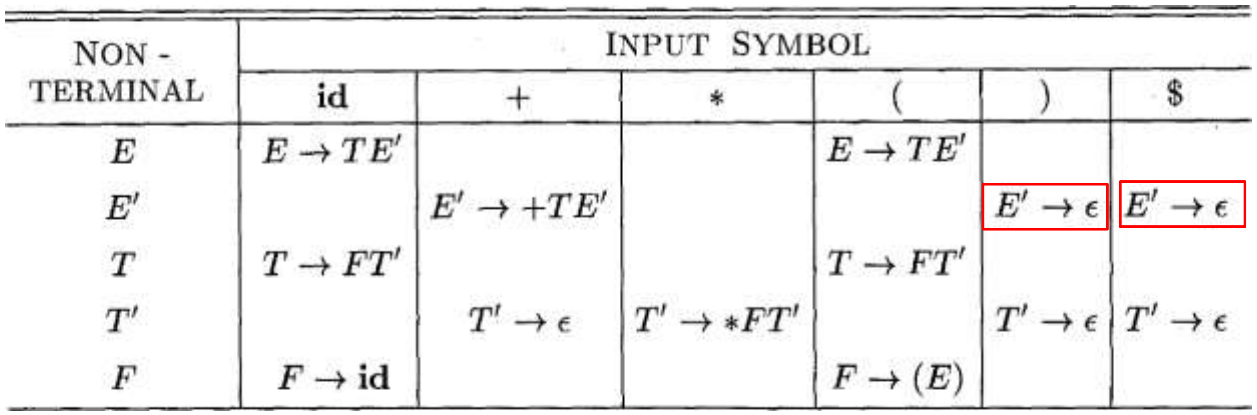
FIRST(TE')

= FIRST(T)

= $\{(, \mathbf{id}\}$

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: $\{(, \mathbf{id}\}$ $\quad E'$: $\{+, \epsilon\}$ $\quad T'$: $\{*, \epsilon\}$
- **FOLLOW sets:** E, E': $\{\$,)\}$ $\quad T, T'$: $\{+, \$,)\}$ $\quad F$: $\{*, +, \$,)\}$



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

For $E' \to \epsilon$:

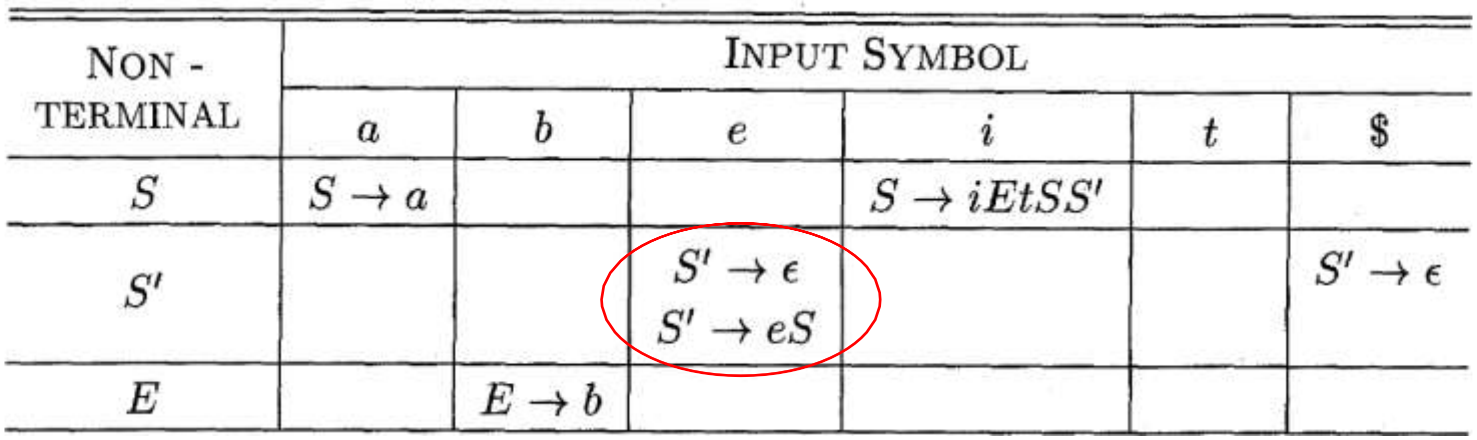
ϵ in FIRST(ϵ)

FOLLOW(E')

= $\{\$,)\}$

Conflicts in Parsing Tables

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

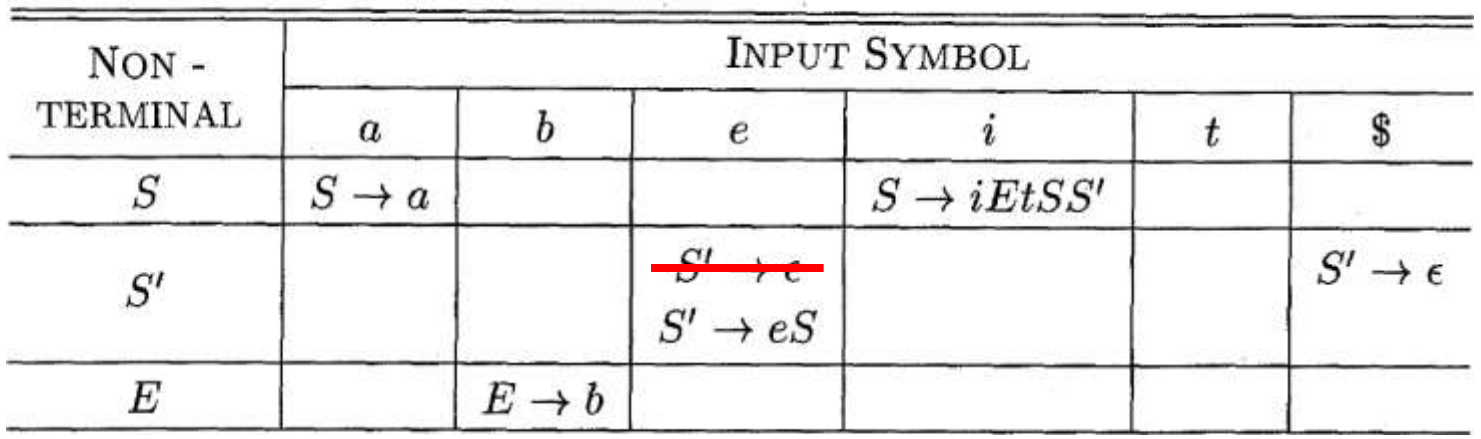


NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

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

Conflicts in Parsing Tables

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



Non - terminal	Input Symbol					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			~~$S' \to \epsilon$~~ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Solution: Choose $S' \to es$ to resolve ambiguity (associating an **`else`** with the closest previous **`then`**)

Recursive-Descent Parsing for LL(1) Grammars

```
    void $A()$ {
1)      Choose an $A$-production, $A \rightarrow X_1X_2 \cdots X_k$;
2)      for ( $i = 1$ to $k$ ) {
3)          if ( $X_i$ is a nonterminal )
4)              call procedure $X_i()$;
5)          else if ( $X_i$ 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
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
    - Recursive-descent parsing
    - Non-recursive predictive parsing
- Bottom-Up Parsing
- Parser Generators (to be discussed in lab sessions)

Non-Recursive Predictive Parsing

```
   void $A()$ {
1)     Choose an $A$-production, $A \rightarrow X_1 X_2 \cdots X_k$;
2)     for ( $i = 1$ to $k$ ) {
3)         if ( $X_i$ is a nonterminal )
4)             call procedure $X_i()$;
5)         else if ( $X_i$ 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?

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 $

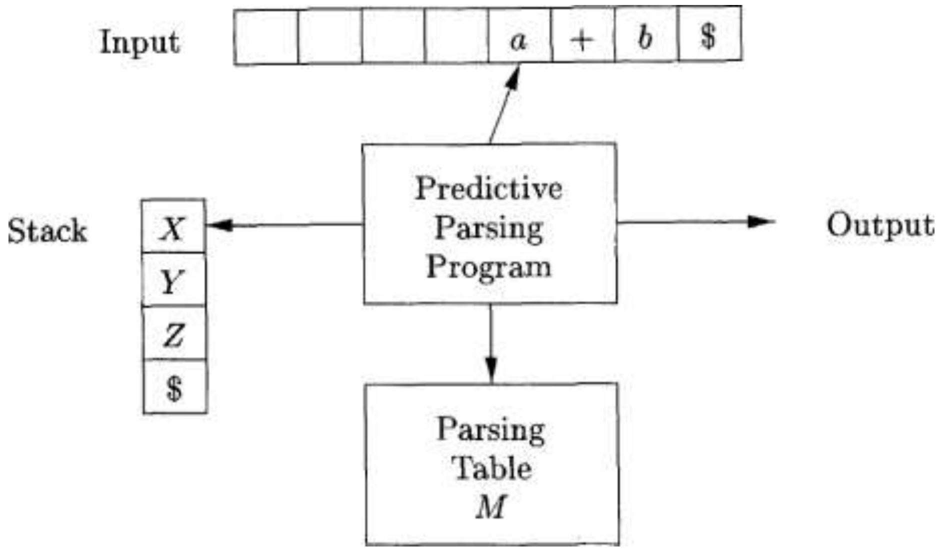
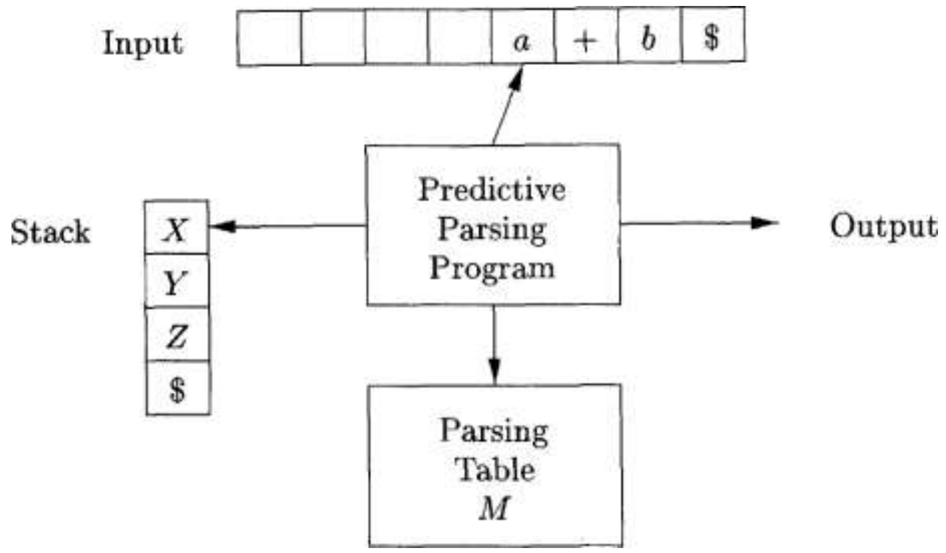


Table-Driven Predictive Parsing

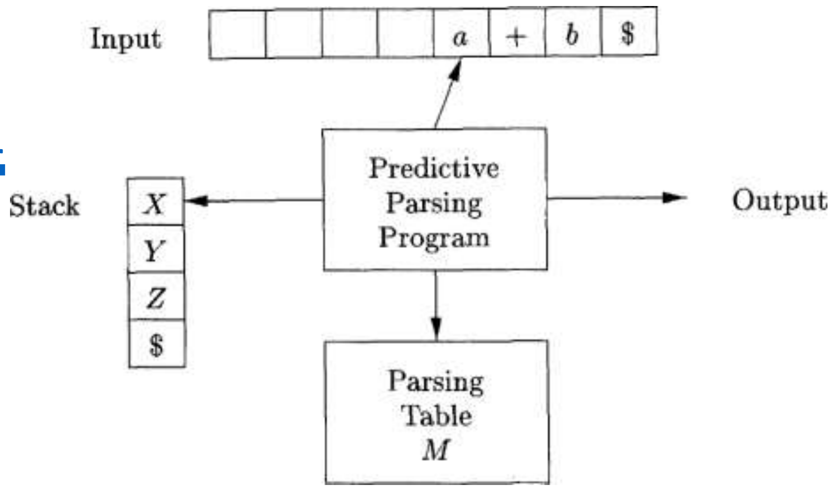
- **Input:** A string ω and a parsing table M for grammar G
- **Output:** If ω is in $L(G)$, a leftmost derivation of ω; otherwise, an error indication



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

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

Parsing Algorithm



```
1.  let $a$ be the first symbol of $\omega$;
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 $\omega$;
5.      else if ( $X$ is a terminal) error();   /* $X$ can only match $a$ */
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. }
```

Example

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

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

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

Input:

id + id * id

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

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

总是最左句型

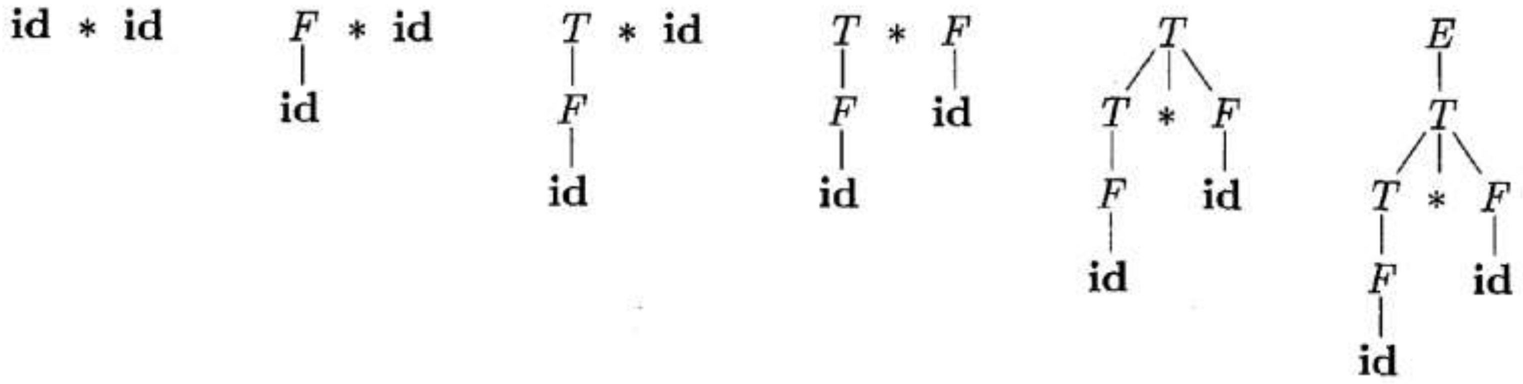
Outline

- Introduction
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing (Recall & A Detailed Look)
- Parser Generators (to be discussed in lab sessions)

Bottom-Up Parsing

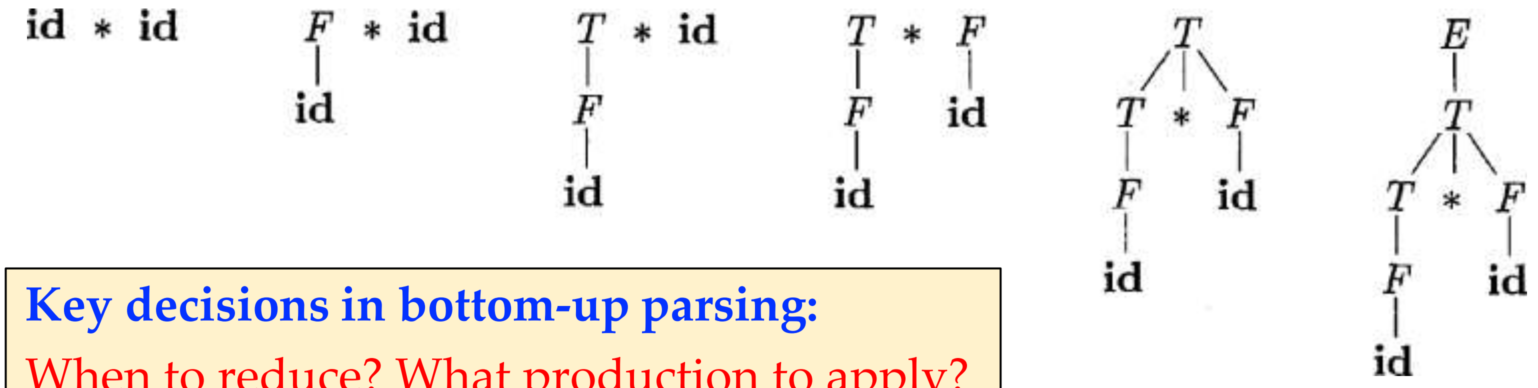
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

- **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)
  - Equivalent to finding a rightmost derivation (in reverse) for an input string
- Shift-reduce parsing (移入-归约分析技术) is a general style of bottom-up parsing (using a stack to hold grammar symbols)



Reductions (归约)

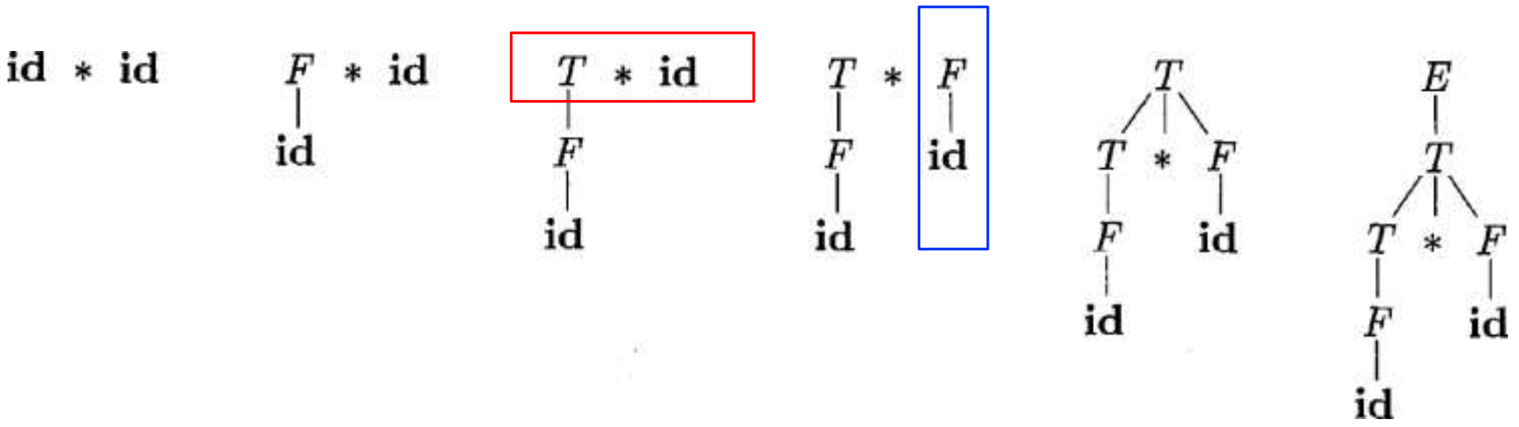
- Bottom-up parsing can be seen as a process of "reducing" a string ω to the start symbol of the grammar
- At each *reduction* step, a specific substring (at the top of the stack) matching the body of a production is replaced by the head of the production (the reverse of a step in derivation)



Key decisions in bottom-up parsing:
When to reduce? What production to apply?

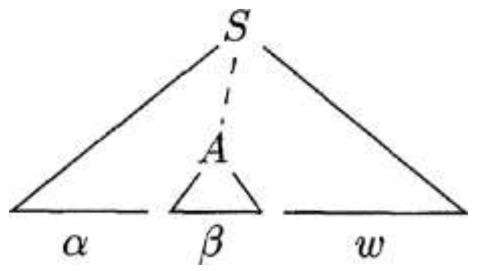
Illustration of Challenges

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

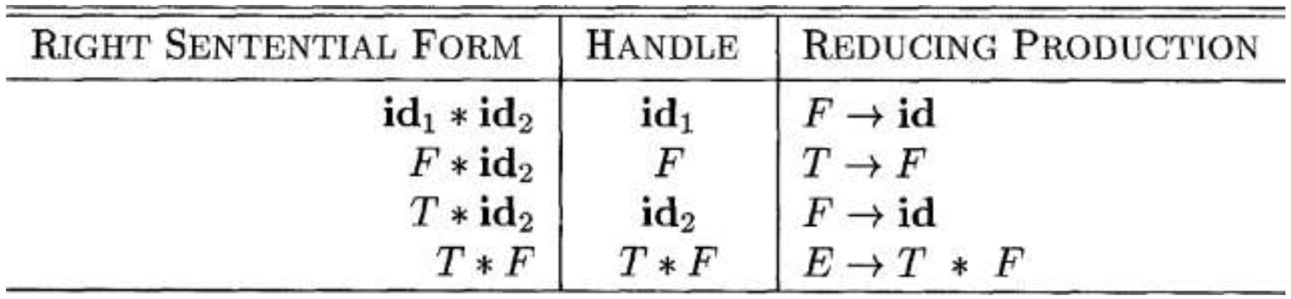


- For sentential form $T * \mathbf{id}$, there are two applicable productions:
  - T is the body of $E \rightarrow T$
  - **id** is the body of $F \rightarrow \mathbf{id}$
- Why reducing **id** to F, instead of reducing T to E?
  - **Reason:** If reducing T to E, $E * \mathbf{id}$ is no longer a sentential form
  - **Challenge:** how do we know this before making choices?

Handles (句柄)

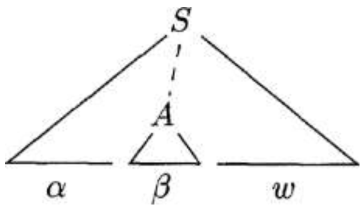


- Informally, a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation
- Formally, if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then production $A \rightarrow \beta$ (or simply β) in the position following α is a handle of the sentential form $\alpha\beta\omega$



RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Handle Pruning (句柄剪枝)



- In a right-sentential form (最右句型), the string to the right of the handle must contain only terminal symbols
- If a grammar is unambiguous, every right-sentential form of the grammar has exactly one handle*
- A rightmost derivation in reverse (bottom-up parsing) can be obtained by **"handle pruning"**

In every step:

- Locate the handle
- Replace it by production head

$$\begin{tikzcd}
S \arrow[r, phantom, "="] & \gamma_0 \arrow[l, bend right] \arrow[r, Rightarrow, "rm"'] & \gamma_1 \arrow[l, bend right] \arrow[r, Rightarrow, "rm"'] & \gamma_2 \arrow[l, bend right] \arrow[r, Rightarrow, "rm"'] & \cdots \arrow[l, bend right] \arrow[r, Rightarrow, "rm"'] & \gamma_{n-1} \arrow[l, bend right] \arrow[r, Rightarrow, "rm"'] & \gamma_n = w \arrow[l, bend right]
\end{tikzcd}$$

* Given an unambiguous grammar, a right-sentential form can only be derived in one way

Shift-Reduce Parsing

- 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

Actions:
Shift
Reduce
Accept
Error

Initial status:

STACK	INPUT
\$	ω\$

Shift-reduce process:

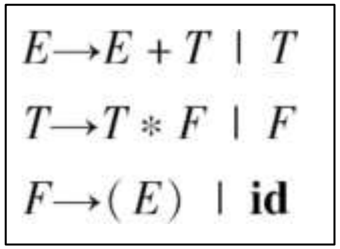
- The parser **shifts** zero or more input symbols onto the stack, until it is ready to reduce a string β (a handle) on top of the stack*
- **Reduce** β to the head of the appropriate production

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

* During shift-reduce parsing, the handle will always eventually appear on top of the stack

Example

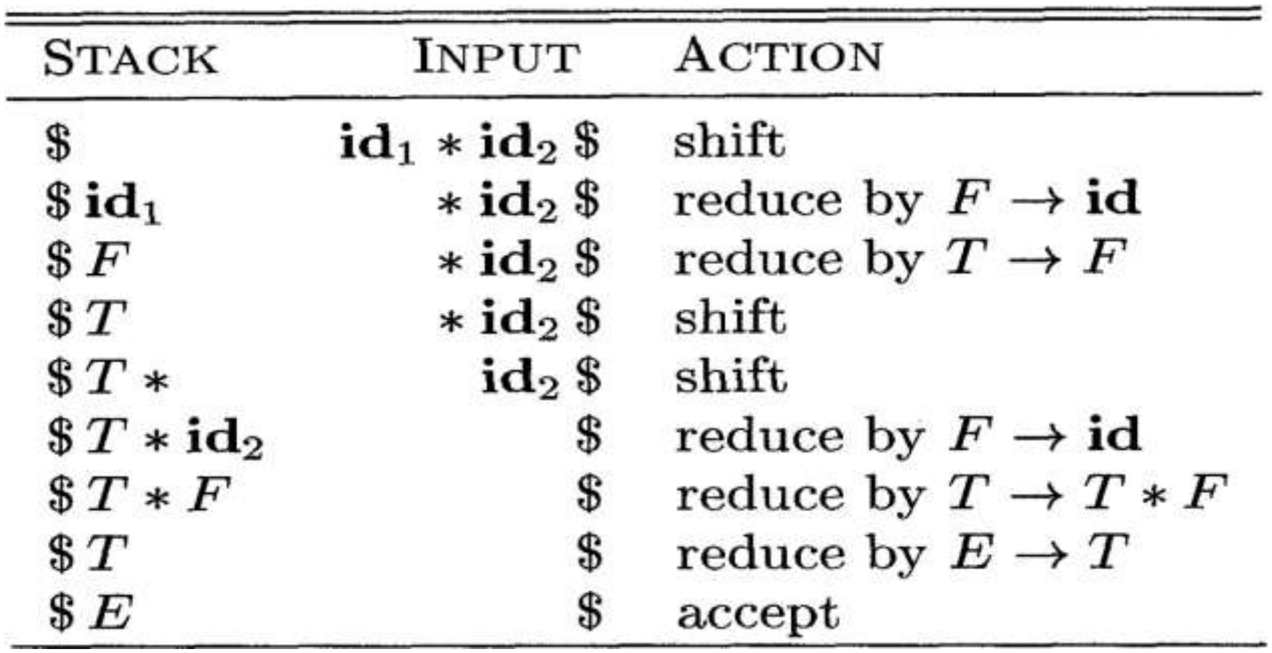
Parsing steps on input $\mathbf{id}_1 * \mathbf{id}_2$



$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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



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

Why Handles Always Appear at Stack Top?

(Analysis by enumeration) Consider the possible forms of two successive steps in any rightmost derivation[*]

$$(1)\quad S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A z \underset{rm}{\Rightarrow} \alpha\beta B y z \underset{rm}{\Rightarrow} \alpha\beta\gamma y z$$

- **Step 1:** the rightmost nonterminal A is replaced by a string with nonterminals. Suppose B is the rightmost nonterminal in the string.
- **Step 2:** B is replaced by a string γ

$$(2)\quad S \underset{rm}{\overset{*}{\Rightarrow}} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x y z \underset{rm}{\Rightarrow} \alpha\gamma x y z$$

- **Step 1:** the rightmost nonterminal A is replaced by a string y with terminals only.
- **Step 2:** the next rightmost nonterminal B (somewhere to the left of y) is replaced by a string γ

[*] The analysis is trivial when there is only one step of derivation

Why Handles Always Appear at Stack Top?

(1) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A z \underset{rm}{\Rightarrow} \alpha\beta B y z \underset{rm}{\Rightarrow} \alpha\beta\gamma y z$

(2) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x y z \underset{rm}{\Rightarrow} \alpha\gamma x y z$

STACK	INPUT	ACTION
…	…	…
$\$\alpha\beta\gamma$	$yz\$$	Reduce
$\$\alpha\beta B$	$yz\$$	Shift
$\$\alpha\beta By$	$z\$$	Reduce
$\$\alpha A$	$z\$$	Shift
…	…	…

STACK	INPUT	ACTION
…	…	…
$\$\alpha\gamma$	$xyz\$$	Reduce
$\$\alpha B$	$xyz\$$	Shift
$\$\alpha Bx$	$yz\$$	Shift
$\$\alpha Bxy$	$z\$$	Reduce
$\$\alpha BxA$	$z\$$	Shift
…	…	…

x, y, z are strings of terminals

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*	$\rightarrow$	**id** (*parameter_list*)
(2)	*stmt*	$\rightarrow$	*expr* := *expr*
(3)	*parameter_list*	$\rightarrow$	*parameter_list* , *parameter*
(4)	*parameter_list*	$\rightarrow$	*parameter*
(5)	$\boxed{\textit{parameter} \rightarrow \textbf{id}}$		
(6)	*expr*	$\rightarrow$	**id** (*expr_list*)
(7)	$\boxed{\textit{expr} \rightarrow \textbf{id}}$		
(8)	*expr_list*	$\rightarrow$	*expr_list* , *expr*
(9)	*expr_list*	$\rightarrow$	*expr*

Reduce by which production?

Outline

- Introduction
- 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
- Parser Generators (to be discussed in lab sessions)

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

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

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

How does a shift/reduce parser know that *T* on stack top is not a handle (so the right action is to shift, not reducing *T* to *E*)?

LR(0) Items (LR(0) 项)

- An LR parser makes shift-reduce decisions by **maintaining states** to keep track of what have been seen in a parse
- 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 point in the parsing process
  - $A \to \cdot XYZ \qquad A \to X \cdot YZ \qquad A \to XY \cdot Z \qquad A \to XYZ \cdot$
  - $A \to X \cdot YZ$: we have just seen on the input a string derivable from X and we hope to see a string derivable from YZ next
- **States:** sets of LR(0) items (LR(0) 项集)

The production $A \to \epsilon$ generates only one item $A \to \cdot$

Canonical LR(0) Collection

- One collection of sets of LR(0) items, called the ***canonical* LR(0) collection** (规范LR（0）项集族), provides the basis for constructing a DFA that is used 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)
    - (2) GOTO

Augmented Grammar

- Augmenting a grammar G with start symbol S
  - Introduce a new start symbol S' to replace 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 $\text{CLOSURE}(I)$ is the set of items constructed from I by the two rules

  1. Initially, add every item in I to $\text{CLOSURE}(I)$
  2. If $A \to \alpha \cdot B\beta$ is in $\text{CLOSURE}(I)$ and $B \to \gamma$ is a production, then add the item $B \to \cdot \gamma$ to $\text{CLOSURE}(I)$, if it is not already there. Apply this rule until no more new items can be added to $\text{CLOSURE}(I)$

- **Intuition:** $A \to \alpha \cdot B\beta$ indicates that we hope to see a substring derivable from $B\beta$. This substring will have 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 $A \to \alpha \cdot B\beta$ in $J$ )
                        for ( each production $B \to \gamma$ of $G$ )
                                if ( $B \to \cdot\gamma$ is not in $J$ )
                                        add $B \to \cdot\gamma$ 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
  - $E' \rightarrow E$ $\quad E \rightarrow E + T \mid T$ $\quad T \rightarrow T * F \mid F$ $\quad F \rightarrow (E) \mid \mathbf{id}$

- 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 $[F \rightarrow \cdot\, \mathbf{id}]$ to the closure (reached fixed point)

The Function GOTO

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

- **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 $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X\beta]$ 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 $\cdot$
  - Then compute the CLOSURE$(\{[E \rightarrow E + \cdot\ T]\})$, which contains
    - $[E \rightarrow E + \cdot T\ ]$
    - $[T \rightarrow \cdot T * F\ ]$, $[T \rightarrow \cdot\ F\ ]$
    - $[F \rightarrow \cdot(E)\ ]$, $[F \rightarrow \cdot\ \mathbf{id}\ ]$

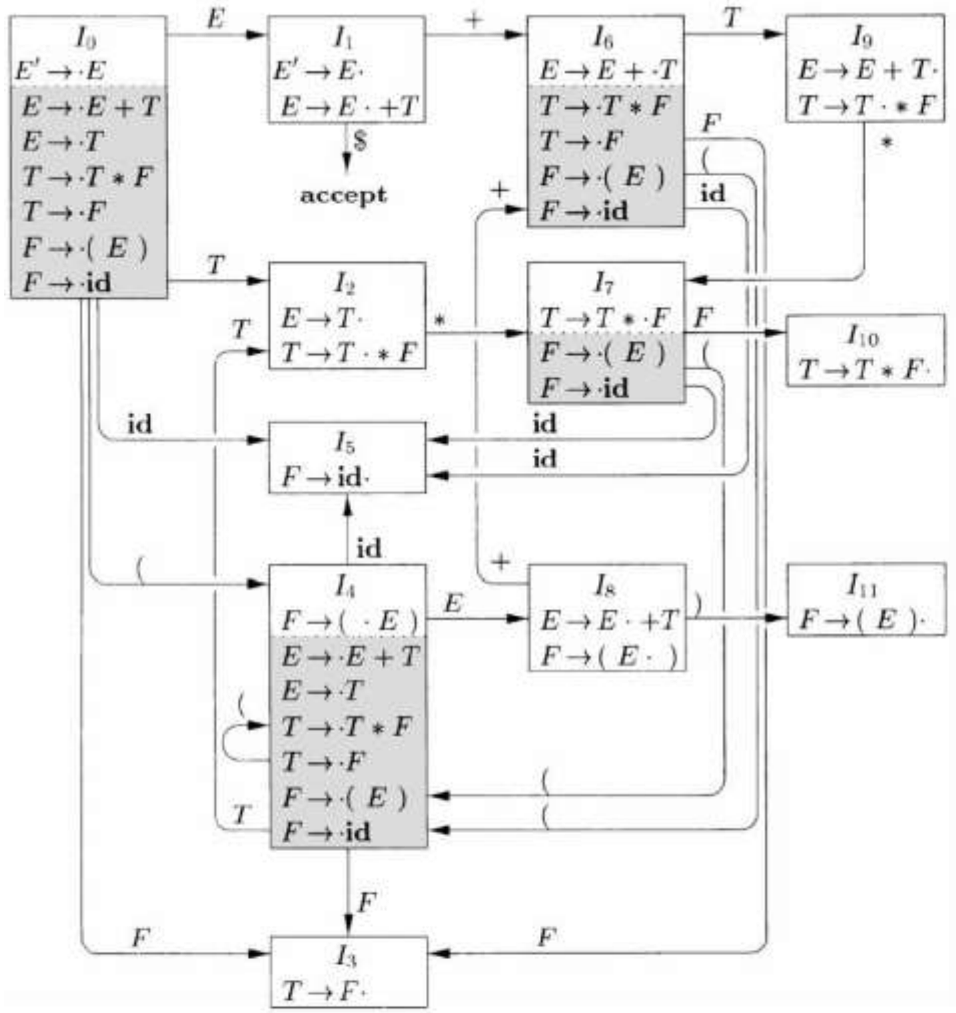
Constructing Canonical LR(0) Collection

```
void items(G') {
        C = {CLOSURE({[S' → ·S]})};        ⟶ Initial item set
        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;
}
```

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

Example

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



(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

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

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 CLOSURE($\{S' \rightarrow \cdot\, S\}$)

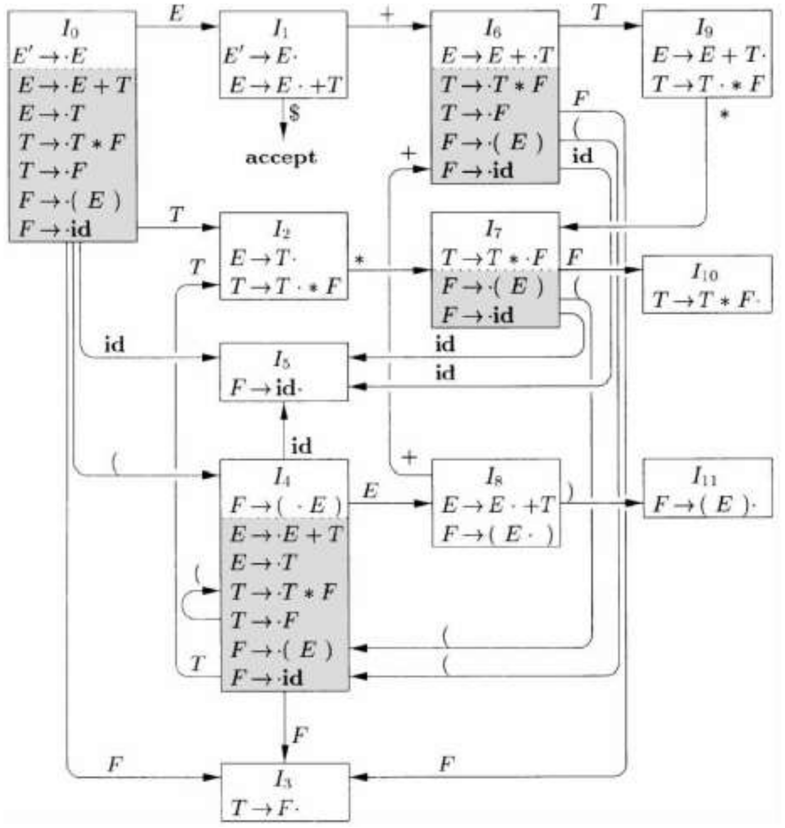
The Use of LR(0) Automaton

Helps make shift-reduce decisions:

- Suppose that the string γ of grammar symbols takes the automaton from the start state 0 to some state j

- **Shift** on next input symbol a if state j has a transition on a

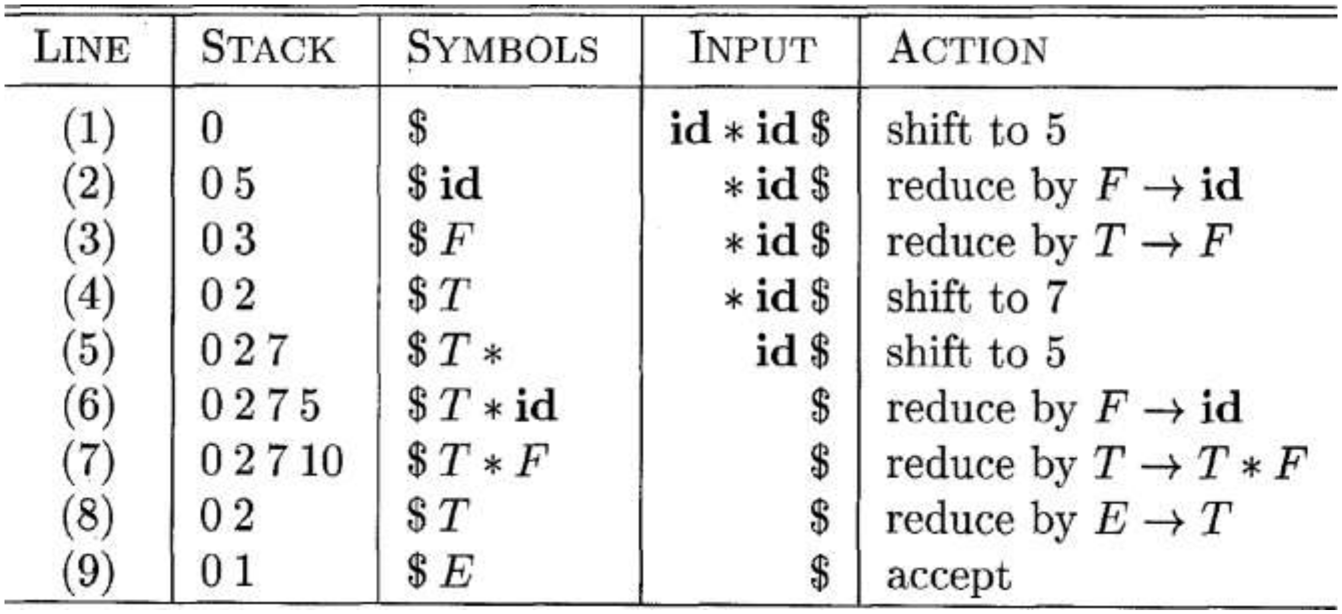
- Otherwise, **reduce**; the items in state j will tell us which production to use



I_0: $E' \to \cdot E$, $E \to \cdot E + T$, $E \to \cdot T$, $T \to \cdot T * F$, $T \to \cdot F$, $F \to \cdot (E)$, $F \to \cdot \text{id}$
I_1: $E' \to E \cdot$, $E \to E \cdot + T$
accept
I_2: $E \to T \cdot$, $T \to T \cdot * F$
I_3: $T \to F \cdot$
I_4: $F \to (\cdot E)$, $E \to \cdot E + T$, $E \to \cdot T$, $T \to \cdot T * F$, $T \to \cdot F$, $F \to \cdot (E)$, $F \to \cdot \text{id}$
I_5: $F \to \text{id} \cdot$
I_6: $E \to E + \cdot T$, $T \to \cdot T * F$, $T \to \cdot F$, $F \to \cdot (E)$, $F \to \cdot \text{id}$
I_7: $T \to T * \cdot F$, $F \to \cdot (E)$, $F \to \cdot \text{id}$
I_8: $E \to E \cdot + T$, $F \to (E \cdot)$
I_9: $E \to E + T \cdot$, $T \to T \cdot * F$
I_{10}: $T \to T * F \cdot$
I_{11}: $F \to (E) \cdot$

Example: Parsing id $*$ id (1)

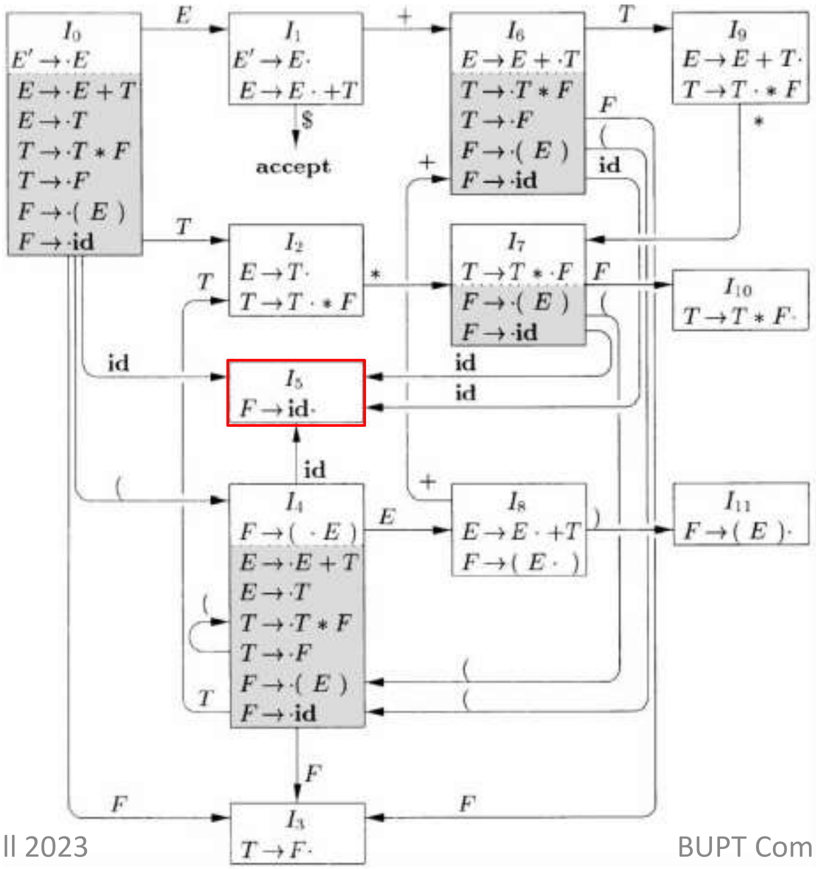
- We only keep states in the stack; grammar symbols can be recovered from the states



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

Example: Parsing id $*$ id (2)

LINE	STACK	SYMBOLS	INPUT	ACTION
(2)	0 5	\$ id	$*$ id \$	reduce by $F \rightarrow$ id
(3)	0 3	\$ F	$*$ id \$	reduce by $T \rightarrow F$



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

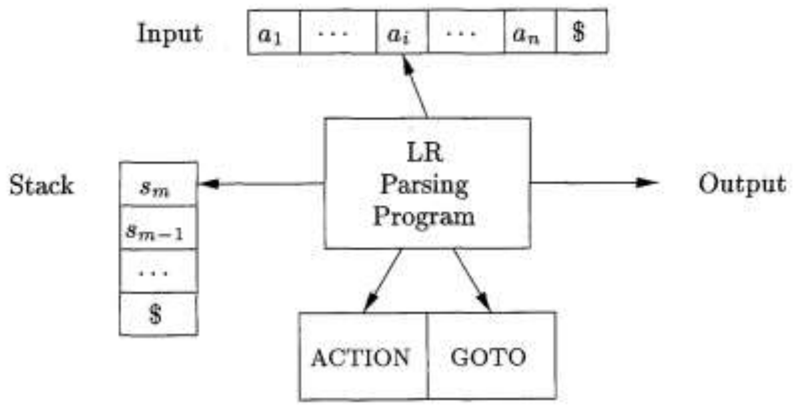
Line 2: State 5 has no transition on input symbol $*$, so reduce

- Pop state 5
- Since **id** will be replaced by F, state 5 should be replaced by state 3; pop state 5 and push state 3 to stack*

*State 0 transits to 3 on F

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
- 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 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 forms of values:
  - **Shift j:** shift input a to the stack, but uses state j to represent a
  - **Reduce $A \rightarrow \beta$:** reduce β on the top of the stack to head A
  - **Accept:** The parser accepts the input and finishes parsing
  - **Error:** syntax errors exist
- The **GOTO** function is extended from the one defined on sets of items to states: if GOTO(I_i, A) = I_j, then GOTO(i, A) = j

LR Parser Configurations (态势)

- "**Configuration**" is notation for representing the complete state of the parser. A *configuration* is a pair:

Stack contents (top on the right) $$(\underline{s_0 s_1 \ldots s_m}, \underline{a_i a_{i+1} \ldots a_n \$})$$ Remaining input

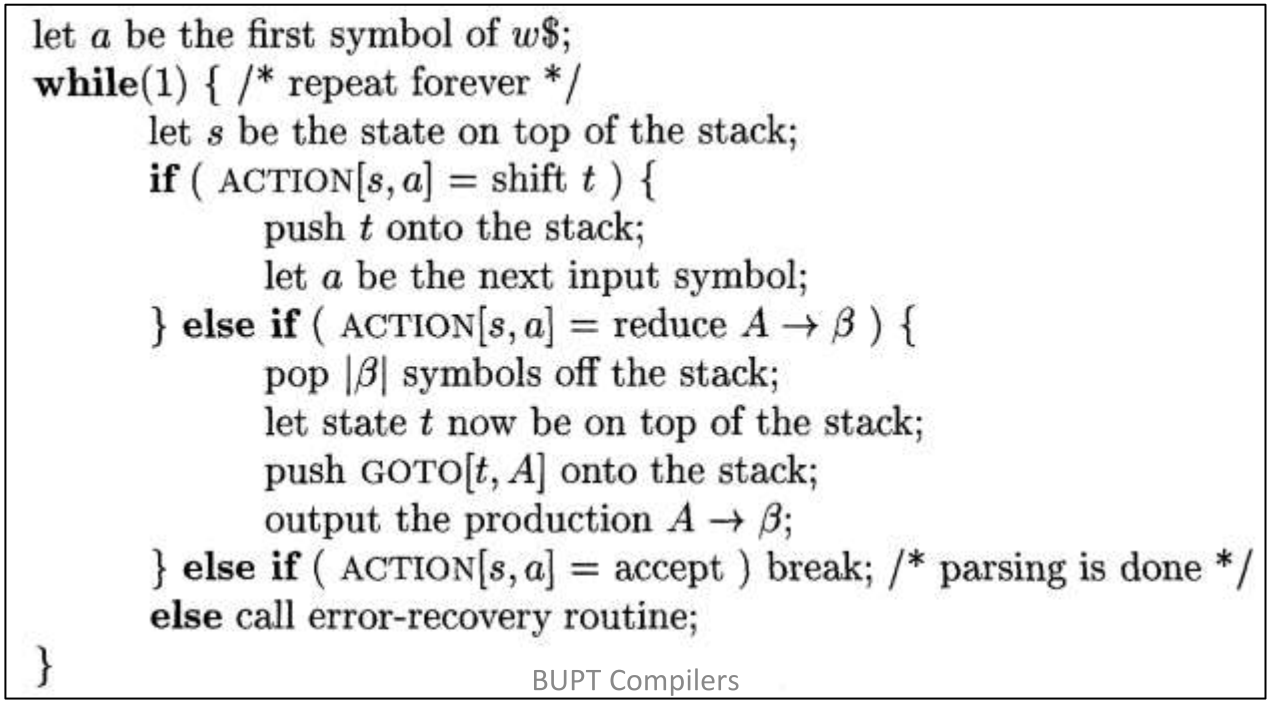
- By construction, each state (except s_0) in an LR parser corresponds to a set of items and a grammar symbol (the symbol that leads to the state transition, i.e., the symbol on the incoming edges)
  - Suppose X_i is the grammar symbol for state s_i
  - Then $X_0 X_1 \ldots X_m a_i a_{i+1} \ldots a_n$ 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 $(s_0 s_1 \dots s_{m\text{-}r} s,\ a_i a_{i+1} \dots a_n \$)$, where r = the length of β, and $s =$ GOTO$(s_{m\text{-}r}, A)$
    - accept: parsing is completed
    - error: the parser has found an error and calls an error recovery routine

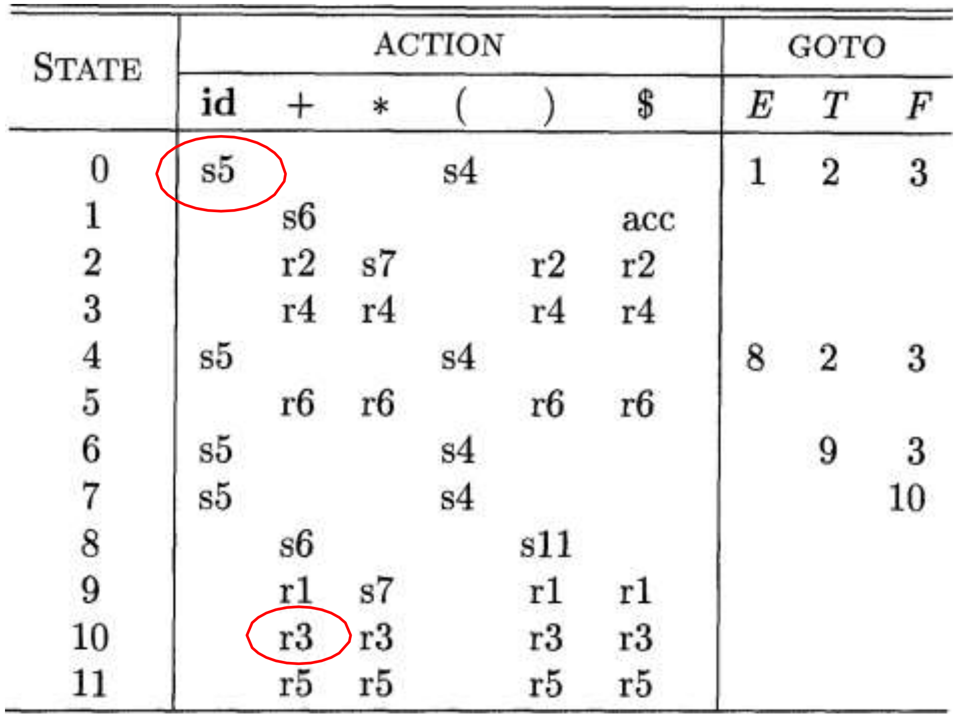
LR-Parsing Algorithm

- **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 $A \to \beta$ ) {
                pop $|\beta|$ symbols off the stack;
                let state $t$ now be on top of the stack;
                push GOTO[$t, A$] onto the stack;
                output the production $A \to \beta$;
        } else if ( ACTION[$s, a$] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Parsing Table Example

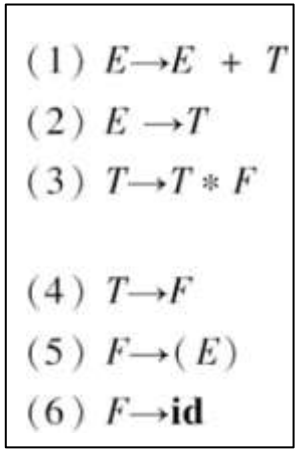


STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

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

- *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 Parsing Example



(1) $E \to E + T$

(2) $E \to T$

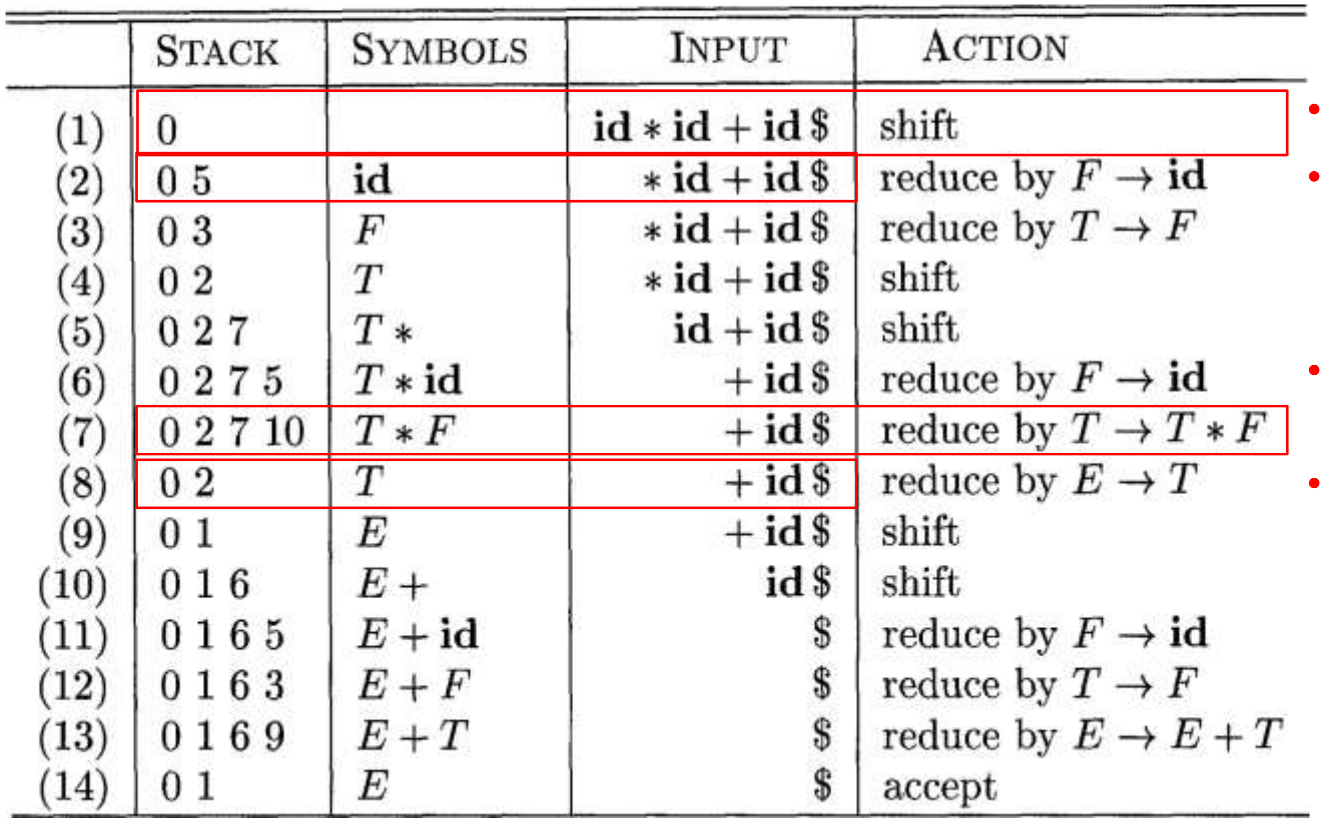
(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

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

- Input string: **id** $*$ **id** $+$ **id**



	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \to$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \to F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T *$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T *$ **id**	$+$ **id** \$	reduce by $F \to$ **id**
(7)	0 2 7 10	$T * F$	$+$ **id** \$	reduce by $T \to T * F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \to T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E +$	**id** \$	shift
(11)	0 1 6 5	$E +$ **id**	\$	reduce by $F \to$ **id**
(12)	0 1 6 3	$E + F$	\$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	\$	reduce by $E \to E + T$
(14)	0 1	E	\$	accept

- Push state 5
- Remove id

- Pop three states 2, 7, 10
- Push state 2 (GOTO$[0, T] = 2$)

Constructing SLR-Parsing Tables (1)

- 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 $\{I_0, I_1, \dots, I_n\}$ for the augmented grammar G'

2. State i is constructed from I_i. ACTION can be determined as follows:

   - If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}\ [I_i, a] = I_\text{j}$, then set $\text{ACTION}[i, a]$ to "shift j" (here a must be a terminal)
   - If $[A \rightarrow \alpha\cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \rightarrow \alpha$" for **all a in FOLLOW(A)**; here A may not be S'
   - If $[S' \rightarrow S\cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept"

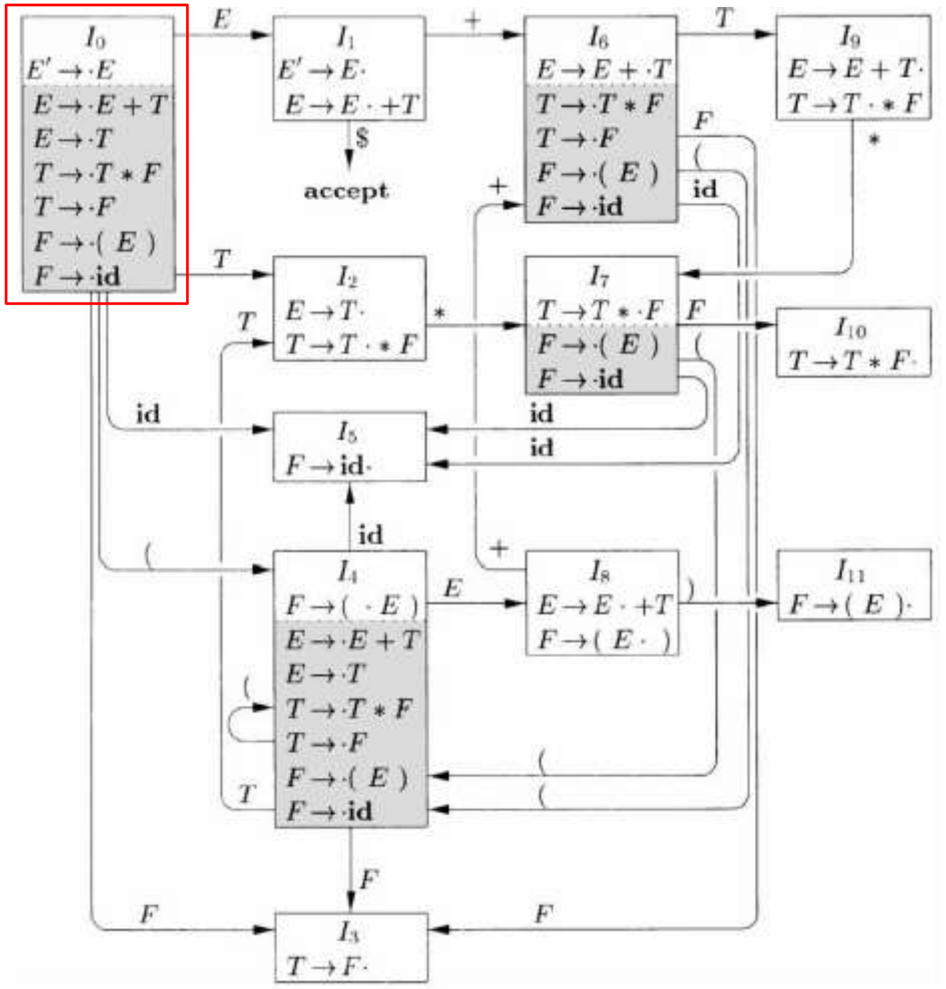
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_\text{i}, A) = I_j$, then $\text{GOTO}(i, A) = j$

Constructing SLR-Parsing Tables (2)

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 $[S' \rightarrow \cdot S]$

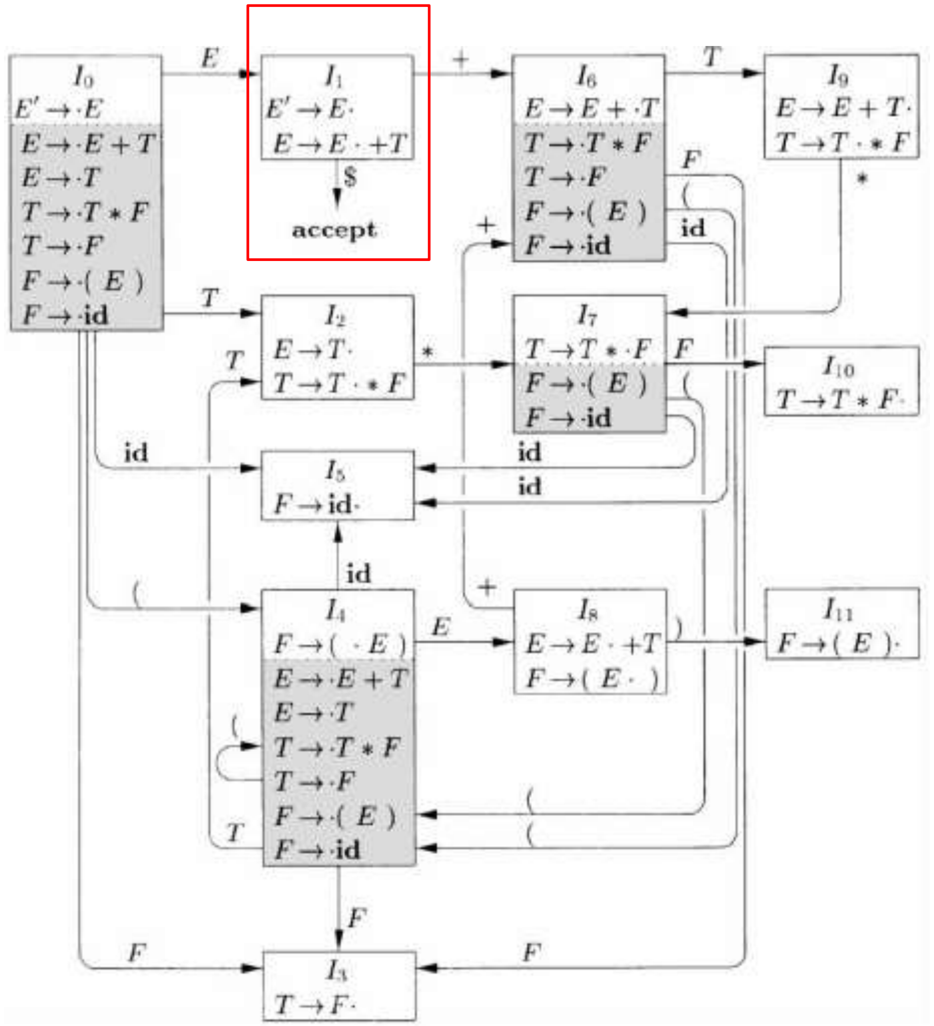
If there is no conflict during the parsing table construction (i.e., multiple actions for a table entry), the grammar is **SLR(1)**

Example (1)



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

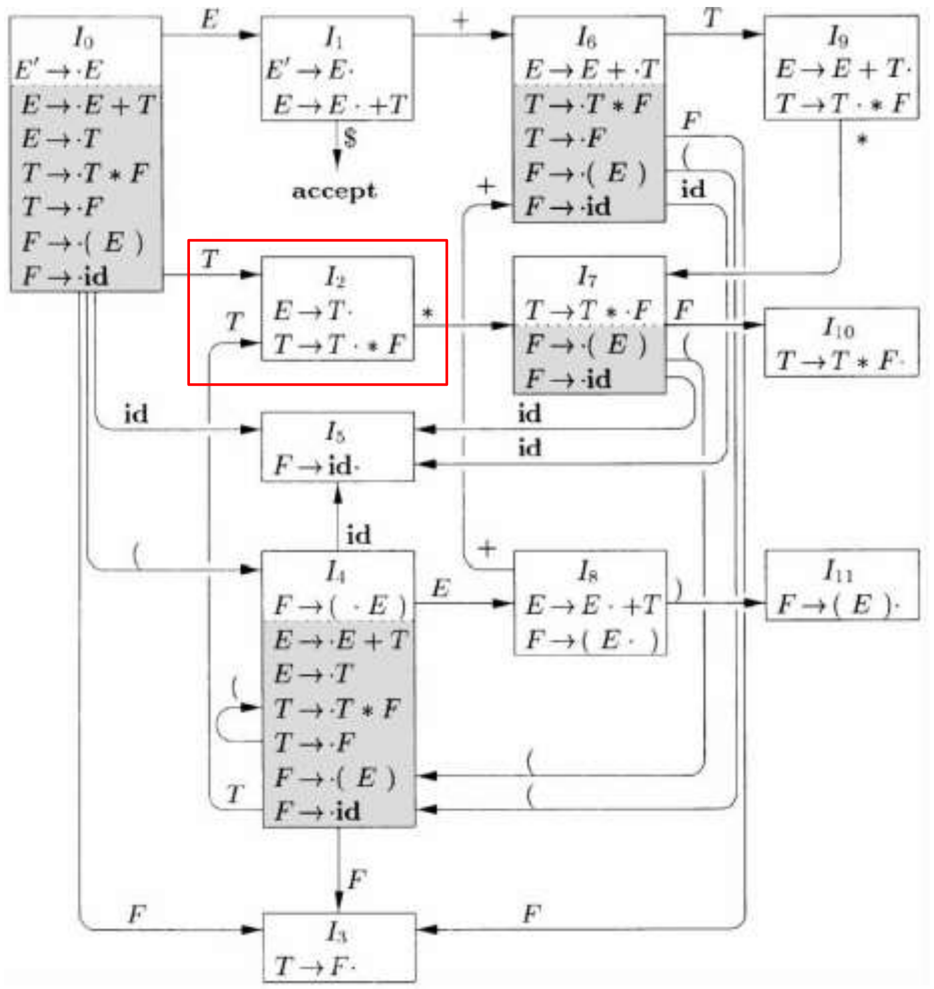
Example (2)



- $\text{ACTION}(1, +) = s6$

- $\text{ACTION}(1, \$) = \text{accept}$

Example (3)

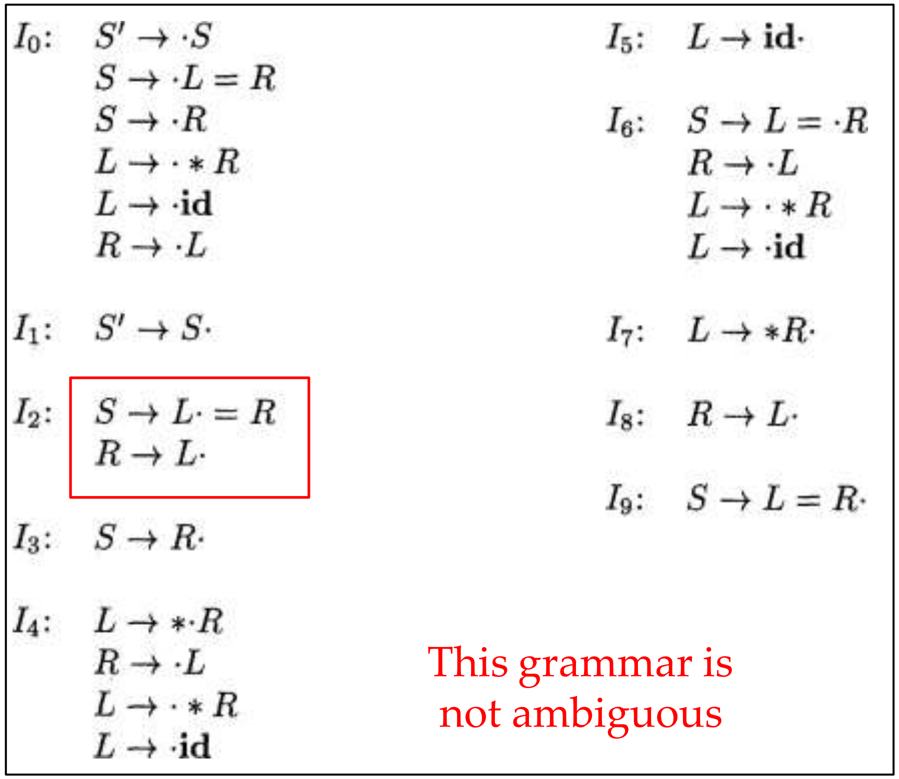


- ACTION(2,∗) = s7
- ACTION(2, \$) = reduce $E \to T$
- ACTION(2, +) = reduce $E \to T$
- ACTION(2,)) = reduce $E \to T$

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: $\boxed{\begin{array}{l} S \rightarrow L \cdot = R \\ R \rightarrow L \cdot \end{array}}$

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.

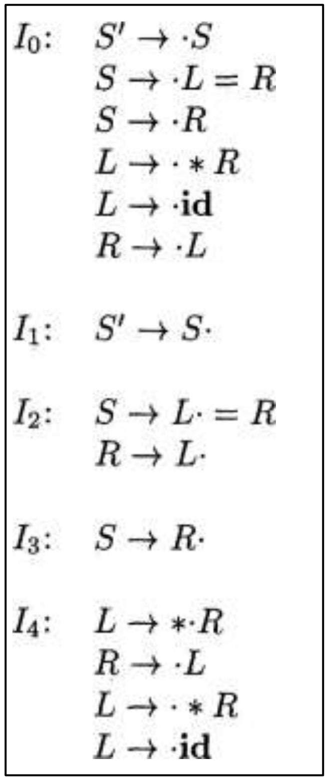
Weakness of the SLR Method

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

- 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

Example: Parsing id = id

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



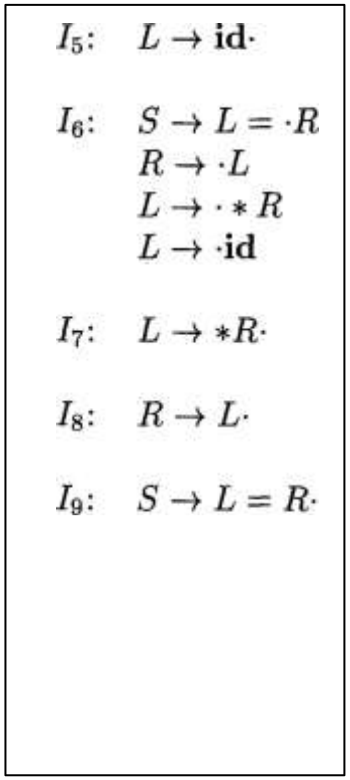
I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

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 L→id
$02	L	= id	Suppose reduce 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.

Outline

- Introduction
- 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
- Parser Generators (to be discussed in lab sessions)

LR(1) Item

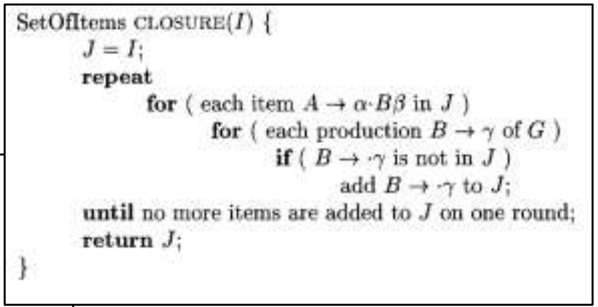
- **Idea:** Carry more information in the state to rule out some invalid reductions (splitting LR(0) states)

- 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 $\mathrm{FOLLOW}(A)$)

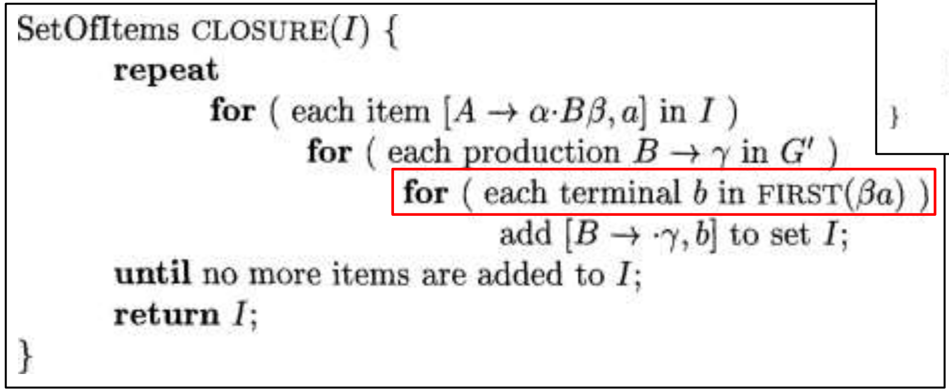
*: LR(0) items do not have lookahead symbols, and hence they are called LR(0)

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) {
    J = I;
    repeat
        for ( each item $A \to \alpha \cdot B\beta$ in $J$ )
            for ( each production $B \to \gamma$ of $G$ )
                if ( $B \to \cdot\gamma$ is not in $J$ )
                    add $B \to \cdot\gamma$ to $J$;
    until no more items are added to $J$ on one round;
    return $J$;
}
```



```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item $[A \to \alpha \cdot B\beta, a]$ in $I$ )
            for ( each production $B \to \gamma$ in $G'$ )
                for ( each terminal $b$ in FIRST($\beta a$) )
                    add $[B \to \cdot\gamma, b]$ to set $I$;
    until no more items are added to $I$;
    return $I$;
}
```

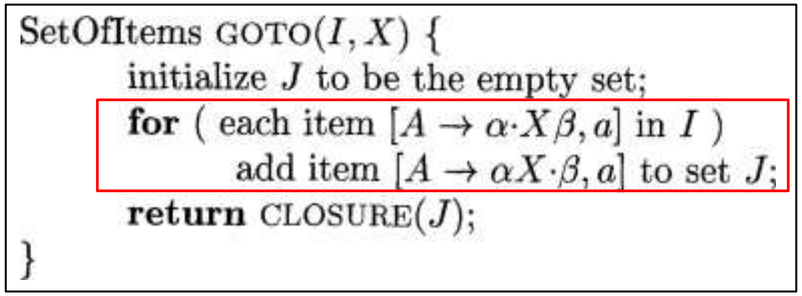
It only generates the new item $[B \to \cdot\gamma, b]$ from $[A \to \alpha \cdot B\beta, a]$ if b is in FIRST(βa)

Why b should be in FIRST(βa)?

- **An informal analysis:**
  - The lookahead b decides when to reduce γ to B
  - Suppose the algorithm also generates an item for b that is NOT in FIRST(βa)
  - At a certain point, when we see γ on stack top and b as next input, we would reduce γ to B ($[B \to \gamma \cdot, b]$ must come from $[B \to \cdot\, \gamma, b]$)
  - Since we generate $[B \to \cdot\, \gamma, b]$ from $[A\to\alpha{\cdot}B\beta, a]$ during closure computation, we hope to see α$B\beta$ at stack top sometime after reducing γ to B and reduce using $A \to \alpha B\beta$ when a is the next input symbol
  - Unfortunately, this is impossible when b is not in FIRST(βa)
  - Then why generate $[B \to \cdot\, \gamma, b]$from $[A\to\alpha{\cdot}B\beta, a]$???

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.



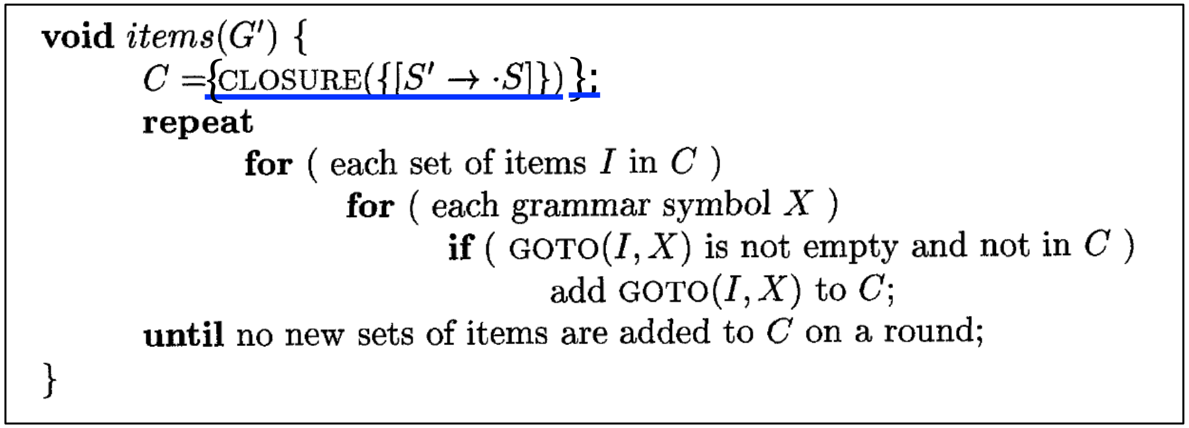
```
SetOfItems GOTO($I, X$) {
        initialize $J$ to be the empty set;
        for ( each item $[A \to \alpha \cdot X\beta, a]$ in $I$ )
                add item $[A \to \alpha X \cdot \beta, a]$ to set $J$;
        return CLOSURE($J$);
}
```

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

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

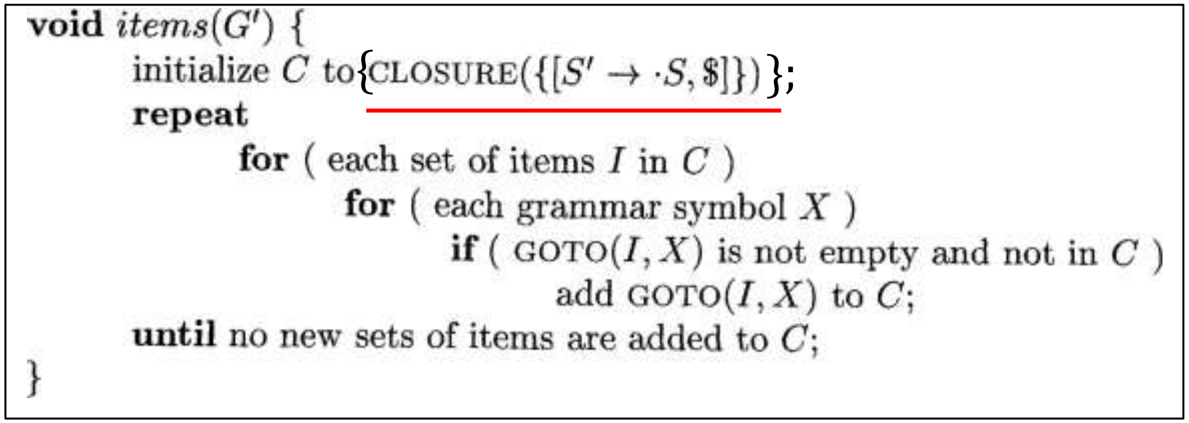
The lookahead symbols are passed to new items from existing items

Constructing LR(1) Item Sets (3)



```
void $items(G')$ {
    $C = \underline{\{\text{CLOSURE}(\{[S' \to \cdot S]\})\}}$;
    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;
}
```

Constructing the collection of LR(0) item sets



```
void $items(G')$ {
    initialize $C$ to $\underline{\{\text{CLOSURE}(\{[S' \to \cdot S, \$]\})\}}$;
    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(1) item sets

LR(1) Item Sets Example

- Augmented grammar:
  - $S' \to S \qquad S \to CC \qquad C \to cC \mid d$
- Constructing I_1 item set and GOTO function:
  - $I_0 = \text{CLOSURE}([S' \to \cdot S, \$\ ]) =$
    - $\{[S' \to \cdot S, \$], [S \to \cdot CC, \$], [C \to \cdot cC, c/d], [C \to \cdot d, c/d]\}$

FIRST($) = {$}

FIRST(C$) = {c, d}

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

FIRST($) = {$}

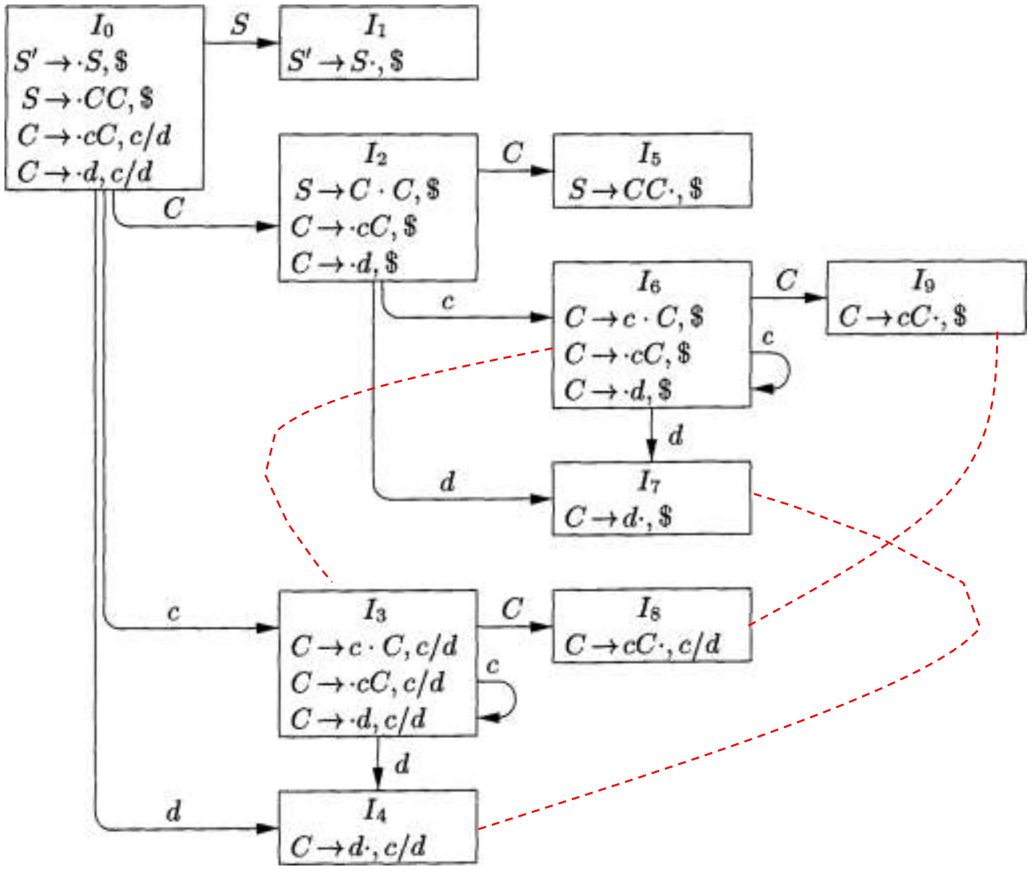
  - $\text{GOTO}(I_0, c) = \text{CLOSURE}(\{[C \to c \cdot C, c/d\ ]\}) =$
    - $\{[C \to c \cdot C, c/d\ ], [C \to \cdot cC, c/d], [C \to \cdot d, c/d]\}$
  - $\text{GOTO}(I_0, d) = \text{CLOSURE}(\{[C \to d \cdot, c/d\ ]\}) = \{[C \to d \cdot, c/d\ ]\}$

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)

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 $\mathrm{GOTO}(I_\mathrm{i}, a) = I_\mathrm{j}$, then set $\mathrm{ACTION}[i, a]$ to "*shift j*." Here, a must be a terminal.

   - If $[\ A \to \alpha\cdot, a]$ is in I_i, $A \neq S'$, then set $\mathrm{ACTION}[i, a]$ to "*reduce* $A \to \alpha$"

   - If $[\ S' \to S\cdot, \$]$ is in I_i, then set $\mathrm{ACTION}[i, \$]$ to "*accept*"

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

Constructing Canonical LR(1) Parsing Tables (2)

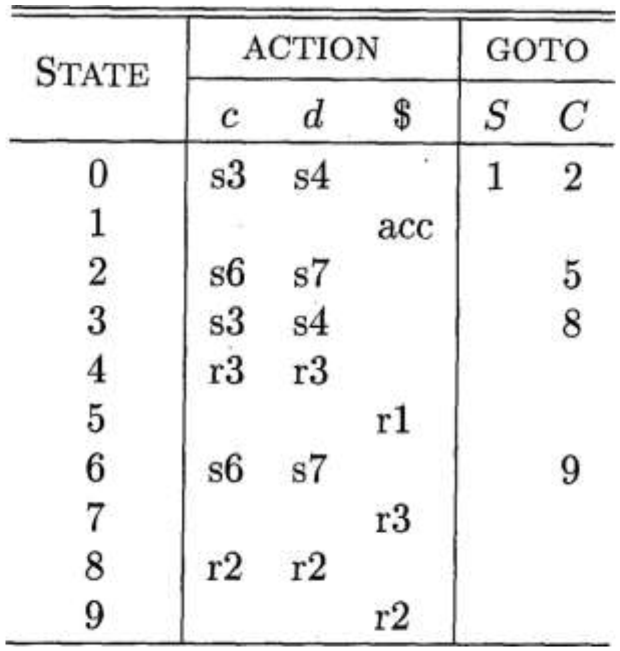
3. The goto transitions for state i are constructed from all nonterminals A using the rule: If $\mathrm{GOTO}(I_i, A) = I_j$, then $\mathrm{GOTO}(i, A) = j$

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:**
  - $S' \rightarrow S$
  - $S \rightarrow CC$
  - $C \rightarrow cC \mid d$
- These pairs of states can be seen as being split from the corresponding LR(0) states:
  - (3, 6)
  - (4, 7)
  - (8, 9)



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		

Outline

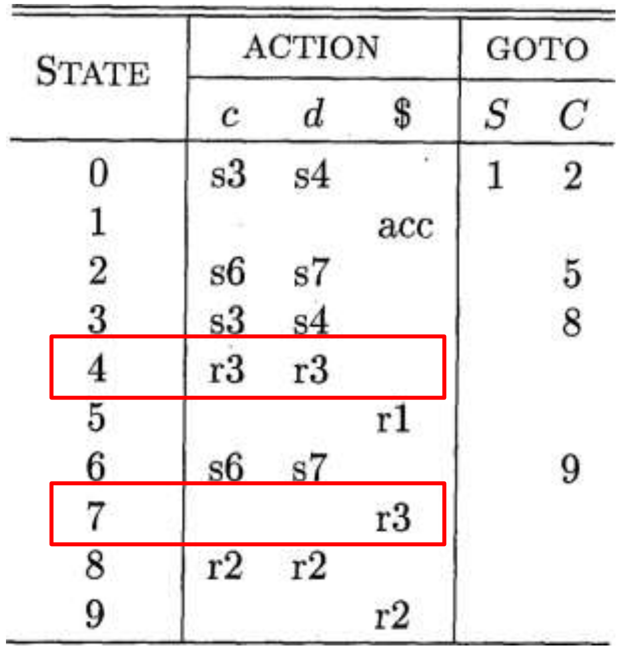
- Introduction
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
    - Simple LR (SLR)
    - Canonical LR (CLR)
    - Look-ahead LR (LALR)
    - Error Recovery
- Parser Generators (to be discussed in lab sessions)

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

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



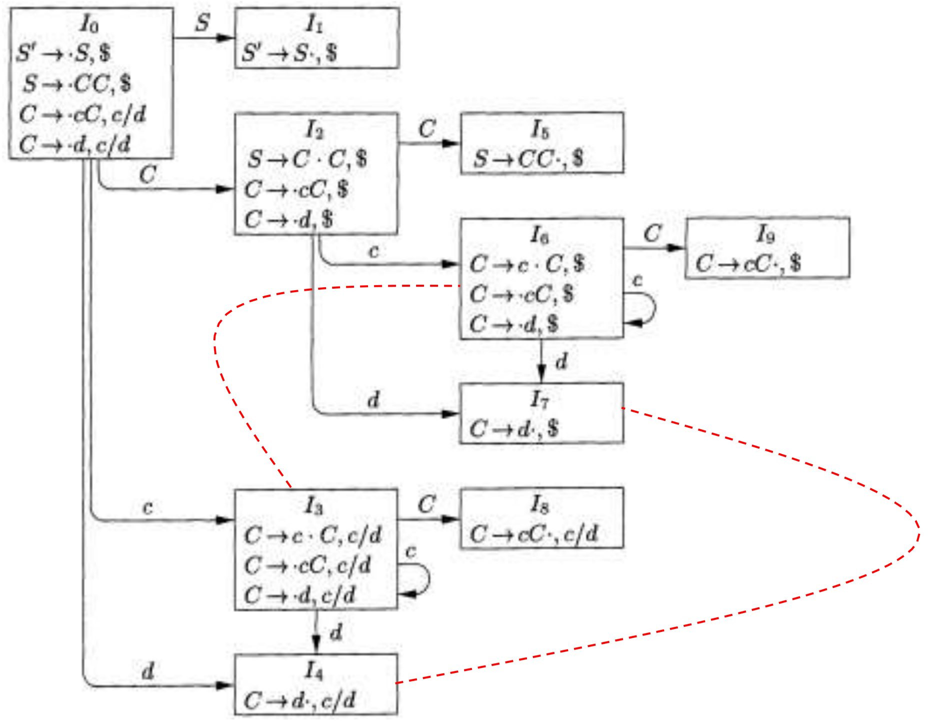
STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

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

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

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 $\{C \to d\,\cdot\}$
    - The core of I_3 and I_6 is $\{C \to c\cdot C, C \to \cdot cC, C \to \cdot d\}$



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

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 (for viable prefix ac)
  - $\{[A \rightarrow c\cdot, d], [B \rightarrow c\cdot, e]\}$
- Another set of valid LR(1) items (for viable prefix bc)
  - $\{[B \rightarrow c\cdot, d], [A \rightarrow c\cdot, e]$
- 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 (1)

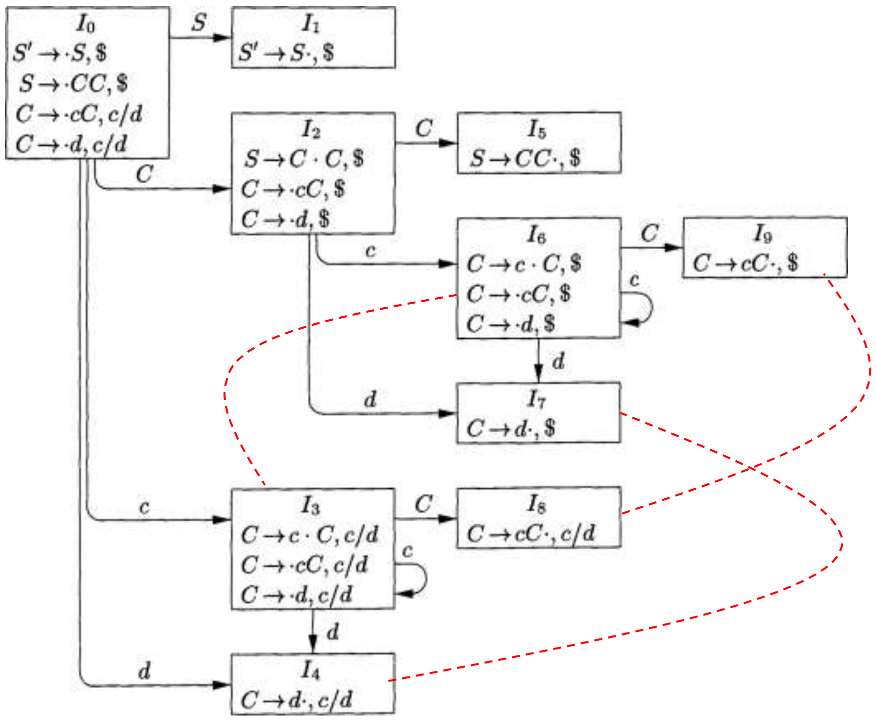
- Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items
- For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union
- Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting collection after merging.
  - The parsing 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)

Constructing LALR Parsing Table (2)

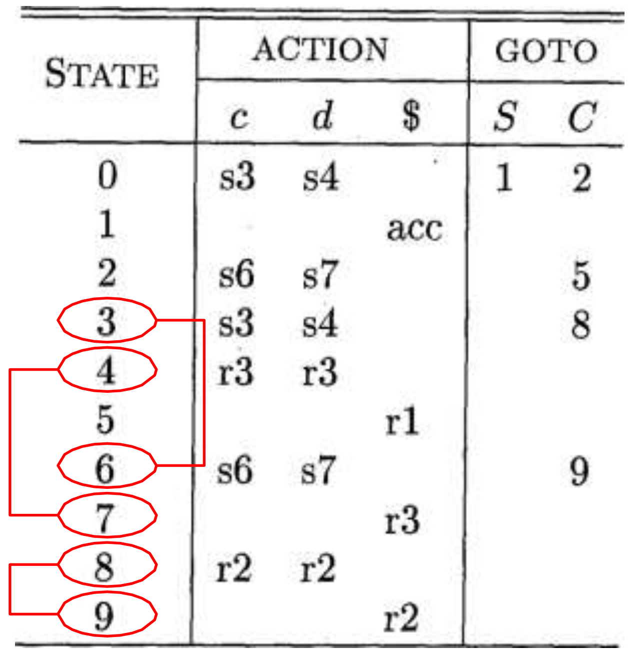
- Construct the GOTO table as follows:
  - If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of $\mathtt{GOTO}(I_1, X)$, $\mathtt{GOTO}(I_2, X)$, …, $\mathtt{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same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

LALR Parsing Table Example (1)

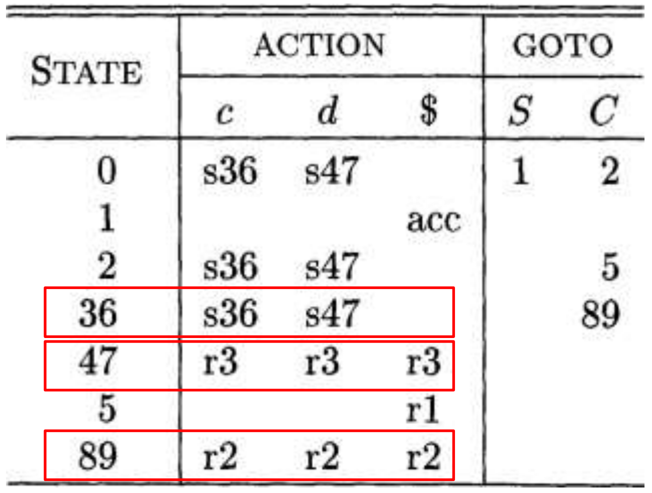
- Merging item sets
  - I_{36}: $[C \to c \cdot C, c/d/\$], [C \to \cdot\ cC, c/d/\$], [C \to \cdot\ d, c/d/\$]$
  - I_{47}: $[C \to d\ \cdot, c/d/\$]$
  - I_{89}: $[C \to cC\ \cdot, c/d/\$]$
- $\mathrm{GOTO}(I_{36}, C) = I_{89}$



LALR Parsing Table Example (2)



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
- 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
- Parser Generators (to be discussed in lab sessions)

Error Recovery in LR Parsing

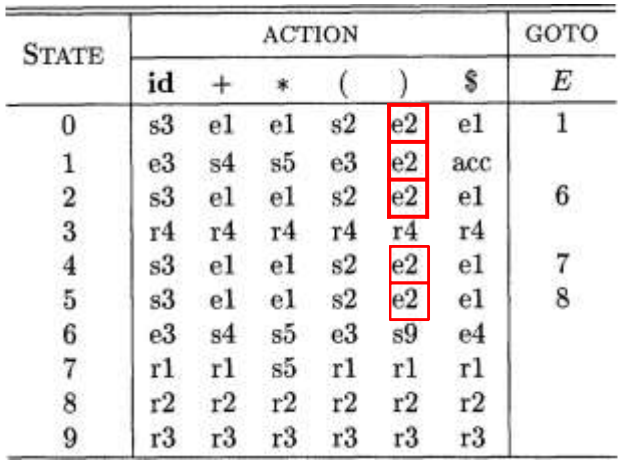
- An LR parser should be able to handle errors:
  - Report the precise location of an error
  - Recover from an error and continue with the parsing
- Two typical error recovery strategies
  - Panic-mode recovery (恐慌模式)
  - Phrase-level recovery (短语层次的恢复)

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:

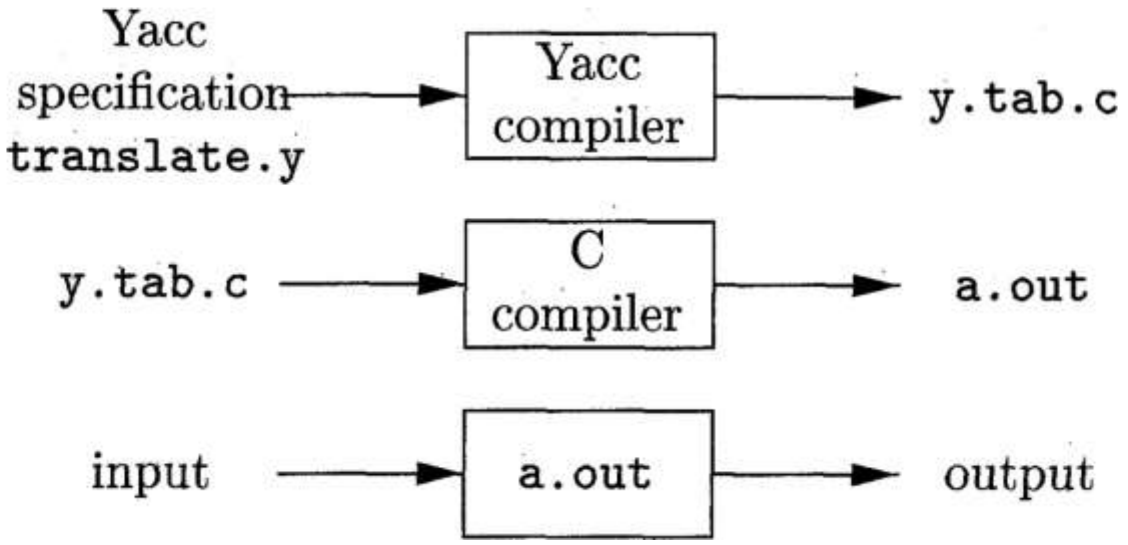
- Remove the right) from the input;
- Issue diagnostic "unbalanced right parenthesis".

Outline

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

The Parser Generator YACC/BISON

- Yacc： yet another compiler-compiler
- Bison： an extension and improvement of Yacc

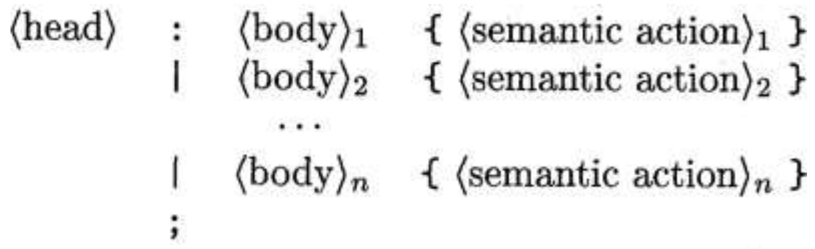


Structure of YACC Source Programs

```
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



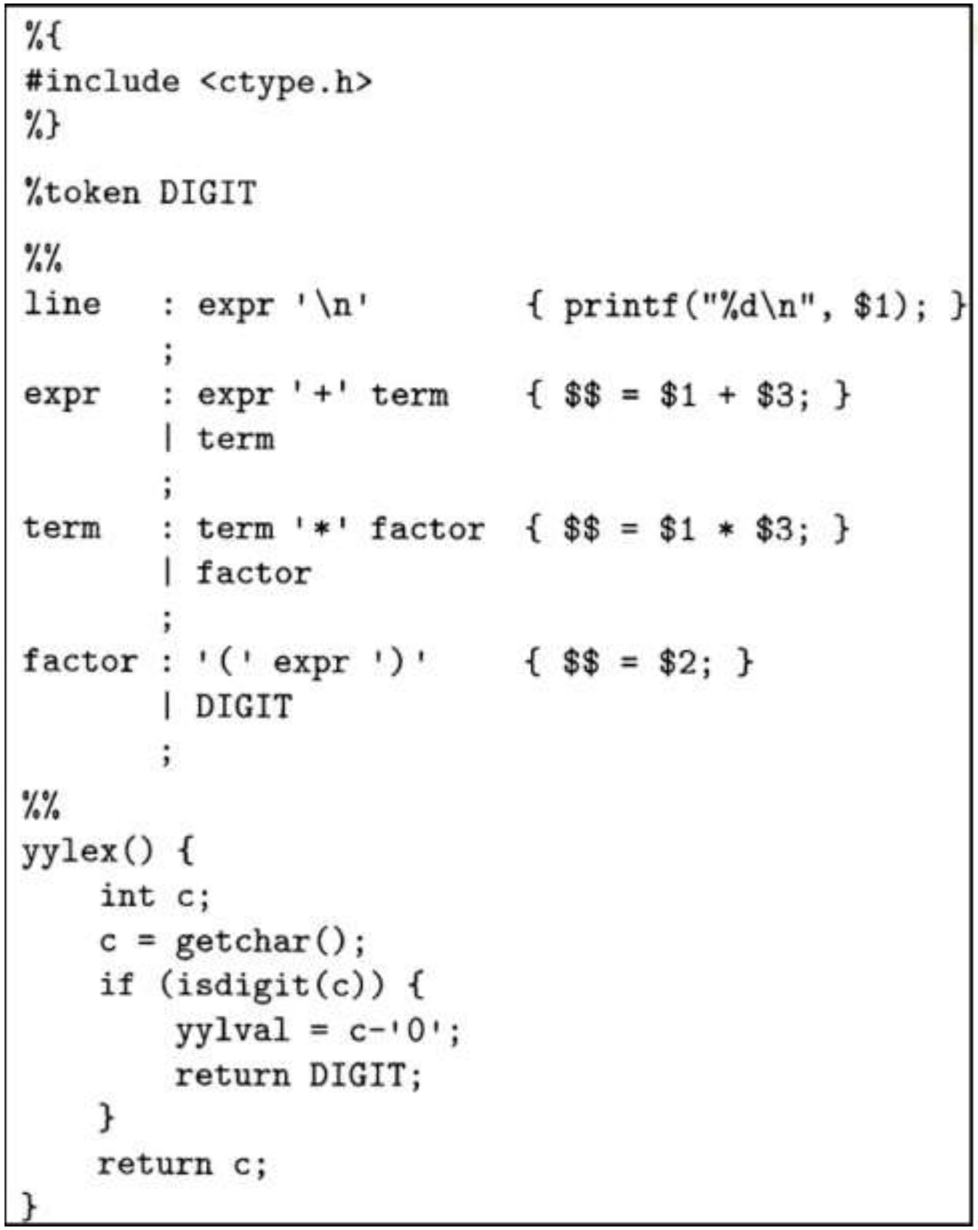
```
⟨head⟩   :   ⟨body⟩₁   { ⟨semantic action⟩₁ }
         |   ⟨body⟩₂   { ⟨semantic action⟩₂ }
               ...
         |   ⟨body⟩ₙ   { ⟨semantic action⟩ₙ }
         ;
```

- 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 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