Syntax Analysis

Md. Khorshed Alam
CSE, NDUB

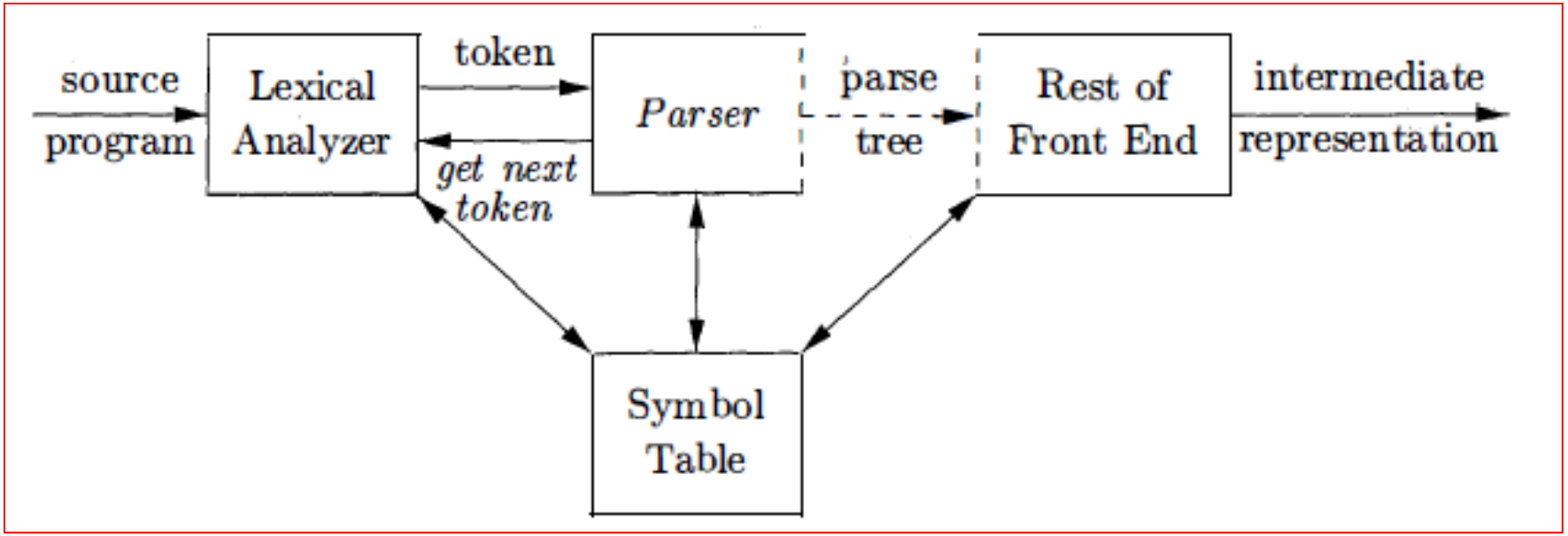
CFG offer significant benefits for both language designers & compiler writers

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
- From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program.
- ➢ As a side benefit, the parser-construction process can reveal syntactic ambiguities & trouble spots that might have slipped through the initial design phase of a language.
- The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code & for detecting errors.
- A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.
- ➢ These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

The Role of the Parser

- Parser obtains a string of tokens from the lexical analyzer & verifies that the string of token names can be generated by the grammar for the source language.
- Parser report any syntax errors in an intelligible fashion & to recover from commonly occurring errors to continue processing the remainder of the program.
- Conceptually, for well-formed programs, the parser constructs a parse tree & passes it to the rest of the compiler for further processing.

Position of parser in compiler model



- **Parsing Approach**
  - Top down
  - Bottom UP

Representative Grammars

- Expression grammar (LR): suitable for bottom-up parsing.

- Not suitable for top-down parsing (left recursive)

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

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

Remove left recursion

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

Producing ambiguity: a + b * c

Syntax Error Handling

- **Common programming errors**
- *Lexical errors* include misspellings of identifiers, keywords, or operators
  - the use of an identifier **elipseSize** instead of **ellipseSize** – and missing quotes around text intended as a string.
- *Syntactic errors* include misplaced semicolons or extra or missing braces; that is, "{" or "}."
  - As another example, in C or Java, the appearance of a case statement without an enclosing **switch** is a syntactic error
- *Semantic errors* include type mismatches between operators & operands.
  - An example is a **return** statement in a Java method with result type **void**.
- *Logical errors* can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.
  - The program containing = may be well formed; however, it may not reflect the programmer's intent.

Error-Recovery Strategies

- Panic-Mode Recovery
- Phrase-Level Recovery
- Error Productions
- Global Correction

- See details in the Book

Context-Free Grammars

- Grammars systematically describe the syntax of programming language constructs like expressions & statements.
- Using a syntactic variable *stmt* to denote statements & variable *expr* to denote expressions, the production

$$\text{stmt} \rightarrow \text{if (expr) stmt else stmt}$$

Formal Definition of CFG

1. **Terminals (T)** are the basic symbols from which strings are formed.

- The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal.
- Terminals are the first components of the tokens output by the lexical analyzer.
- Keywords: **if, else, (,)**

stmt → if (expr) stmt else stmt

stmt → if (expr) stmt else stmt

2. **Non-terminals (V)** are syntactic variables that denote sets of strings.

- Non-terminals: s**tmt, expr**
- The sets of strings denoted by non-terminals help define the language generated by the grammar.
- Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

stmt → if (expr) stmt else stmt

3. **Start symbol (S)**: one non-terminal is distinguished as the start symbol,

- ✓ the set of strings it denotes is the language generated by the grammar.
- ✓ Conventionally, the productions for the start symbol are listed first.

stmt → if (expr) stmt else stmt

4. **Productions (P)** of a grammar specify the manner in which the terminals & non-terminals can be combined to form strings. Each production consists of:

- (a) A non-terminal called the **head** or left side of the production; this production defines some of the strings denoted by the head.
- (b) The symbol →. Sometimes **: : =** has been used in place of the arrow.
- (c) A body or right side consisting of zero or more terminals and non-terminals.

Example: Grammar for simple arithmetic expressions

expression → expression + term

expression → expression - term

expression → term

term → term * factor

term → term / factor

term → factor

factor → (expression)

factor → id

Terminals:

id, +, -, *, /, (,)

Non-terminals:

expression, term, factor

Start Symbol

expression

Notational Conventions

1. Terminals:

-Lowercase letters early in the alphabet: **a, b, c**.

-Operator symbols: +, *

-Punctuation symbols: parentheses, comma,

-Digits: 0, 1, . . . , 9.

-Boldface strings: **id, if**, each of which represents a single terminal symbol.

2. Non-terminals:

-Uppercase letters early in the alphabet: A, B, C.

-Letter S: start symbol.

-Lowercase, ***italic:*** *expr, stmt*.

-Uppercase letters: **E, T**, **F**

3. Uppercase letters late in the alphabet: as X, Y, Z, represent grammar symbols(T or V).

4. Lowercase letters late in the alphabet: u, v, ... ,z, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, δ: represent (possibly empty) strings of grammar symbols. Thus , a generic production can be written as $A \rightarrow \alpha$, where A-head & α-body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ... , $A_k \rightarrow \alpha_k$ with a common head A (call them A-productions) , written

$A \rightarrow \alpha_1 \mid A \rightarrow \alpha_2 \mid \ldots \mid A_k \rightarrow \alpha_k$

Call α_1, α_2, ..., α_k are the alternatives for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Example

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

- **V**: E, T, F
- **S**: E
- **T**: remaining symbols
- **P**: 08

Derivation

Consider the following example grammar with 5 productions:

1. $S \to AB$	2. $A \to aaA$	4. $B \to Bb$
	3. $A \to \lambda$	5. $B \to \lambda$

1. $S \to AB$ 2. $A \to aaA$ 4. $B \to Bb$

3. $A \to \lambda$ 5. $B \to \lambda$

Leftmost derivation order of string: aab

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

At each step, we substitute the leftmost variable

1. $S \to AB$ 2. $A \to aaA$ 4. $B \to Bb$

3. $A \to \lambda$ 5. $B \to \lambda$

Rightmost derivation order of string: aab

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

At each step, we substitute the rightmost variable

1. $S \to AB$ 2. $A \to aaA$ 4. $B \to Bb$

3. $A \to \lambda$ 5. $B \to \lambda$

Leftmost derivation of: aab

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

Rightmost derivation of: aab

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

Derivation Trees

Consider the same example grammar:

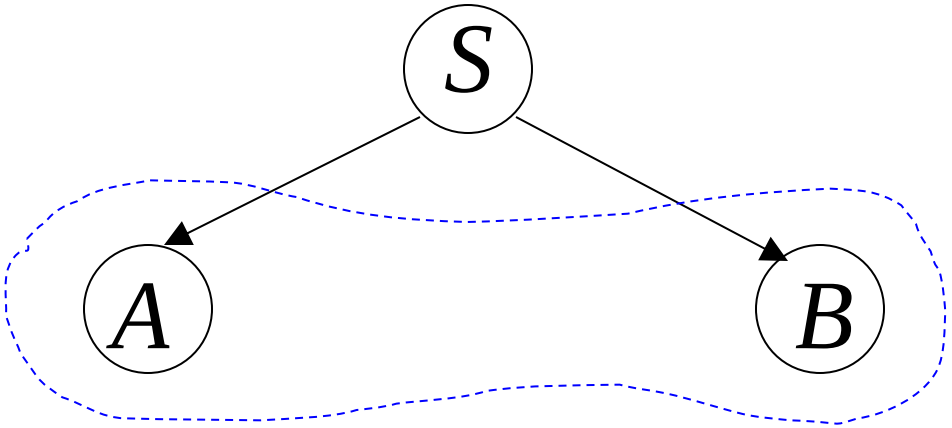
$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

And a derivation of: aab

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$

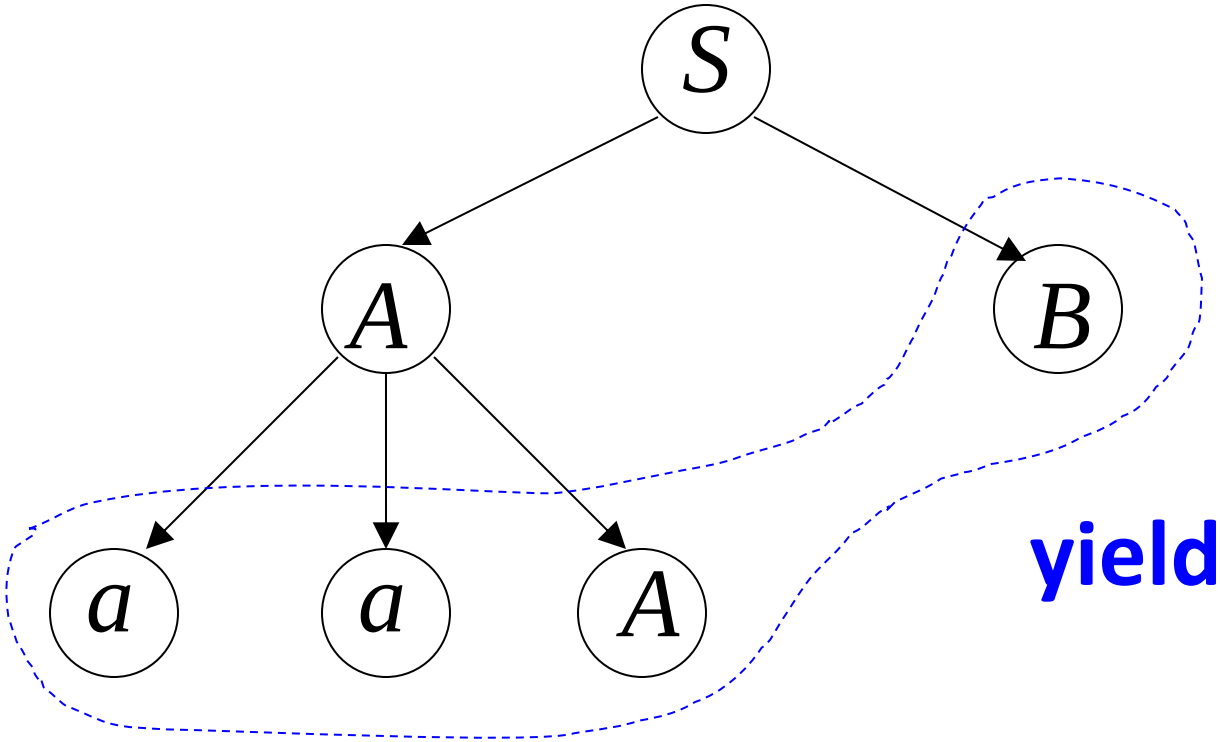
$S \Rightarrow AB$



yield AB

$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

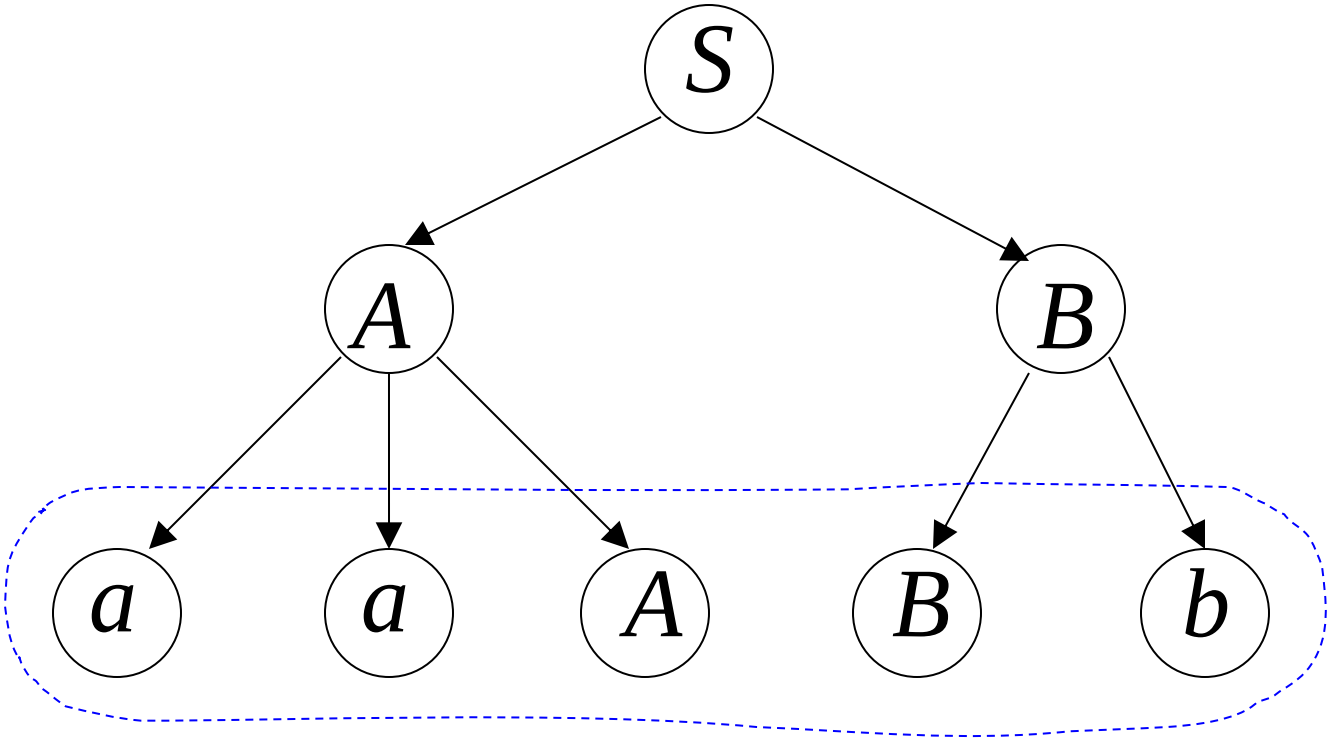
$$S \Rightarrow AB \Rightarrow aaAB$$



yield $aaAB$

$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

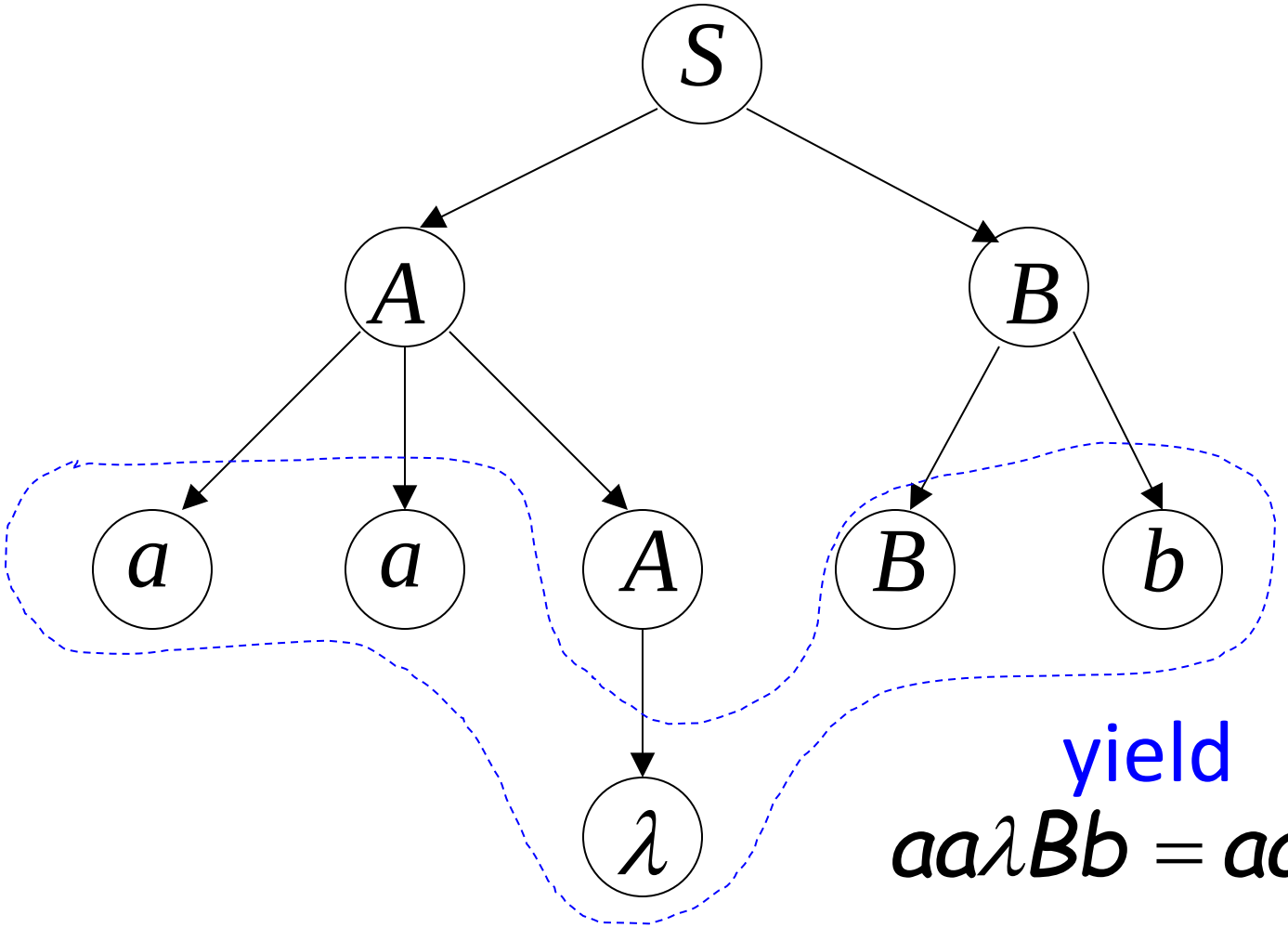
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



yield $aaABb$

$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$

$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$



yield

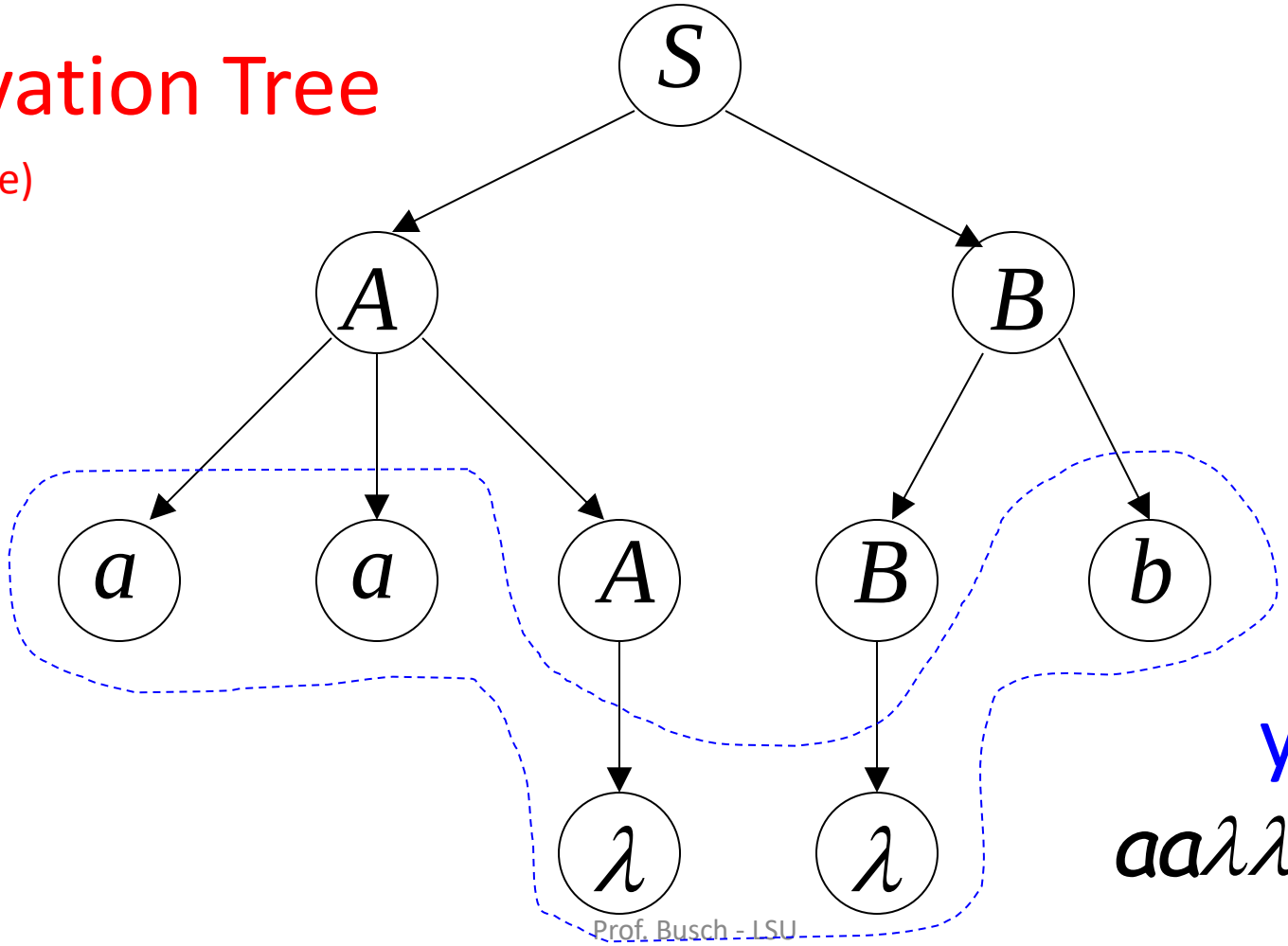
$aa\lambda Bb = aaBb$

$$S \to AB \qquad A \to aaA \mid \lambda \qquad B \to Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Derivation Tree

(parse tree)



yield

$$aa\lambda\lambda b = aab$$

Sometimes, derivation order doesn't matter

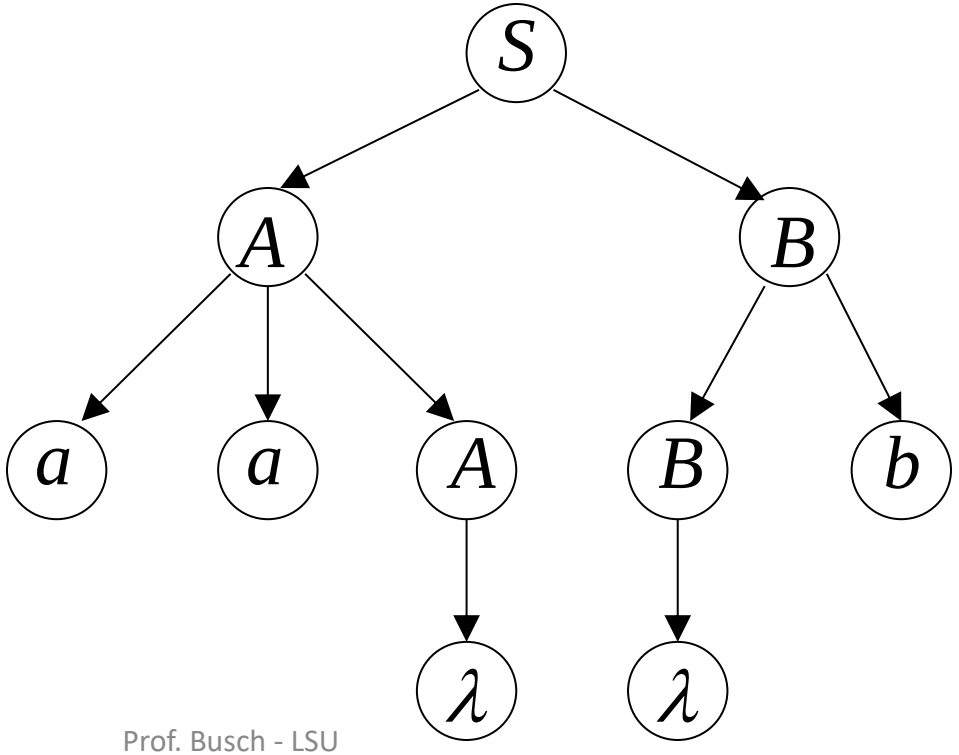
Leftmost derivation:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

Rightmost derivation:

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

Give same derivation tree



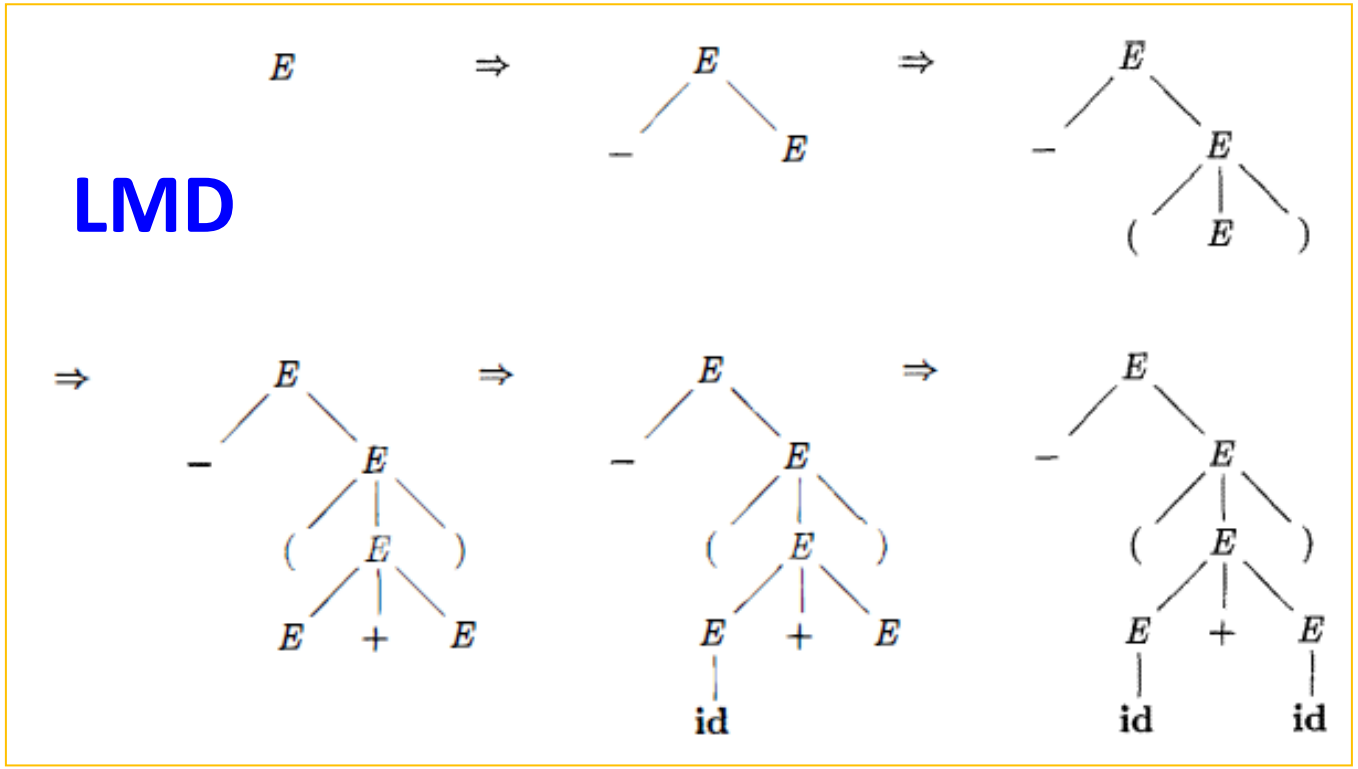
Parse Tree & Derivation

-(id+id)

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

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

LMD



Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- an ambiguous grammar is one that produces more than
  - ✓ one leftmost derivation or
  - ✓ more than one rightmost derivation

Grammar for mathematical expressions

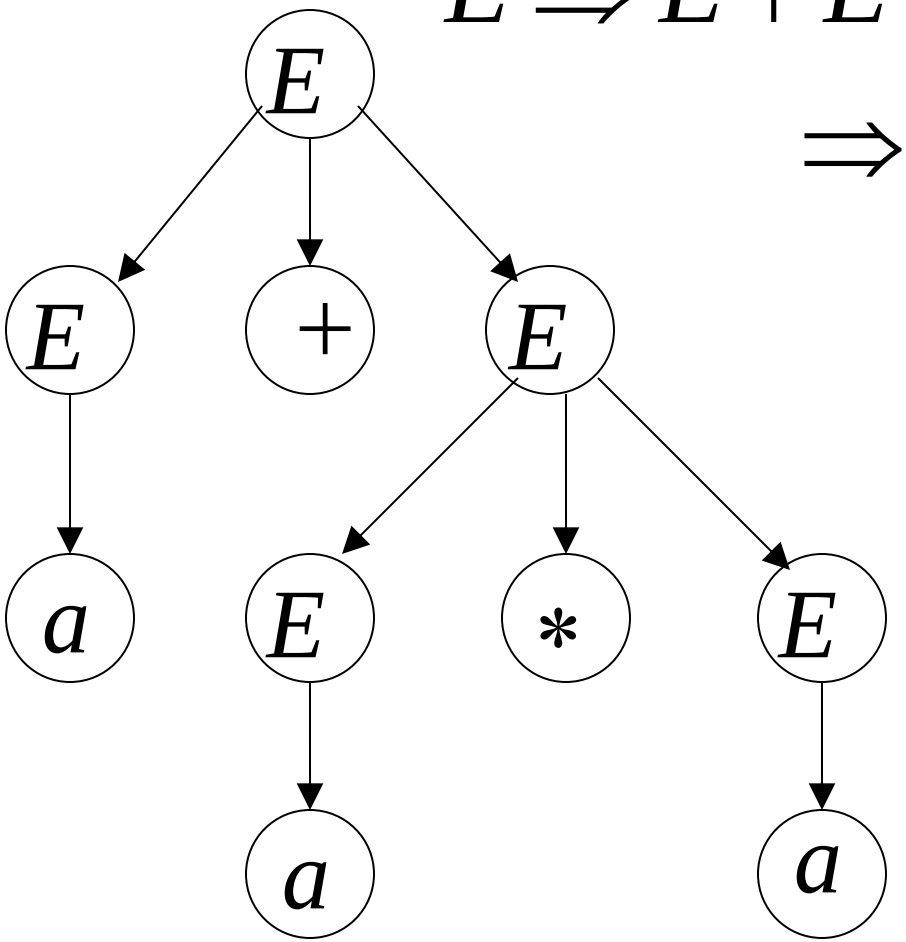
$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Example strings:

$$(a + a) * a + (a + a * (a + a))$$

Denotes any number

$$E \rightarrow E+E \mid E*E \mid (E) \mid a$$



$$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E$$

$$\Rightarrow a+a*E \Rightarrow a+a*a$$

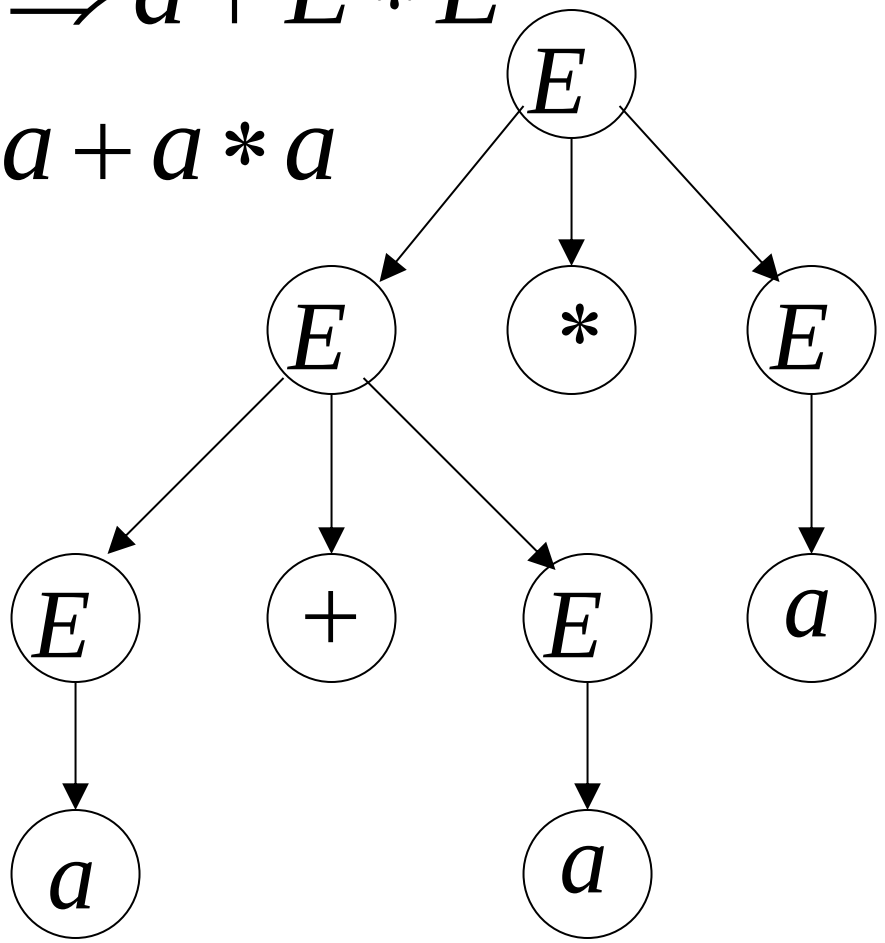
A leftmost derivation for $a+a*a$

$$E \to E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$

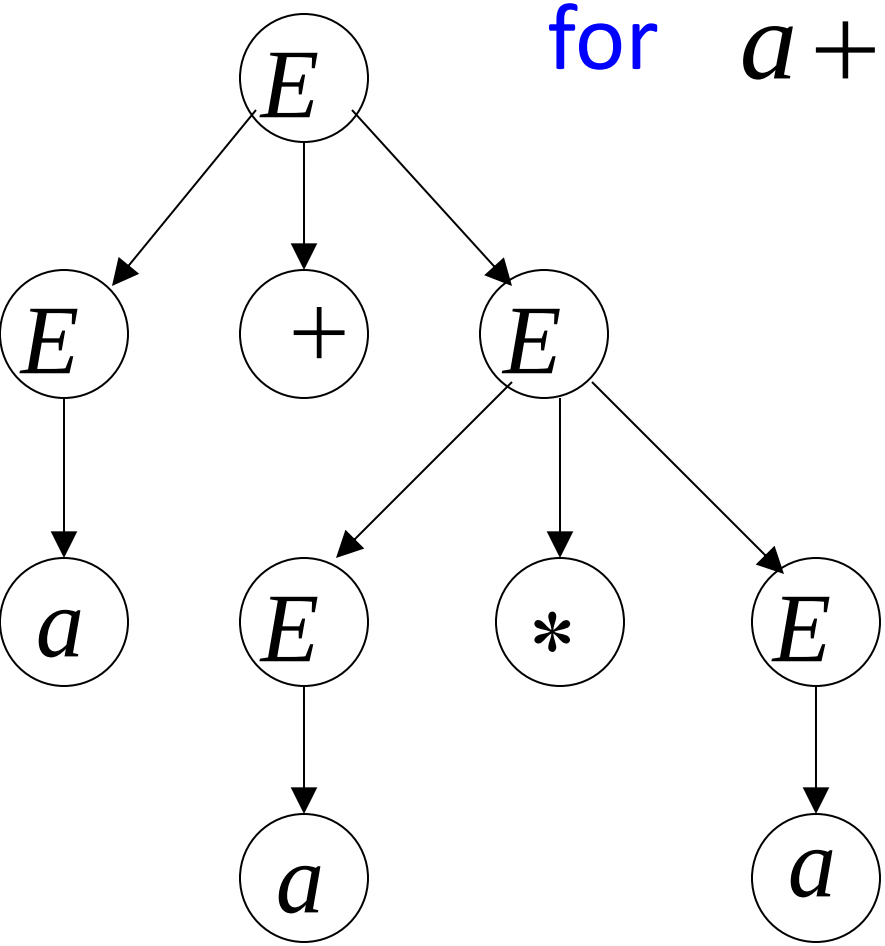
$$\Rightarrow a + a * E \Rightarrow a + a * a$$

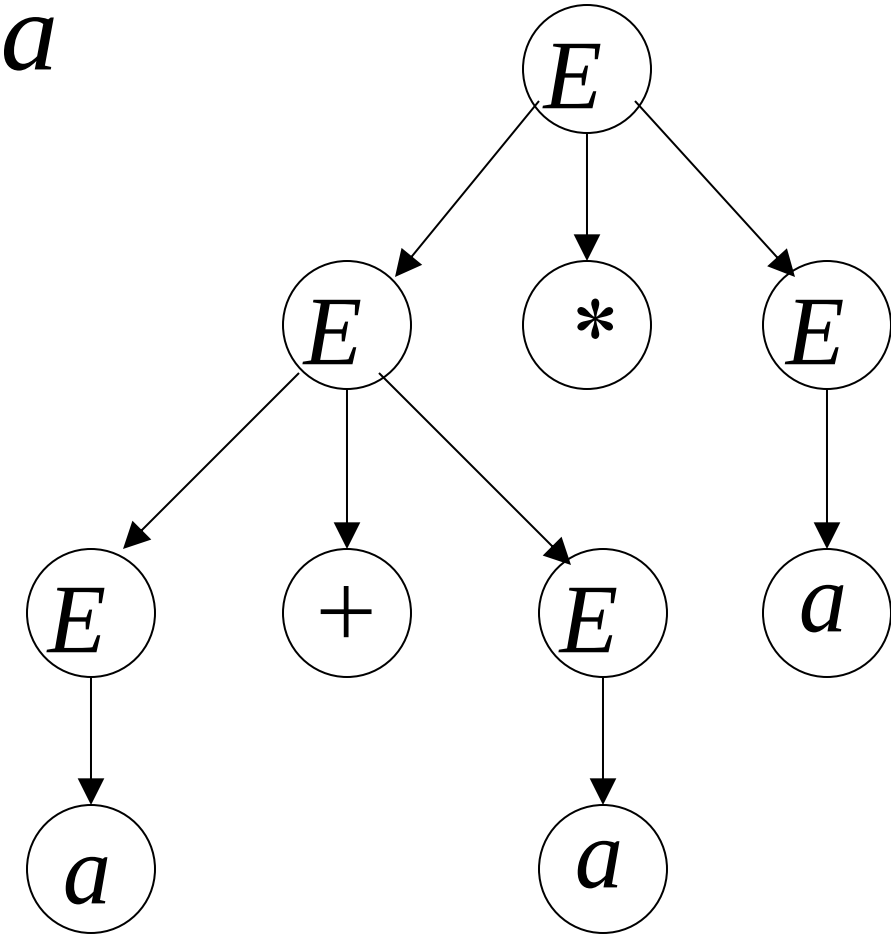
Another leftmost derivation for $a + a * a$



$$E \rightarrow E+E \mid E*E \mid (E) \mid a$$

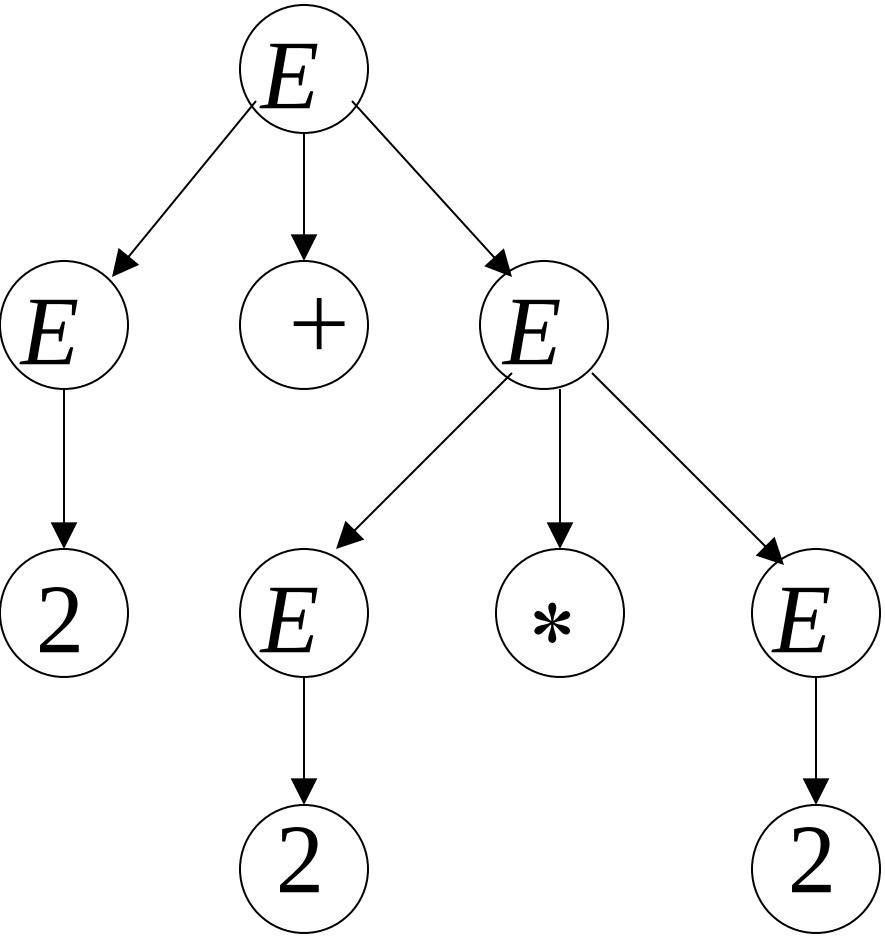
Two derivation trees for $a+a*a$

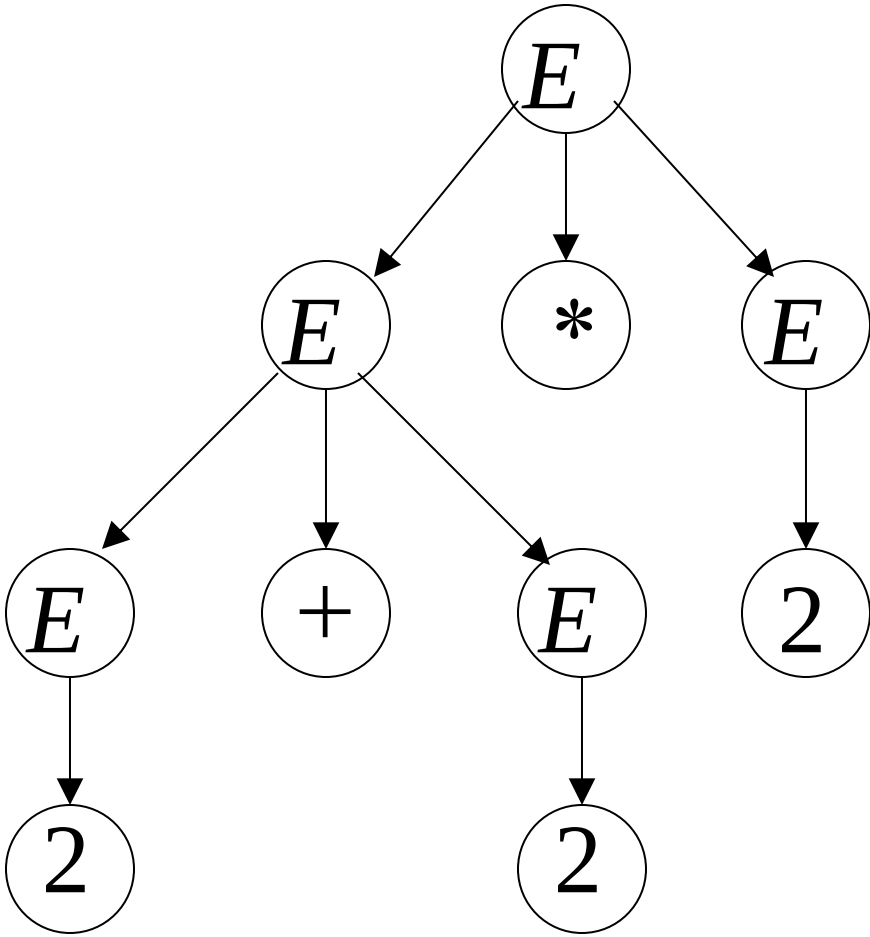




take $a = 2$

$$a + a * a = 2 + 2 * 2$$





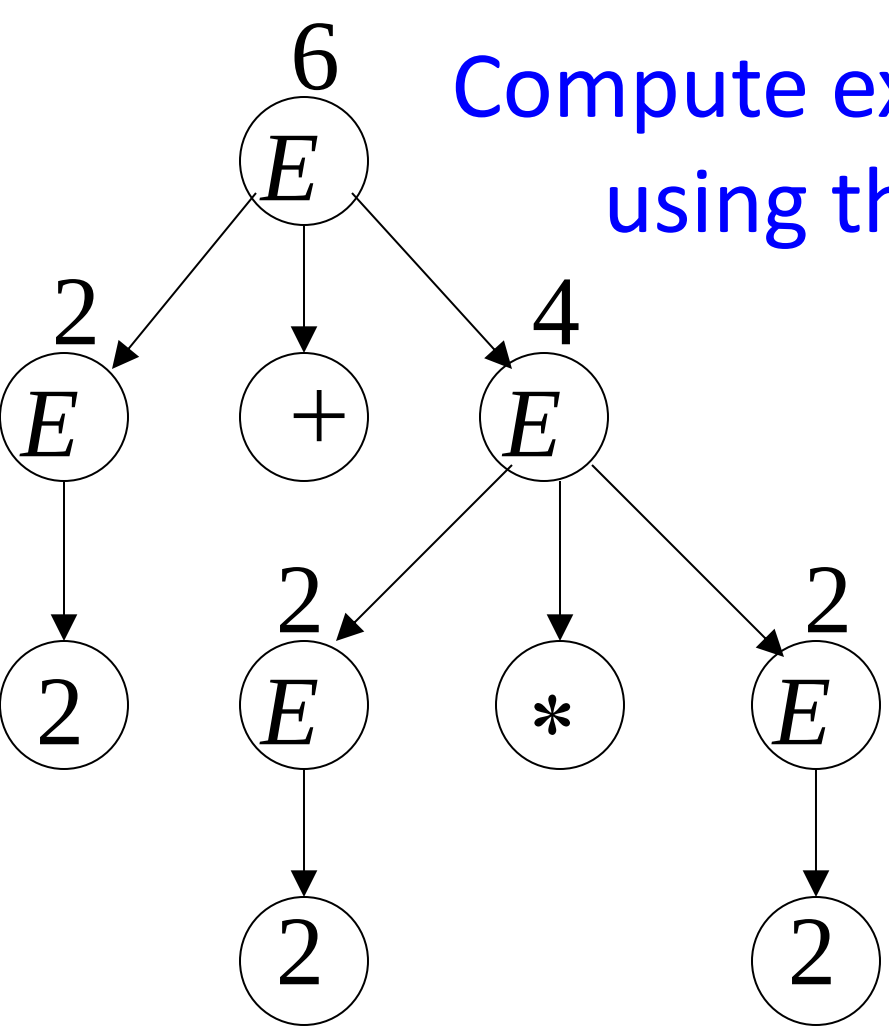
Good Tree

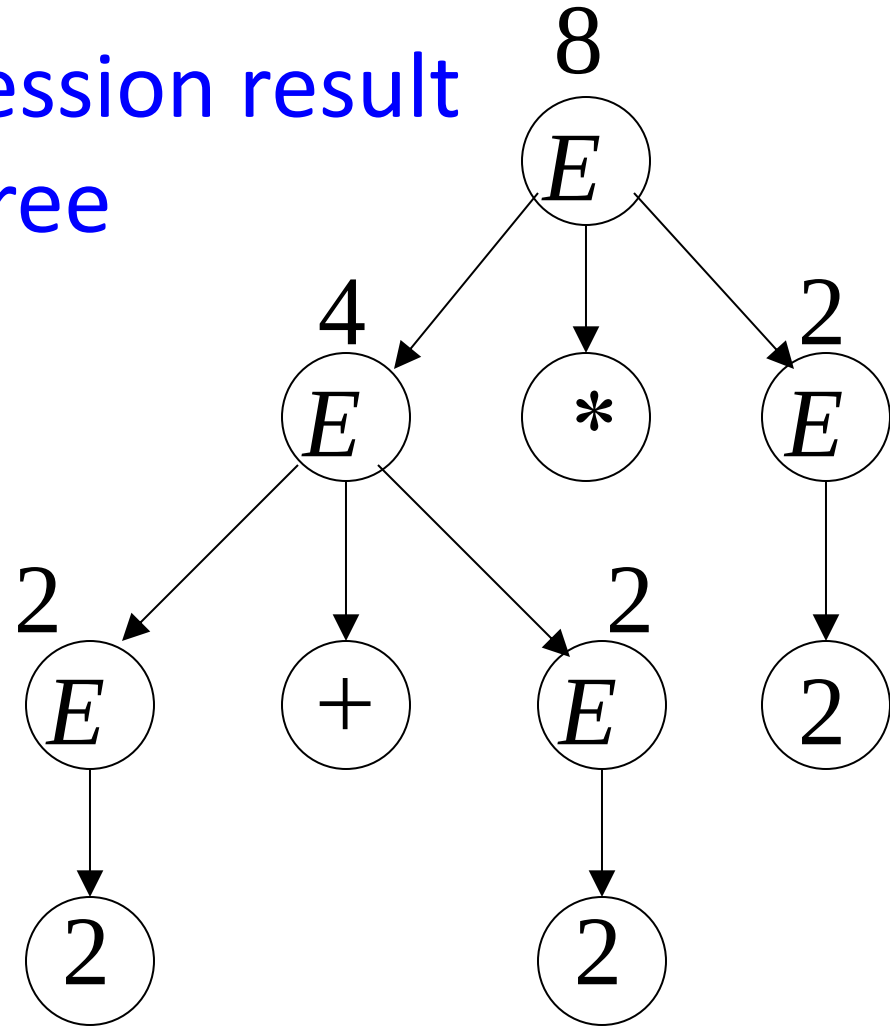
$$2+2*2=6$$

Bad Tree

$$2+2*2=8$$

Compute expression result using the tree





Two different derivation trees
may cause problems in applications which use the derivation trees:

- Evaluating expressions

- In general, in compilers for programming languages

Ambiguous Grammar:

A context-free grammar G is ambiguous if there is a string $w \in L(G)$ which has:

two different derivation trees

or

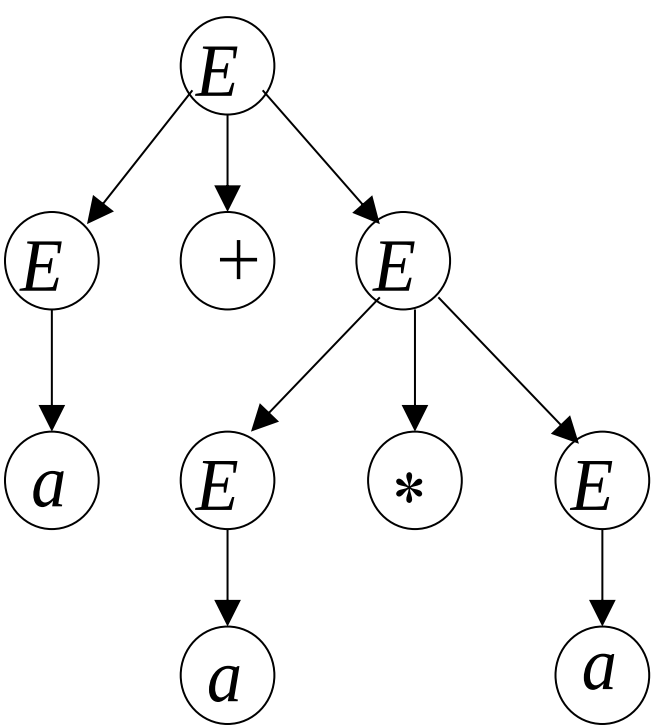
two leftmost derivations

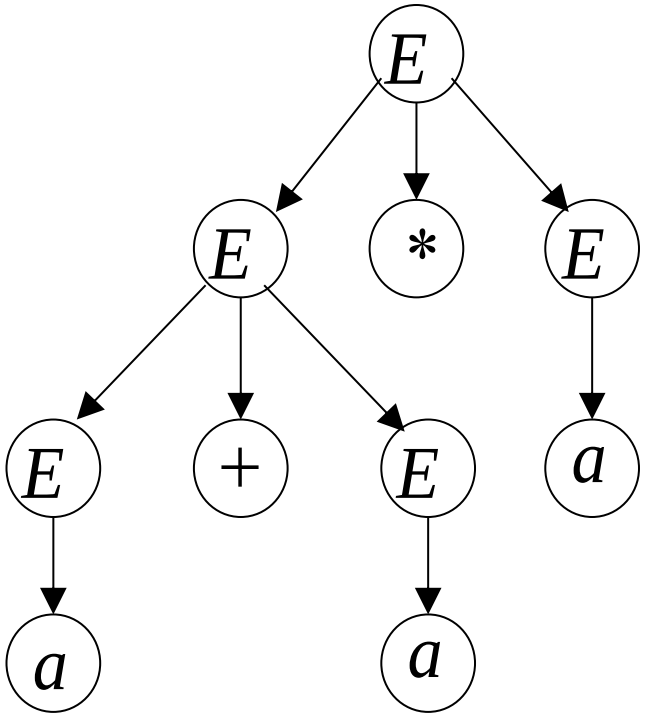
(Two different derivation trees give two different leftmost derivations and vice-versa)

Example: $E \rightarrow E+E \mid E*E \mid (E) \mid a$

this grammar is ambiguous since

string $a+a*a$ has two derivation trees





$$E \rightarrow E+E \mid E*E \mid (E) \mid a$$

this grammar is ambiguous also because

string $a+a*a$ has two leftmost derivations

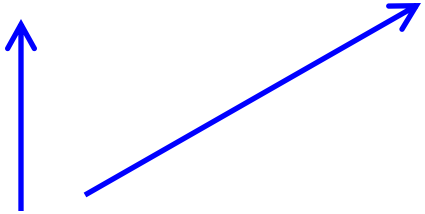
$$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E$$
$$\Rightarrow a+a*E \Rightarrow a+a*a$$

$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E$$
$$\Rightarrow a+a*E \Rightarrow a+a*a$$

Another ambiguous grammar:

IF_STMT → if EXPR then STMT

| if EXPR then STMT else STMT

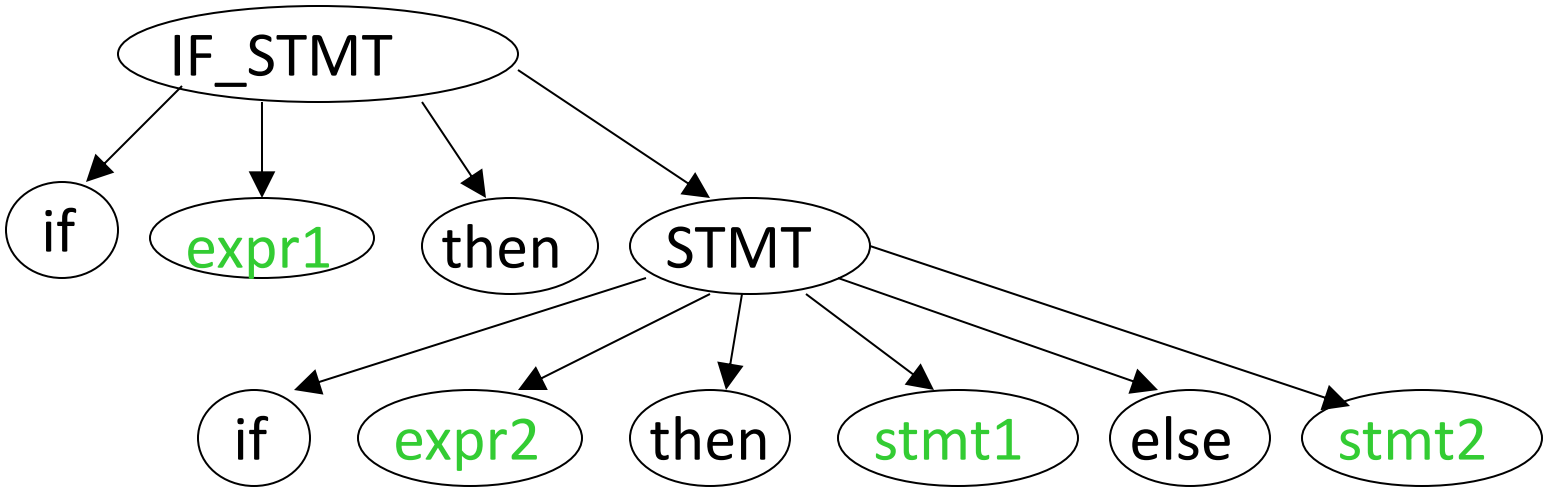


Variables

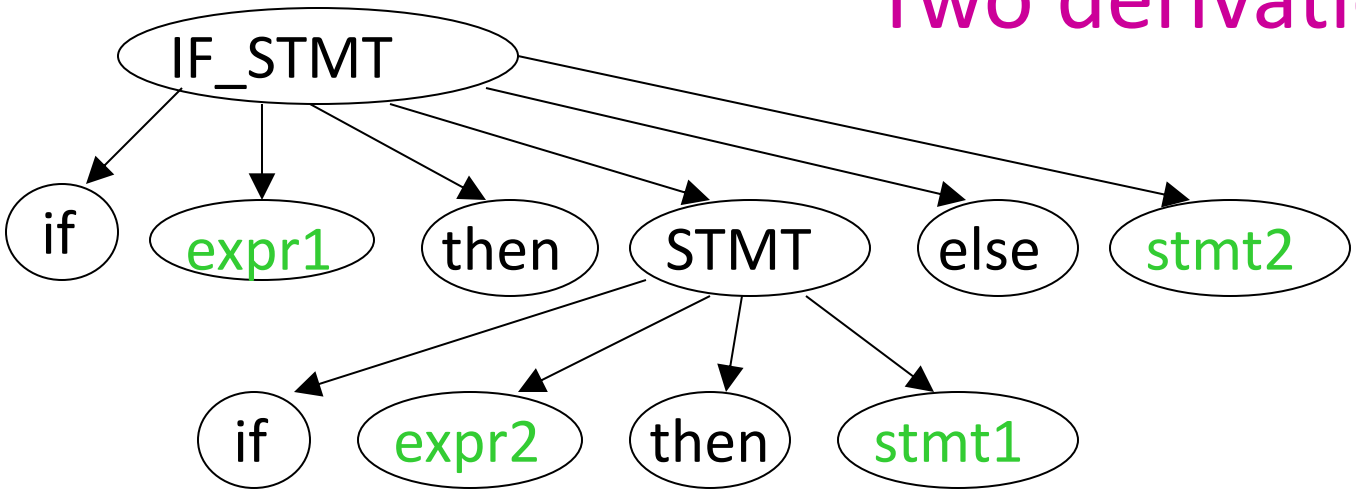
Terminals

Very common piece of grammar
in programming languages

If expr1 then if expr2 then stmt1 else stmt2



Two derivation trees



In general, ambiguity is bad
and we want to remove it

Sometimes it is possible to find
a non-ambiguous grammar for a language

But, in general ıt is difficult to achieve this

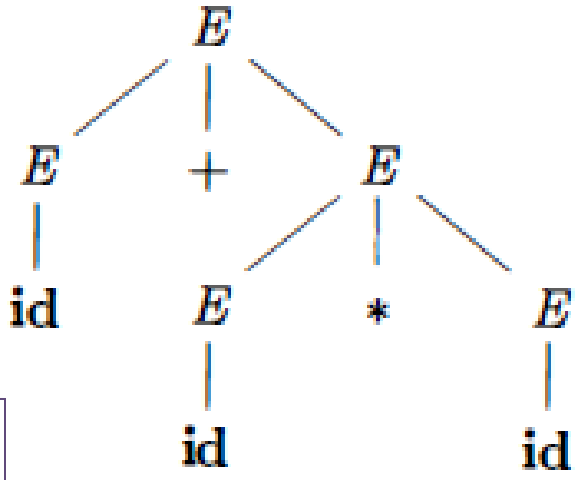
id+id*id

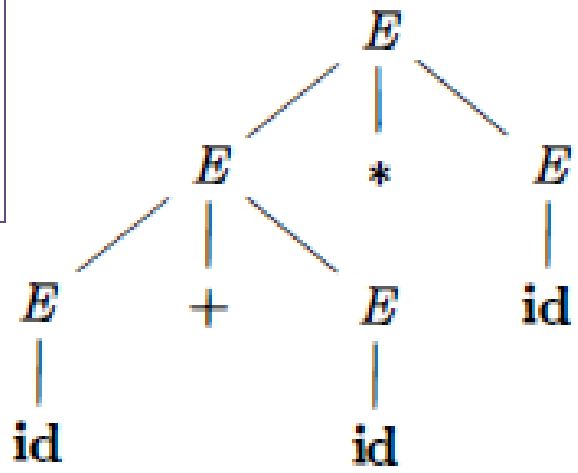
$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Two LMD

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

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





Lexical Versus Syntactic Analysis

- Everything that can be described by a regular expression can also be described by a grammar.

- **Why use regular expressions to define the lexical syntax of a language?**

Several reasons

1. Separating the syntactic structure of a language into lexical and Nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

RE: most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space.

CFG: most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's

Nested structures cannot be described by regular expressions.

Eliminating Ambiguity

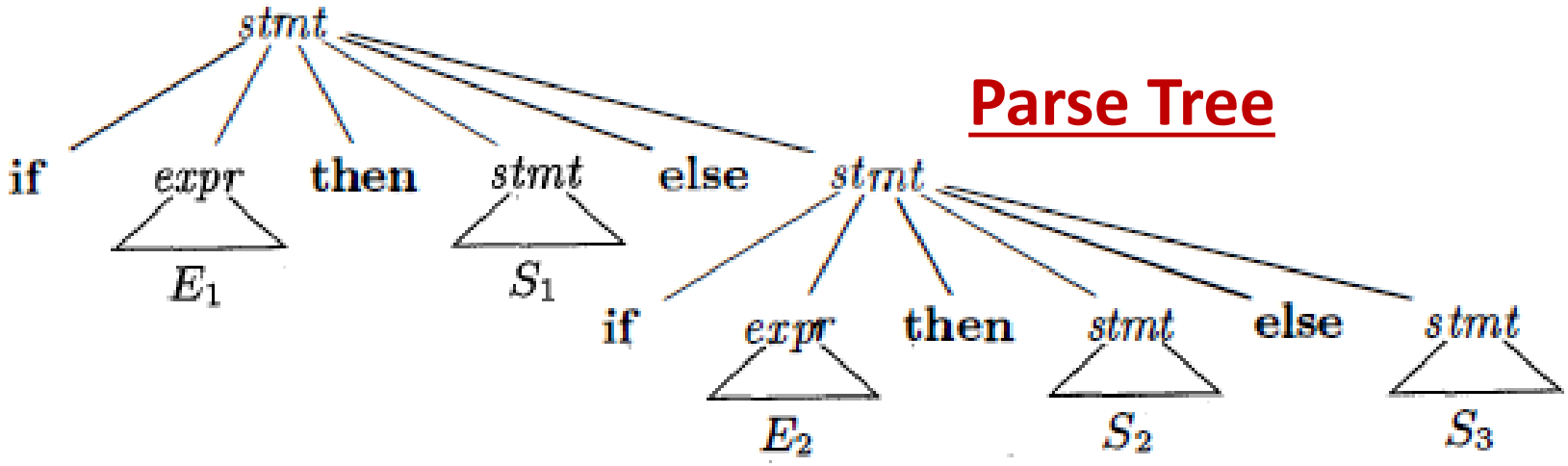
- Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.
- As an example, we shall eliminate the ambiguity from the following "*dangling-else*" grammar:

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

- "**other**" stands for any other statement.

Compound conditional statement:

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3



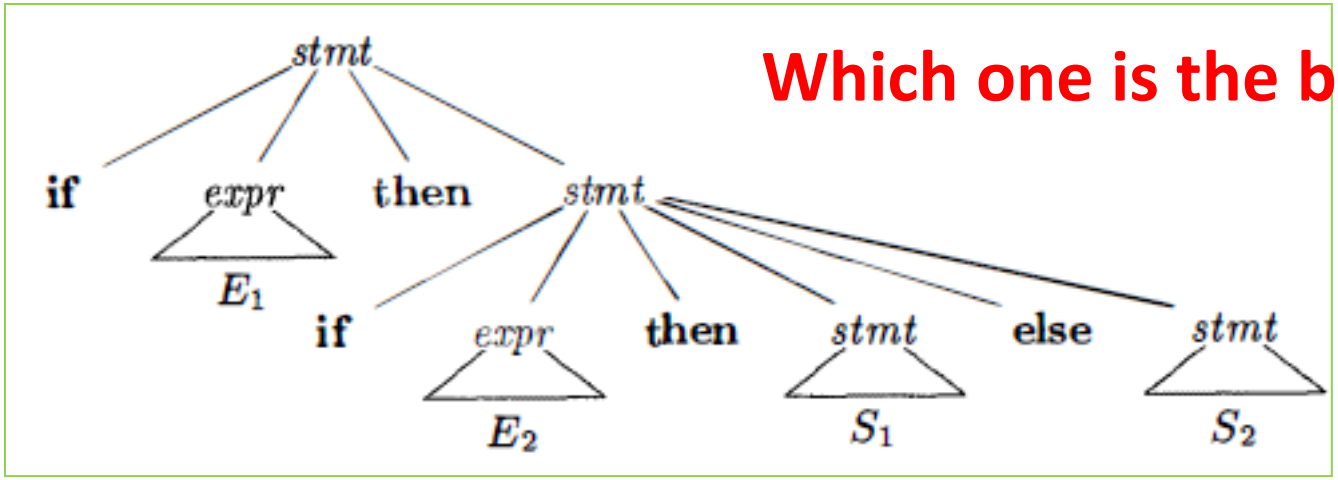
Parse Tree

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

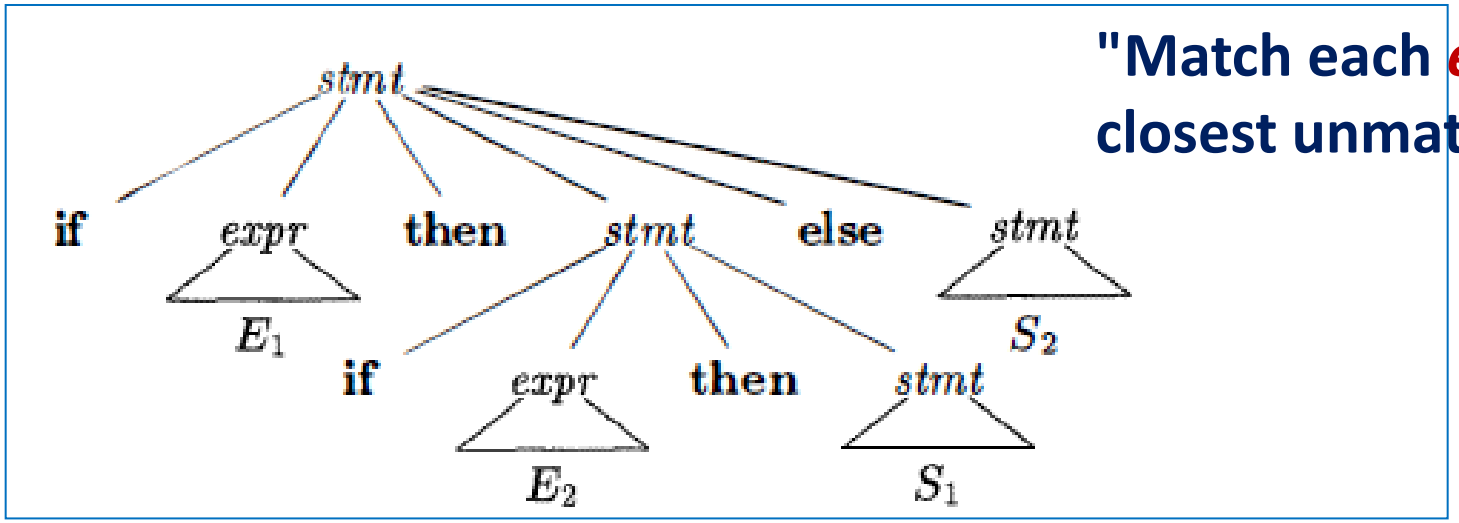
Ambiguous or non-ambiguous??

Lets try:

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$



Which one is the best??



"Match each *else* with the closest unmatched *then*"

Rewrite the dangling-else grammar:

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

❑**Ideas**: a statement appearing between a **then** and an **else** must be "matched" ; -the interior statement must not end with an unmatched or open **then**.

▪A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement.

Only one Parse Tree

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

A successful example:

Ambiguous Grammar

$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to a$$

Equivalent

Non-Ambiguous Grammar

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

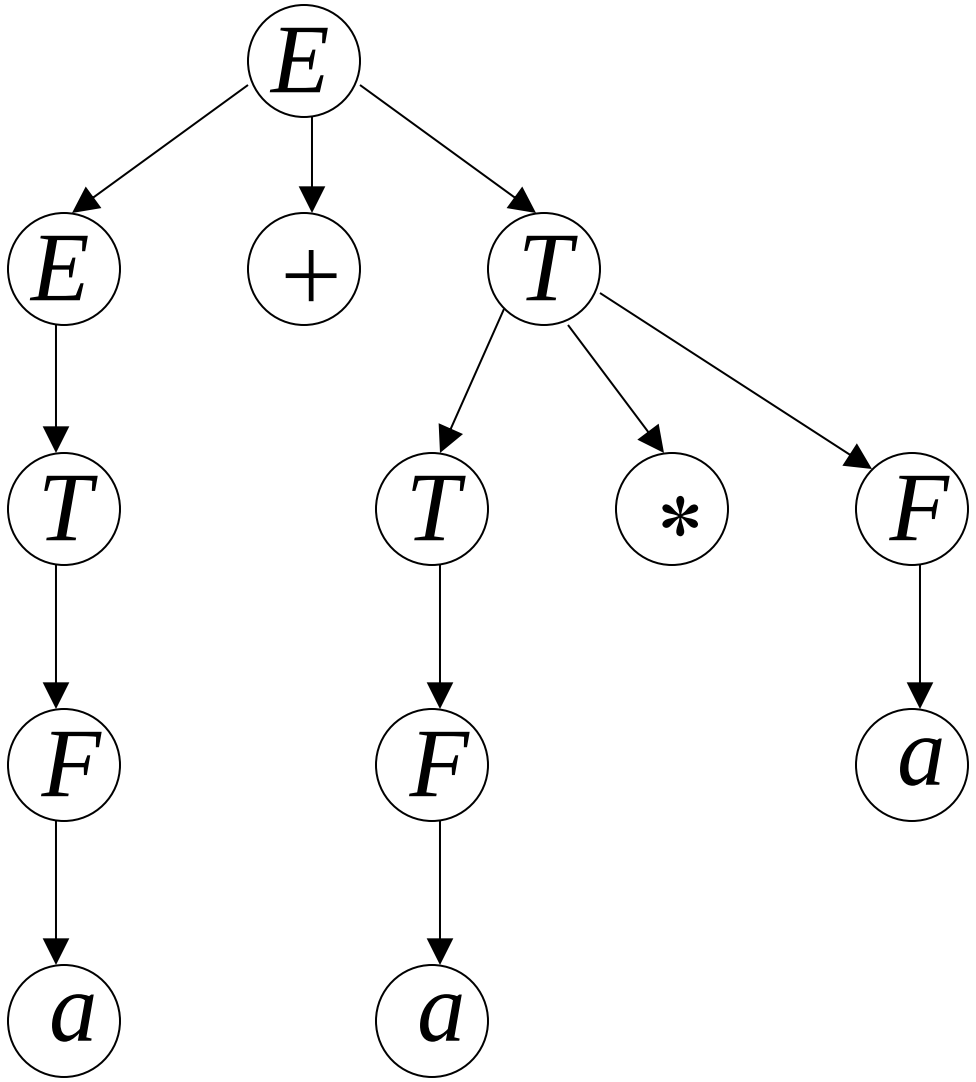
generates the same language

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$$
$$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$$

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid a$$

Unique derivation tree for

$$a+a*a$$



Elimination of Left Recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\,\alpha$ for some string α .
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.
  - $A \rightarrow A\,\alpha \mid \beta$

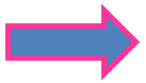
replaced by the non-left-recursive productions:

- $A \rightarrow \beta\,A'$
- $A' \rightarrow \alpha\,A' \mid \varepsilon$

- without changing the strings derivable from A.

Example : The non-left-recursive expression grammar,

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$



$$
\begin{array}{l}
E \rightarrow T\ E' \\
E' \rightarrow + T\ E' \\
T \rightarrow F\ T' \\
T' \rightarrow * F\ T' \\
F \rightarrow (E) \mid \mathbf{id}
\end{array}
$$

- eliminating immediate left recursion from the expression grammar.
- The left-recursive pair of productions

E → E + T | T are replaced by

E →T E' and E' → T E‘ | ε

- Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions.
- First, group the productions as

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

- where no β_i begins with an A. Then, replace the A-productions by

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

- The non-terminal A generates the same strings as before but is no longer left recursive.
- This procedure eliminates all left recursion from the A and A' productions (provided no α_i is ε) , but it does not eliminate left recursion involving derivations of two or more steps.

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon$$

The non-terminal **S** is left recursive because

S ⇒ Aa ⇒ Sda

but it is not immediately left recursive.

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A_1, A_2, ..., A_n.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form A_i → A_j γ by the
                productions A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ, where
                A_j → δ_1 | δ_2 | ··· | δ_k are all current A_j-productions
5)      }
6)      eliminate the immediate left recursion among the A_i-productions
7)  }
```

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon$$

Example

- Order the non-terminals: **S, A**.

❑No immediate left recursion among the S-productions, so nothing happens during the outer loop for i =i.

❑For i = 2, Substitute for S in **A→S d** to obtain the following A-productions: **A → Ac I Aad I bd I ε**

- Eliminating the immediate left recursion:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow b\ d\ A' \mid A'$$
$$A' \rightarrow c\ A' \mid a\ d\ A' \mid \epsilon$$

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.
- When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

if we have the two productions

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

- on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*.

- if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, & input begins with a nonempty string derived from α,
- Whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$!!!
- We may defer the decision by expanding A to αA`
- Then, after seeing the input derived from α, Expand A` to β_1 or β_2.
- *Left-factored:*

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

Algorithm: Left factoring a grammar.

INPUT: Grammar G

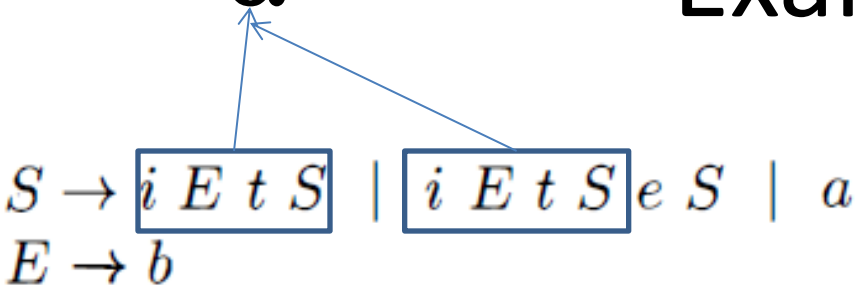
OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

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

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example



- Here, i, t, and e stand for **if, then**, and **else**;
- E: conditional expression
  S: Conditional statement

Left-factored grammars:

$$S \rightarrow i\ E\ t\ S\ S' \mid a$$
$$S' \rightarrow e\ S \mid \epsilon$$
$$E \rightarrow b$$

Longest common: iEtS

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

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

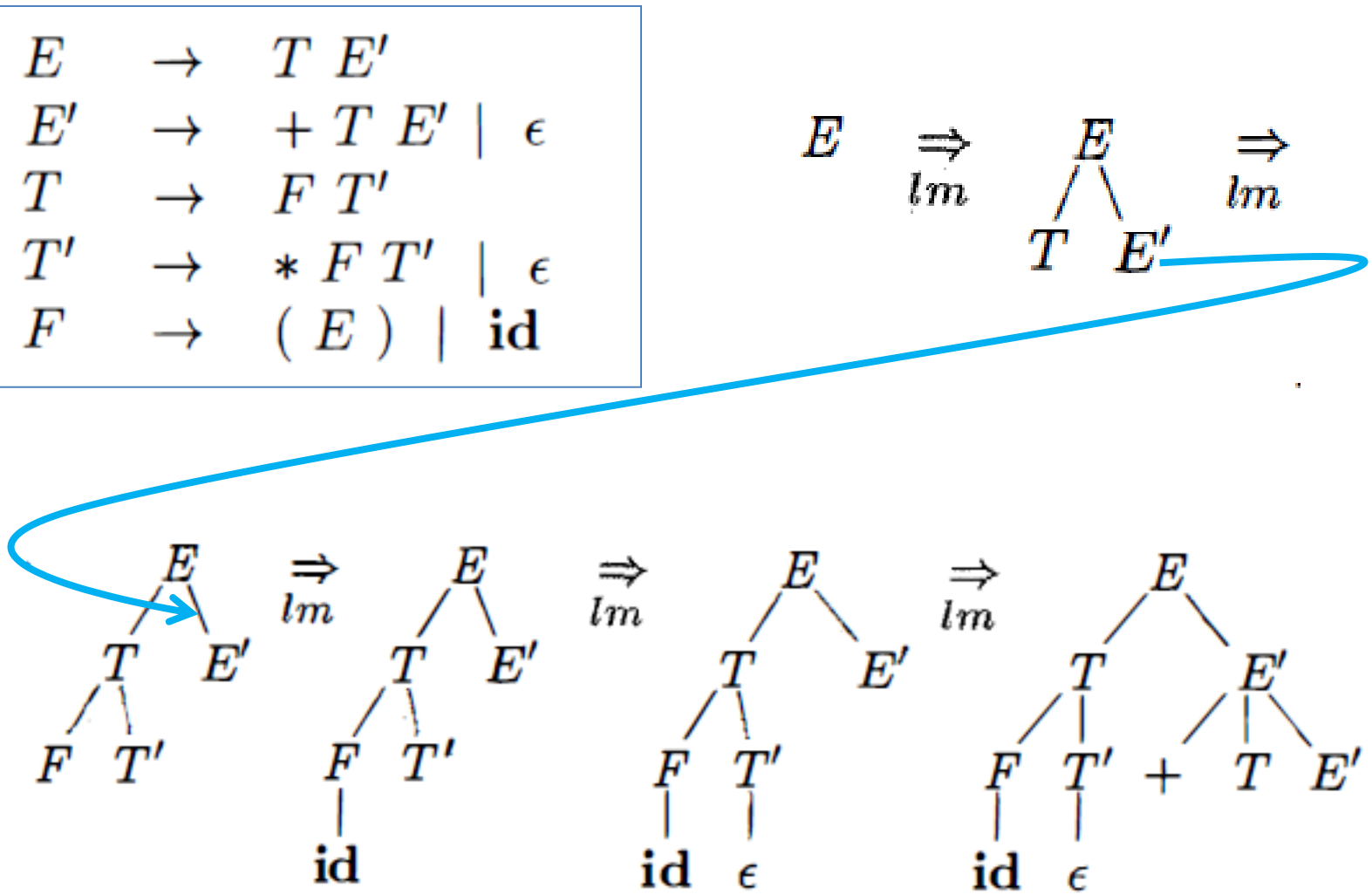
Top-Down Parsing

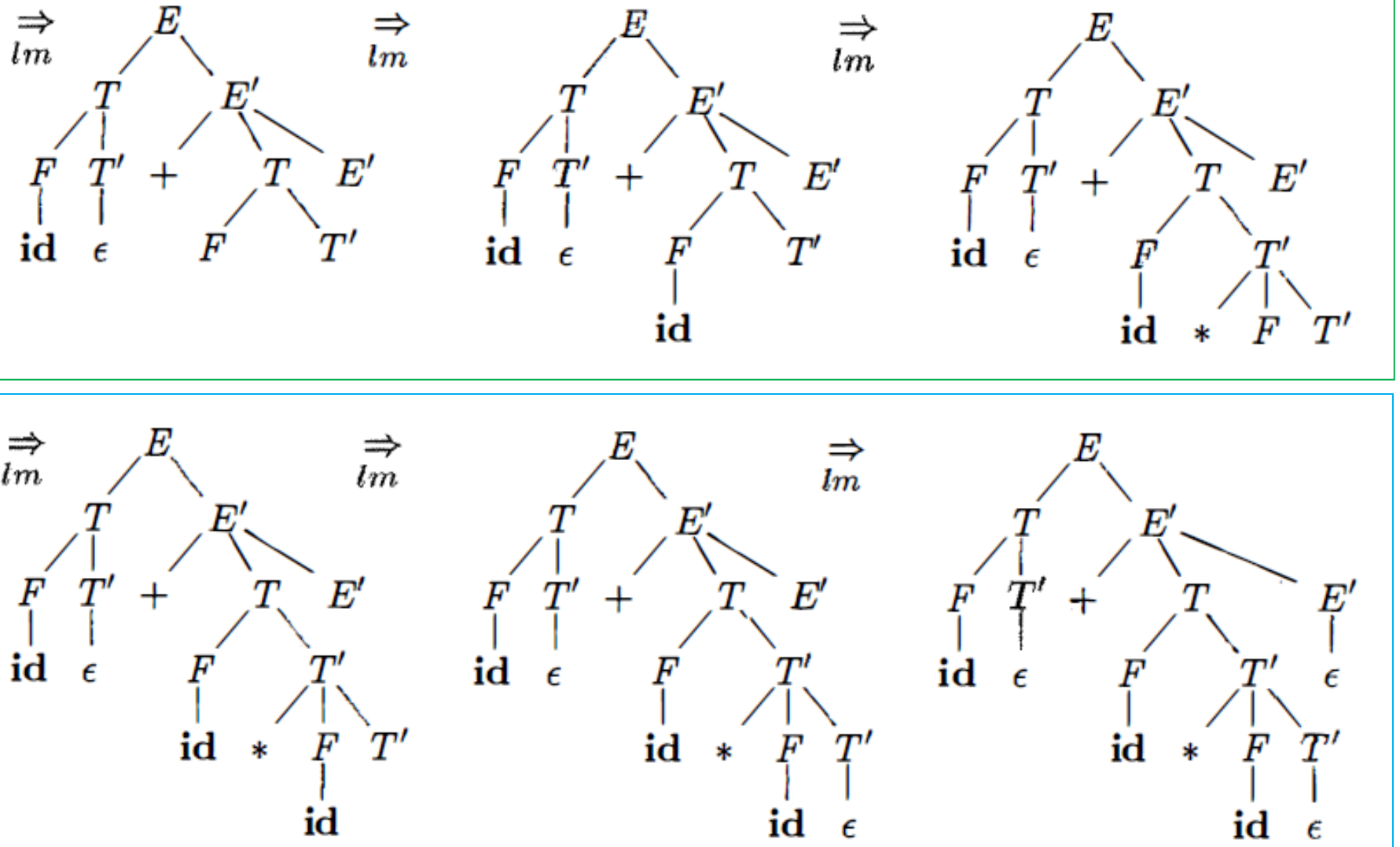
- Constructing a parse tree
- Starting from the root and creating the nodes of the parse tree in preorder (depth-first)

- Equivalent to a leftmost derivation

Input: id+id*id

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




⇒ lm
E
T E′
F T′ + T E′
id ϵ F T′
⇒ lm
E
T E′
F T′ + T E′
id ϵ F T′
id
⇒ lm
E
T E′
F T′ + T E′
id ϵ F T′
id * F T′
⇒ lm
E
T E′
F T′ + T E′
id ϵ F T′
id * F T′
id
⇒ lm
E
T E′
F T′ + T E′
id ϵ F T′
id * F T′
id ϵ
⇒ lm
E
T E′
F T′ + T E′
id ϵ F T′
id * F T′
id ϵ

Recursive-Descent Parsing

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

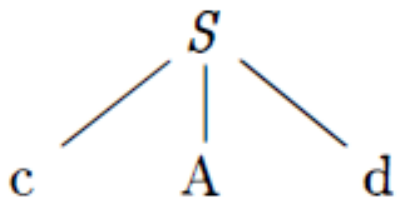
- A recursive-descent parsing program consists of a set of procedures, one for each nonterminal.
- Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.
- General recursive-descent may require backtracking; that is, it may require repeated scans over the input.
- However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.
- Even for situations like natural language parsing, backtracking is not very efficient

Modify Recursive-Descent Parsing

- To allow backtracking, recursive decent parsing algorithm needs to be modified.
- First, we cannot choose a unique A-production at line (1) , so we must try each of several productions in some order.
- Then, failure at line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production.
- Only if there are no more A-productions to try do we declare that an input error has been found.
- In order to try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1) .
- Thus, a local variable is needed to store this input pointer for future use.

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$

- *w=cad*

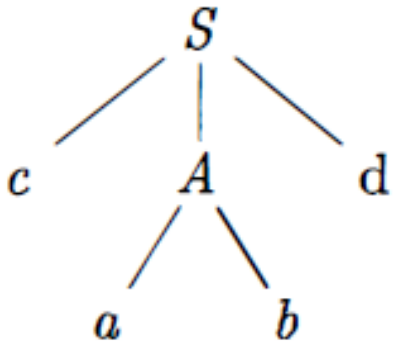


❑Match for the second input symbol, a, so advance the input pointer to **d**, the third input symbol,

-compare *d* against the next leaf, labeled **b**.

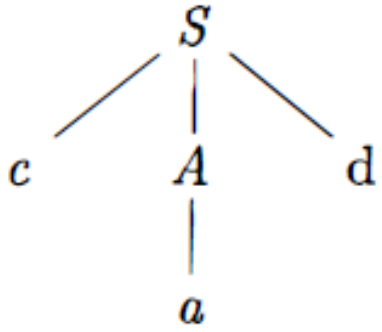
-Since **b** does not match **d**,

-report failure and go back to A to see whether there is another alternative for A that has not been tried



❑In going back to A, Reset the input pointer to position 2 , the position it had when we first came to A,

-which means that the procedure for A must store the input pointer in a local variable.



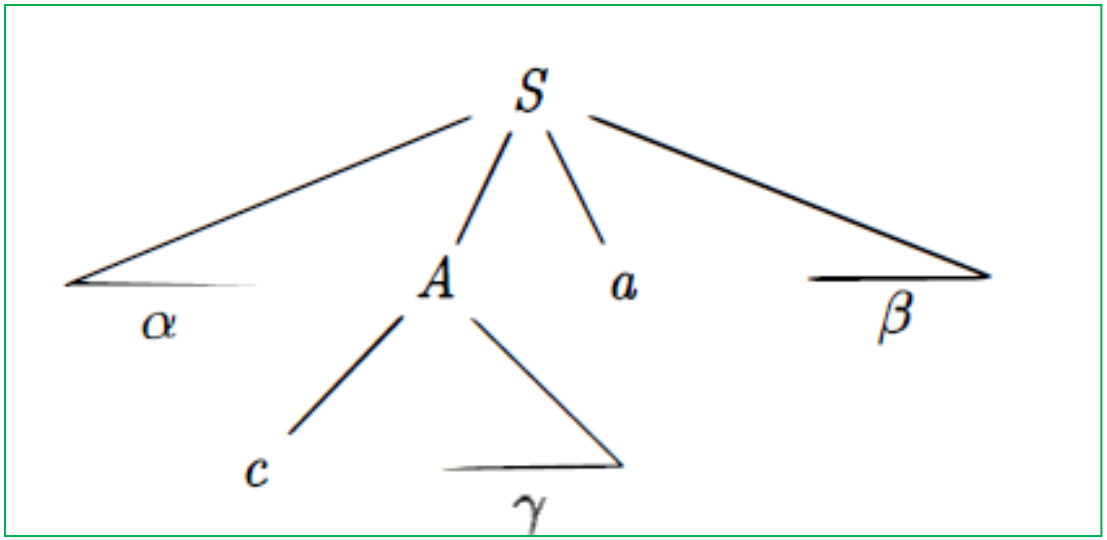
FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions:
- **FIRST**
- **FOLLOW**,
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

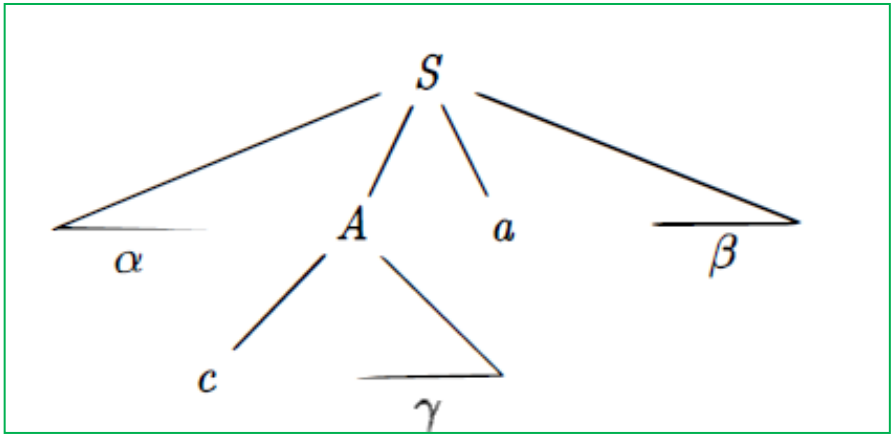
FIRST

- Define **FIRST(α)**, where α is any string of grammar symbols, to be the *set of terminals* that begin strings derived from α.
- If $\alpha \Rightarrow^* \varepsilon$, then ε is also in FIRST(α).

- $A \Rightarrow^* c\gamma$,
- FIRST(A) ={c}



FOLLOW



- FOLLOW(A) for non-terminal A to be the set of terminals ***a*** that can appear immediately to the right of A in some sentential form ; that is; the set of terminals ***a*** such that there exists a derivation of the form S $\Rightarrow^*$ αA**a**β, for some α and β.
- Note that there may have been symbols between **A** & ***a***, at some time during the derivation, but if so, they derived **ε** & disappeared.
- If **A** can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A) ;
- $ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

- To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set .

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

- To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

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

1. $\text{FIRST}(F) = \text{FIRST}(T) = \text{FIRST}(E) = \{(, \mathbf{id}\}$. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, $\text{FIRST}(T)$ must be the same as $\text{FIRST}(F)$. The same argument covers $\text{FIRST}(E)$.

2. $\text{FIRST}(E') = \{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal $+$, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. $\text{FIRST}(T') = \{*, \epsilon\}$. The reasoning is analogous to that for $\text{FIRST}(E')$.

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

4. FOLLOW(E) = FOLLOW(E') = {), \$}. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW(T') = {+,), \$}. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol +. However, since FIRST(E') contains ϵ (i.e., $E' \stackrel{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

6. FOLLOW(F) = {+, *,), \$}. The reasoning is analogous to that for T in point (5).

LL (1) Grammars

- Predictive parsers (recursive-descent) needing no backtracking,
- can be constructed for a class of grammars called **LL(1):**
- "L" : for scanning the input from left to right,
- "L" : for producing a leftmost derivation,
- “1“: for using one input symbol of look-ahead at each step

- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha | \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

Algorithm 4.31 : Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

FIRST(F)=FIRST(T)=FIRST(E)={(, **id**}

FIRST (E`) = {+, ε}

FIRST(T`) = {*, ε}

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

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

FOLLOW(T)=FOLLOW(T`) ={+,),$}

FOLLOW(F)= {+, *,),$}

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

Consider production $E \to TE'$. Since

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \mathbf{id}\}$$

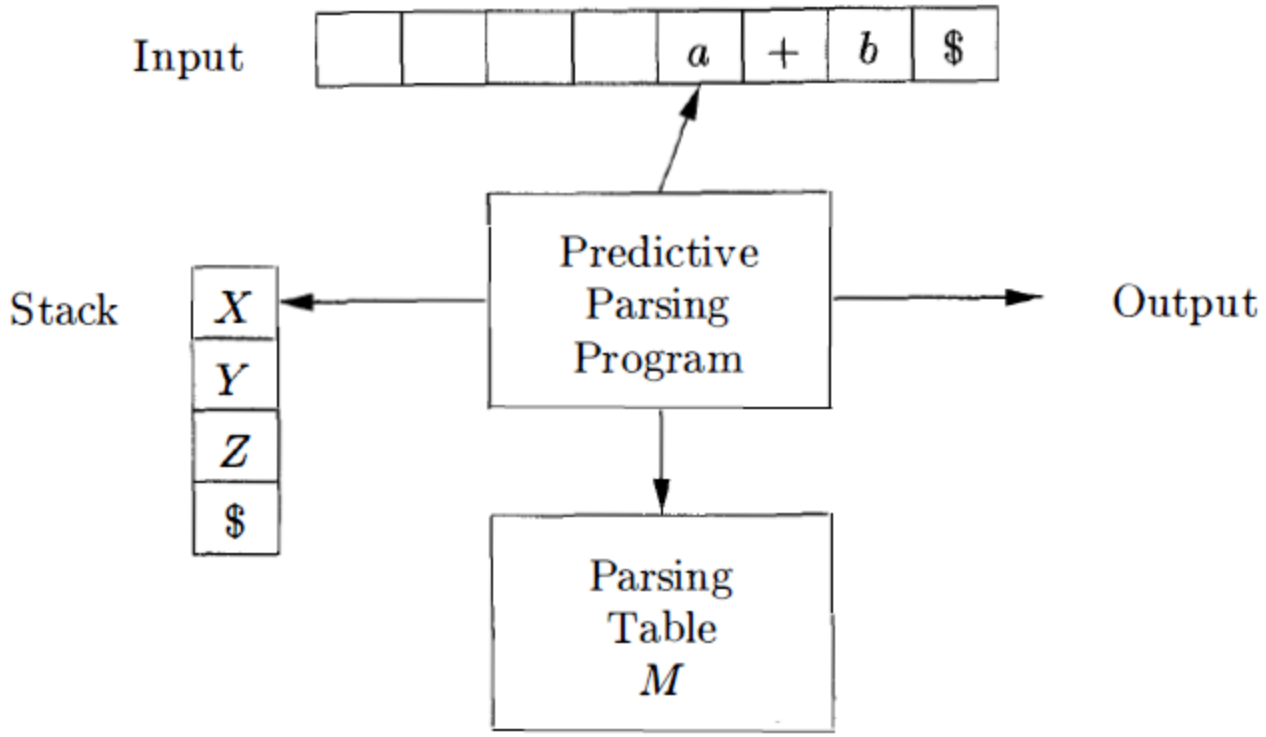
this production is added to $M[E, (]$ and $M[E, \mathbf{id}]$. Production $E' \to +TE'$ is added to $M[E', +]$ since $\text{FIRST}(+TE') = \{+\}$. Since $\text{FOLLOW}(E') = \{), \$\}$, production $E' \to \epsilon$ is added to $M[E',)]$ and $M[E', \$]$. $\square$

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

Ambiguous (multiple Entry)

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		$E \rightarrow b$				

Non-recursive Predictive Parsing



Algorithm 4.34 : Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

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

Example 4.35: Consider grammar (4.28); we have already seen its the parsing table in Fig. 4.17. On input **id** + **id** ∗ **id**, the nonrecursive predictive parser of Algorithm 4.34 makes the sequence of moves in Fig. 4.21. These moves correspond to a leftmost derivation (see Fig. 4.12 for the full derivation):

$$
E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots
$$

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

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 $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$*\ \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$*\ \mathbf{id}\$$	output $T' \rightarrow *\ FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Error Recovery in Predictive Parsing

Example 4.36 : Using FIRST and FOLLOW symbols as synchronizing tokens works reasonably well when expressions are parsed according to the usual grammar (4.28). The parsing table for this grammar in Fig. 4.17 is repeated in Fig. 4.22, with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the nonterminal in question. The FOLLOW sets for the nonterminals are obtained from Example 4.30.

FOLLOW(E)=FOLLOW(E`) ={),$}
FOLLOW(T)=FOLLOW(T`) ={+,),$}
FOLLOW(F)= {+, *,),$}

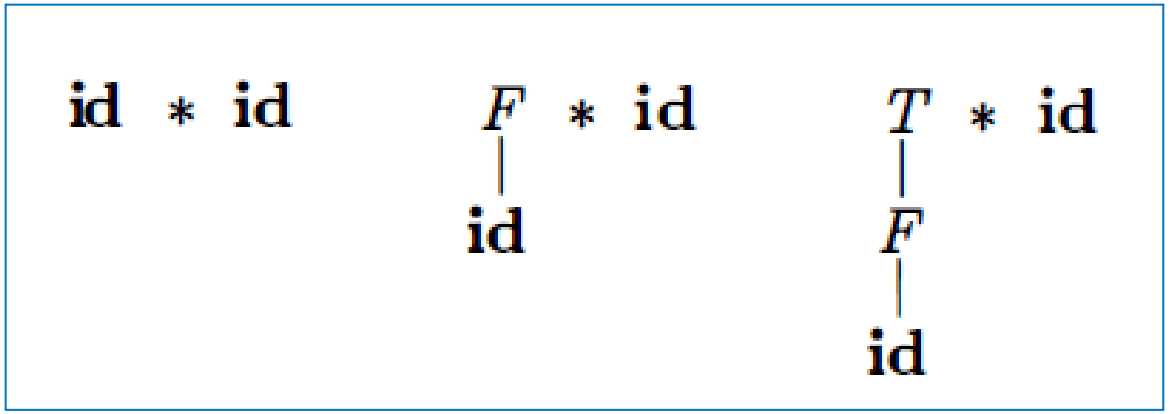
NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$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

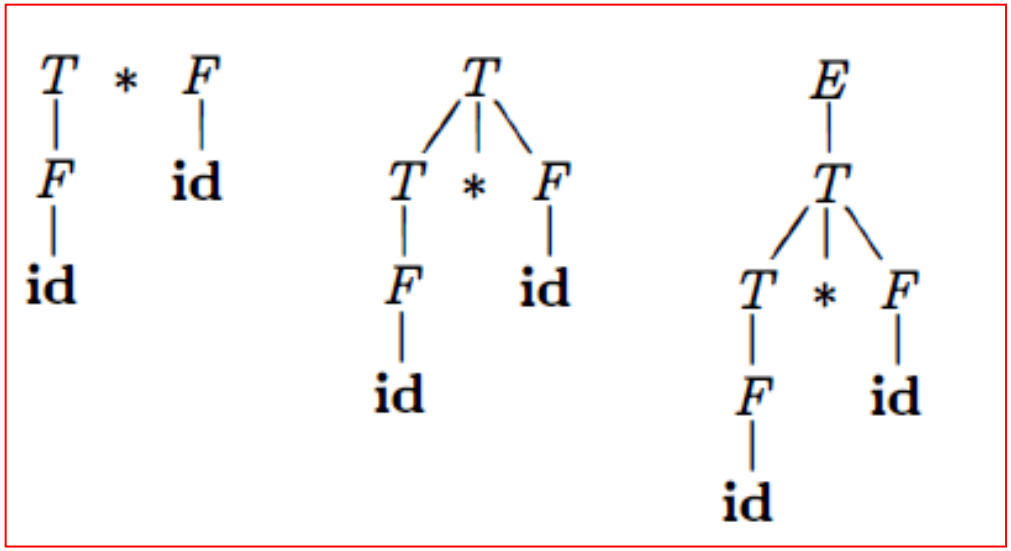
$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

STACK	INPUT	REMARK
$E\ \$$	$)\ \mathbf{id} * + \mathbf{id}\ \$$	error, skip)
$E\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	**id** is in FIRST(E)
$T\ E'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$F\ T'\ E'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$\mathbf{id}\ T'\ E'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$T'\ E'\ \$$	$* + \mathbf{id}\ \$$	
$*\ F\ T'\ E'\ \$$	$* + \mathbf{id}\ \$$	
$F\ T'\ E'\ \$$	$+\ \mathbf{id}\ \$$	error, $M[F, +] =$ synch
$T'\ E'\ \$$	$+\ \mathbf{id}\ \$$	F has been popped
$E'\ \$$	$+\ \mathbf{id}\ \$$	
$+\ T\ E'\ \$$	$+\ \mathbf{id}\ \$$	
$T\ E'\ \$$	$\mathbf{id}\ \$$	
$F\ T'\ E'\ \$$	$\mathbf{id}\ \$$	
$\mathbf{id}\ T'\ E'\ \$$	$\mathbf{id}\ \$$	
$T'\ E'\ \$$	$\$$	
$E'\ \$$	$\$$	
$\$$	$\$$	

Bottom-Up Parsing

Shift
Reduce
Algorithm





Reductions

- process of *"reducing"* a string **w** to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.
- **The key decisions:**

✓ when to reduce

✓ what production to apply

Reductions

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

Example 4.37 : The snapshots in Fig. 4.25 illustrate a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string $\mathbf{id} * \mathbf{id}$. The first reduction produces $F * \mathbf{id}$ by reducing the leftmost **id** to F, using the production $F \rightarrow \mathbf{id}$. The second reduction produces $T * \mathbf{id}$ by reducing F to T.

Now, we have a choice between reducing the string T, which is the body of $E \rightarrow T$, and the string consisting of the second **id**, which is the body of $F \rightarrow \mathbf{id}$. Rather than reduce T to E, the second **id** is reduced to T, resulting in the string $T * F$. This string then reduces to T. The parse completes with the reduction of T to the start symbol E. □

- By definition, ***a reduction is the reverse of a step in a derivation***
- The goal of bottom-up parsing is therefore to construct a derivation in reverse.

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

This derivation is in fact a rightmost derivation.

Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.
- Informally, a "**handle**" is a substring that matches the body of a production,
- and whose reduction represents one step along the reverse of a rightmost derivation.

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

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

A Handle

Formally, if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$, as in Fig. 4.27, then production $A \to \beta$ in the position following α is a *handle* of $\alpha\beta w$. Alternatively, a handle of a right-sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ.

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we refer to the body β rather than $A \to \beta$ as a handle. Note we say "a handle" rather than "the handle," because the grammar could be ambiguous, with more than one rightmost derivation of $\alpha\beta w$. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

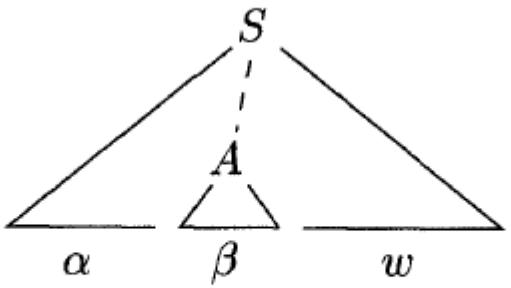


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

Shift Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.
- We use $ to mark the bottom of the stack and also the right end of the input.
- Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing.

- Initially, the stack is empty, and the string w is on the input, as follows

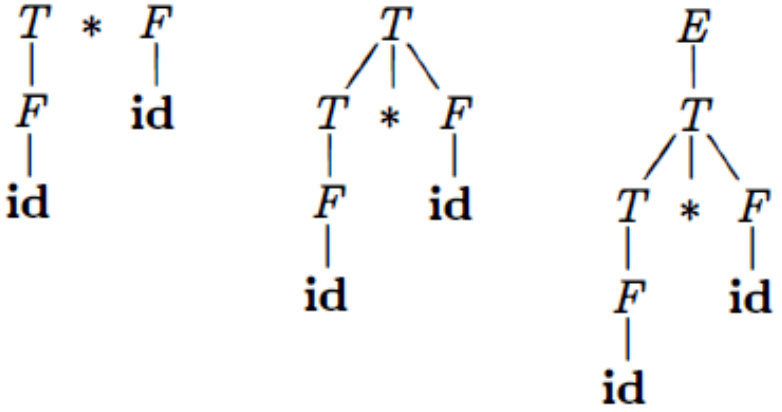
STACK	INPUT
$	w$

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
- It then reduces β to the head of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Shift-Reduce Parsing

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



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

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

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

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

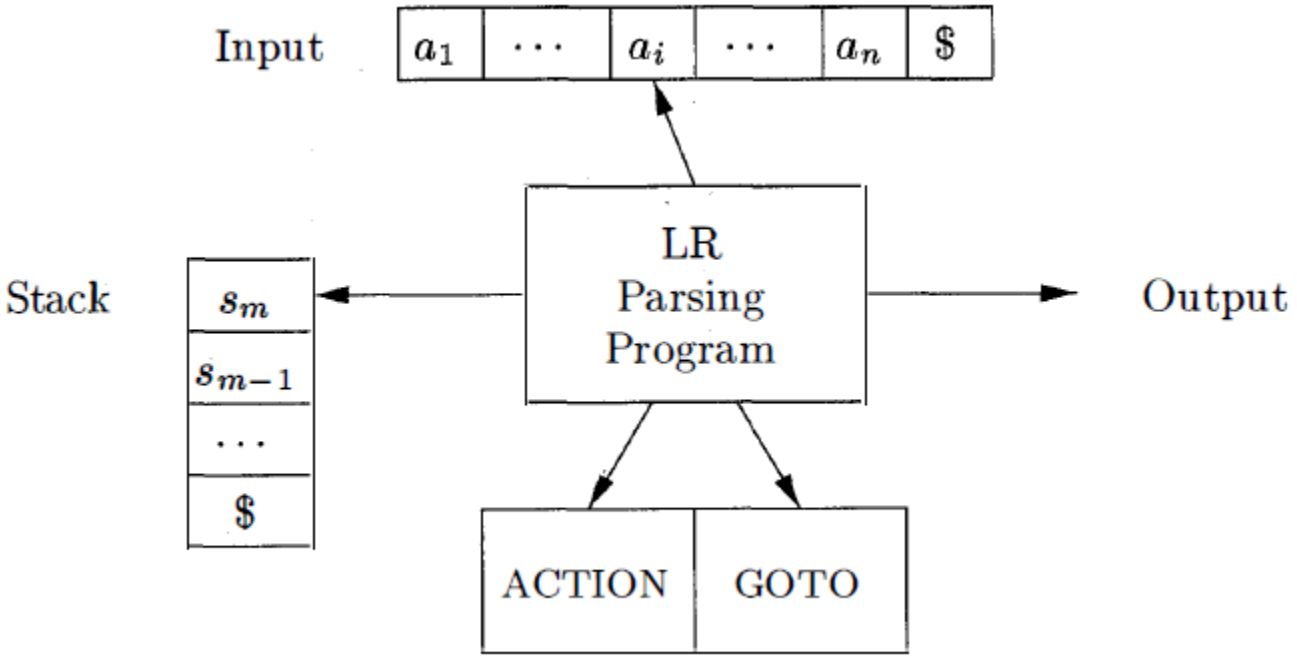
Why LR Parsers?

- LR parsers are table-driven, much like the nonrecursive LL parsers of Section
- 4.4.4. A grammar for which we can construct a parsing table using one of
- the methods in this section and the next is said to be an LR grammar. Intuitively,
- for a grammar to be LR it is sufficient that a left-to-right shift-reduce
- parser be able to recognize handles of right-sentential forms when they appear
- on top of the stack.
- LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.
- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods (see the bibliographic notes).
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.

- The principal drawback of the LR method is that it is too much work to
- construct an LR parser by hand for a typical programming-language grammar.
- A specialized tool, an LR parser generator, is needed. Fortunately, many such
- generators are available, and we shall discuss one of the most commonly used
- ones, Yacc , in Section 4.9. Such a generator takes a context-free grammar and
- automatically produces a parser for that grammar. If the grammar contains
- ambiguities or other constructs that are difficult to parse in a left-to-right scan
- of the input, then the parser generator locates these constructs and provides
- detailed diagnostic messages.

The LR-Parsing Algorithm



OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

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

Example 4.45: Figure 4.37 shows the ACTION and GOTO functions of an LR-parsing table for the expression grammar (4.1), repeated here with the productions numbered:

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

The codes for the actions are:

1. si means shift and stack state i,
2. rj means reduce by the production numbered j,
3. acc means accept,
4. blank means error.

Note that the value of GOTO[s, a] for terminal a is found in the ACTION field connected with the shift action on input a for state s. The GOTO field gives GOTO[s, A] for nonterminals A. Although we have not yet explained how the entries for Fig. 4.37 were selected, we shall deal with this issue shortly.

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			

On input **id** $*$ **id** $+$ **id**, the sequence of stack and input contents is shown in Fig. 4.38. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 4.37 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has been pushed onto the stack, and **id** has been removed from the input.

Then, $*$ becomes the current input symbol, and the action of state 5 on input $*$ is to reduce by $F \rightarrow$ **id**. One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on F is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly. $\square$