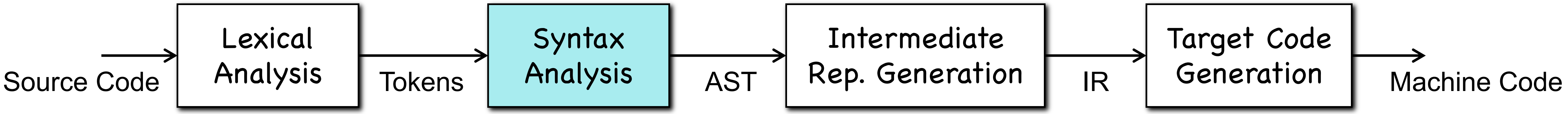
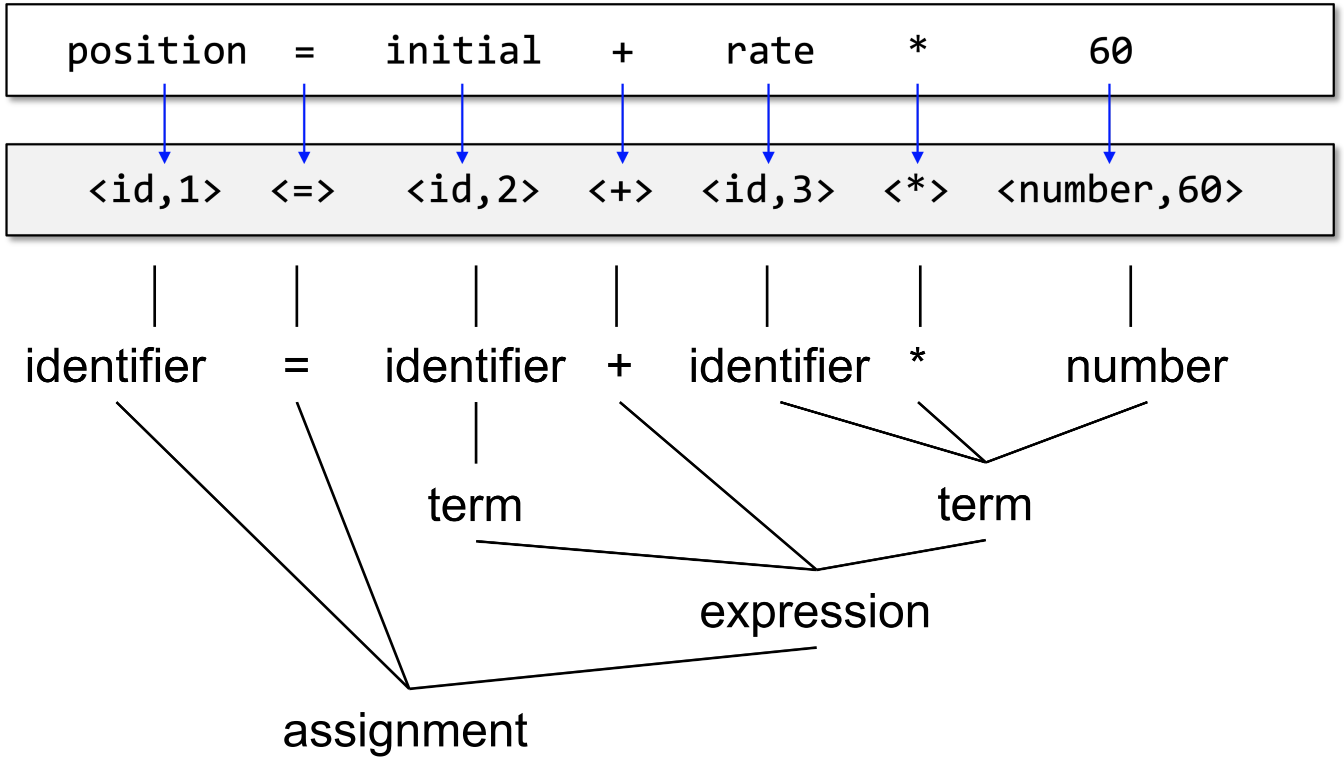
Chapter 4-2
Syntax Analysis

Syntax Analysis



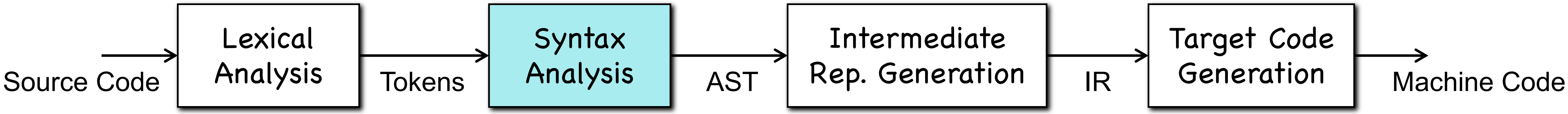

position = initial + rate * 60
<id,1> <=> <id,2> <+> <id,3> <*> <number,60>
identifier
=
identifier
+
identifier
*
number
term
term
expression
assignment

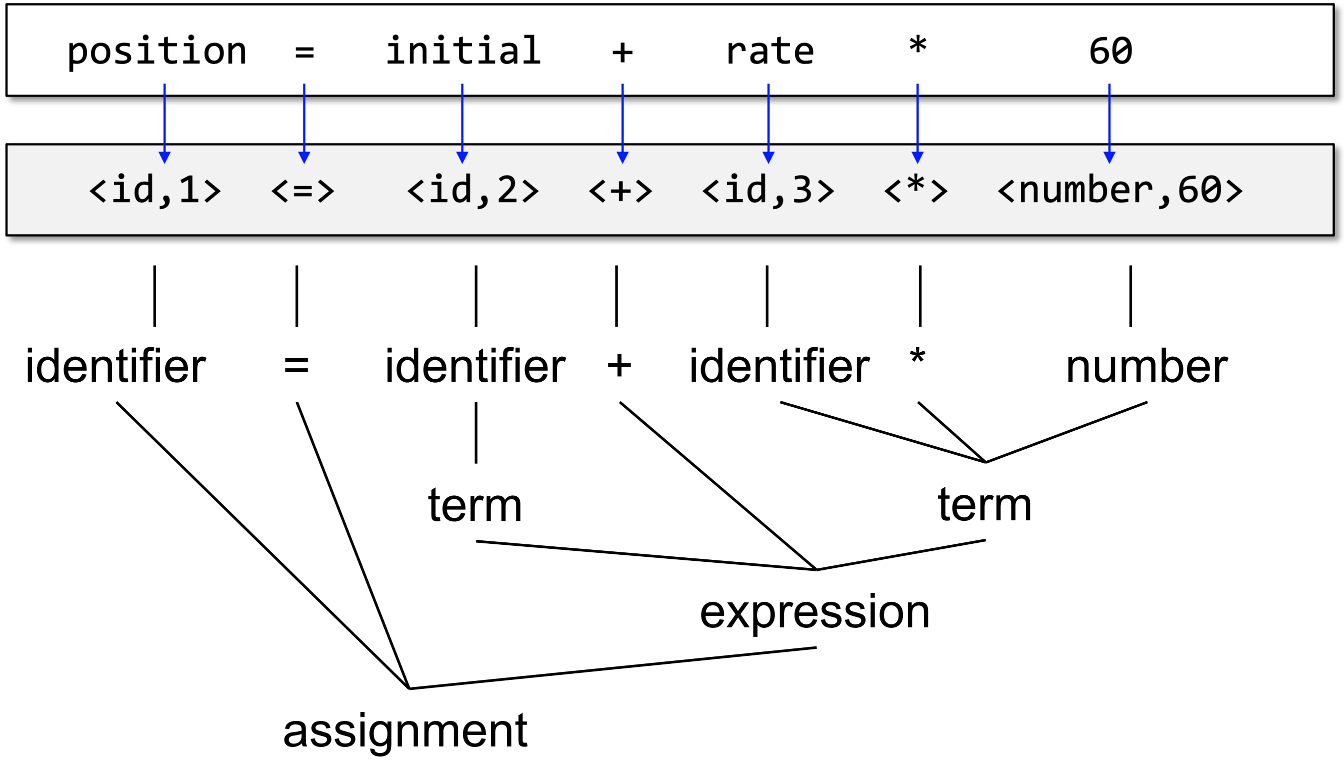
assignment → identifier = expression

expression → term + term

term → identifier
| identifier * number

Syntax Analysis




position = initial + rate * 60
<id,1> <=> <id,2> <+> <id,3> <*> <number,60>
identifier
=
identifier
+
identifier
*
number
term
term
expression
assignment

- A procedure of building the parse or syntax tree
  - Top-down parsing
  - Bottom-up parsing

```
assignment → identifier = expression
expression → term + term
      term → identifier
           | identifier * number
```

PART I: Top-Down Parsing

Top-Down Parsing

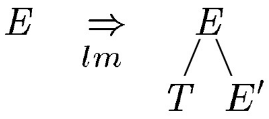
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

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

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

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

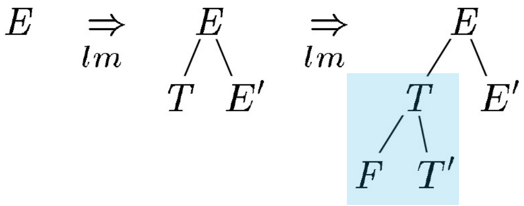


$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

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

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

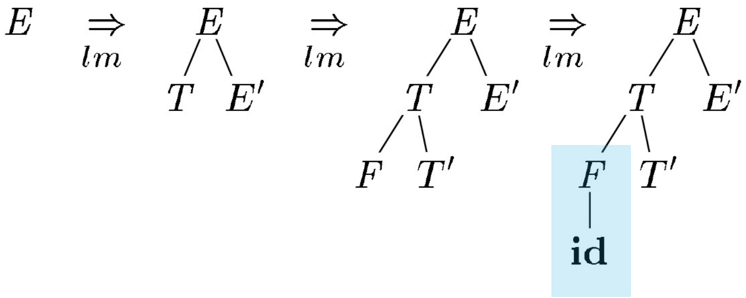


$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

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

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

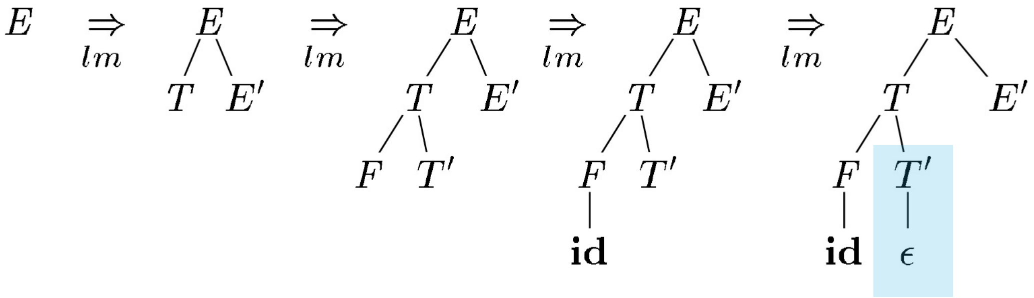


$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

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

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

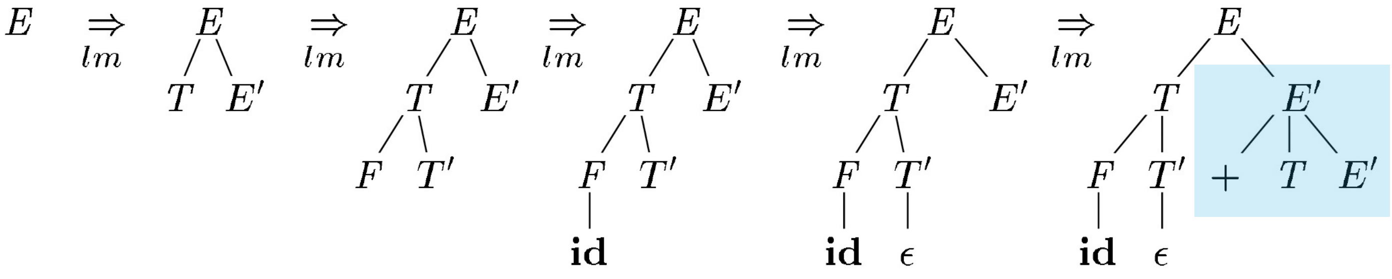


$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

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

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

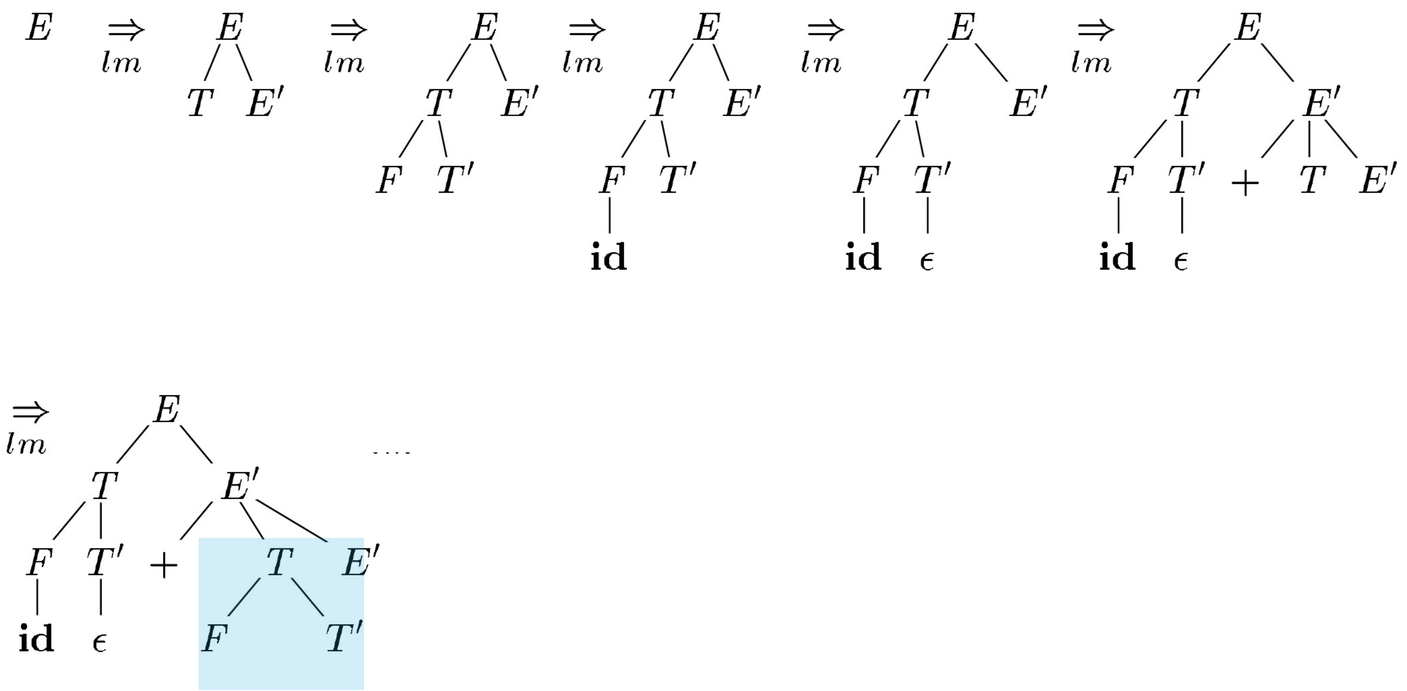


$$\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ \mid\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ \mid\ \epsilon \\
F & \rightarrow & (\ E\ )\ \mid\ \mathbf{id}
\end{array}$$

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

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

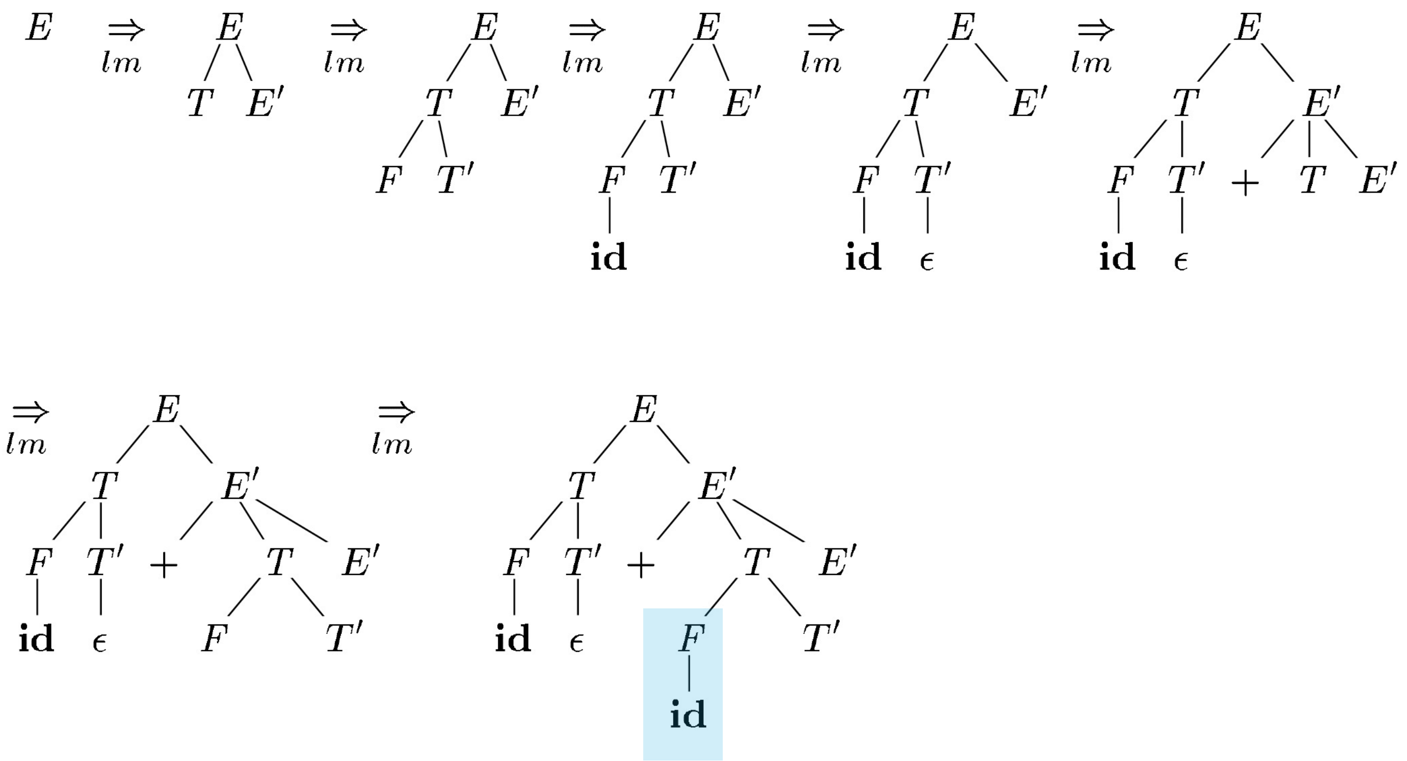


$$\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \textbf{id}
\end{array}$$

$\textbf{id} + \textbf{id} * \textbf{id}$

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**



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

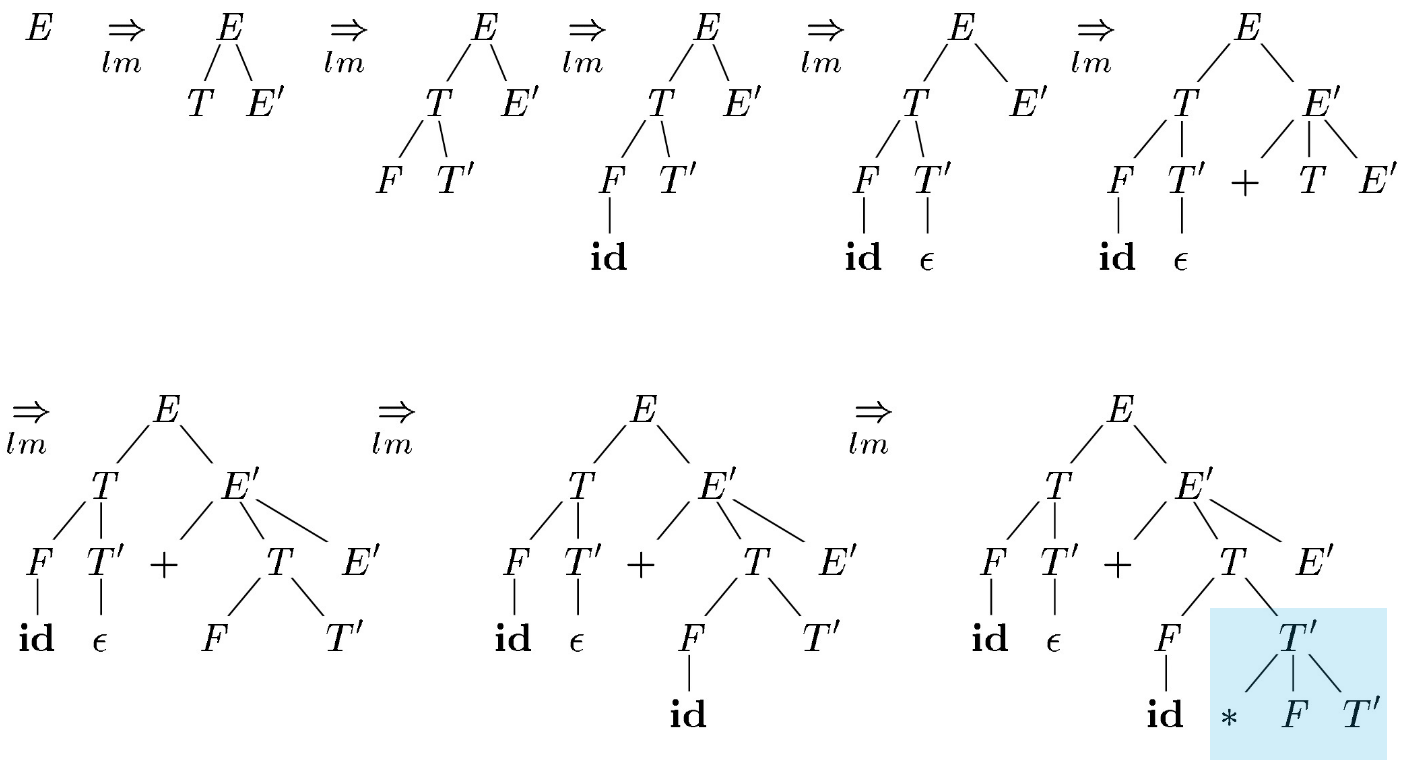
$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

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

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

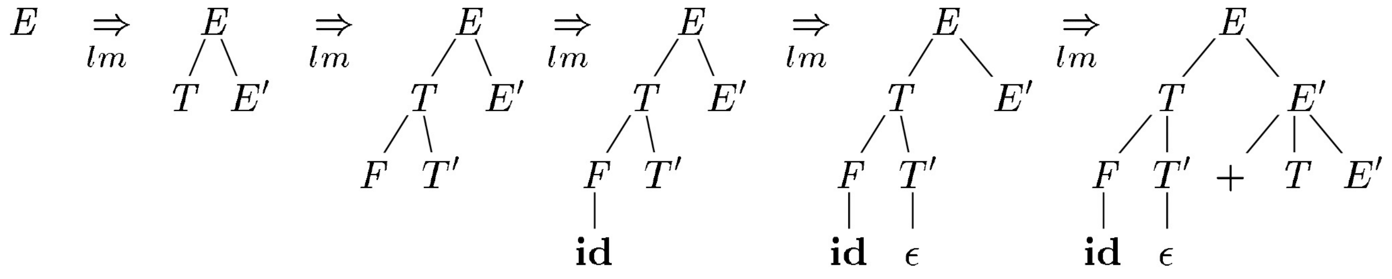


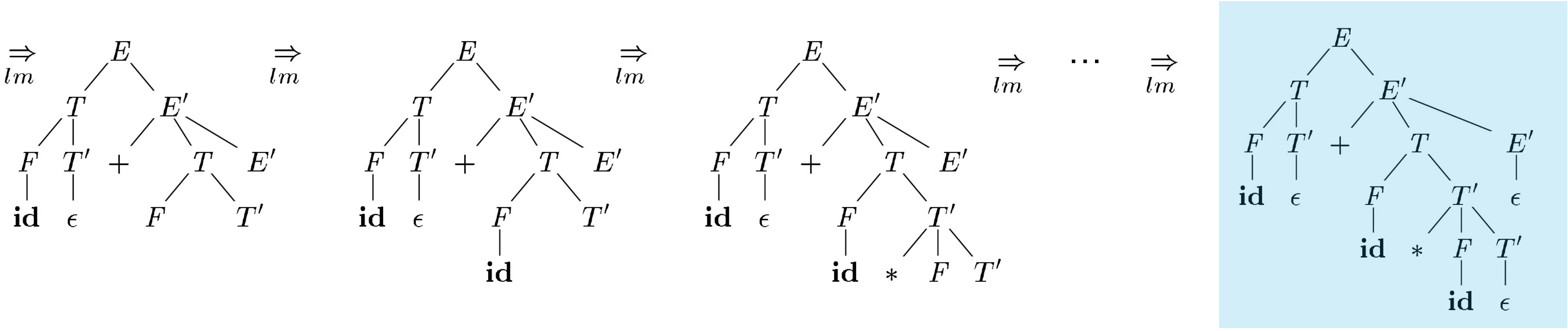
Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**

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

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




⇒ 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 ··· ⇒ lm
E T E' F T' + T E' id ϵ F T' ϵ id * F T' id ϵ

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow c\ A\ b;\ \ A \rightarrow a\ b\ |\ a$

```
bool A() {








}
```

```
bool S() {






}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow c\ A\ b;\ \ A \rightarrow a\ b \mid a$

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

or function for parsing

```
TreeNode* S() {
    TreeNode *S = new TreeNode;
    if (*cursor == ‘c’) { cursor++; S.addChildNode(‘c’); }
    else return null;

    if (TreeNode *A = A()) { S.addChildNode(A); }
    else return null;

    if (*cursor == ‘b’) { cursor++; S.addChildNode(‘b’); }
    else return null;

    return S;
}
```

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow c\ A\ b;\ \ A \rightarrow a\ b\ |\ a$

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \to c\,A\,b;\;\; A \to a\,b \mid a$

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow c\ A\ b;\quad A \rightarrow a\ b \mid a$

Let’s parse

c	a	b

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$

Let’s parse

c	a	b

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow c\ A\ b;\quad A \rightarrow a\ b \mid a$

Let’s parse

c	a	b

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow c\ A\ b;\quad A \rightarrow a\ b \mid a$

Let’s parse

c	a	b

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;
    if (!A()) return false;
    if (*cursor == ‘b’) cursor++;
    else return false;
    return true;
}
```

???

```
bool A() {
    temp = cursor;
    cursor = temp;
    if (*cursor == ‘a’) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1**: Backtracking may be necessary
  - when one derivation does not work, we may try others

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1**: Backtracking may be necessary
  - when one derivation does not work, we may try others
- **Problem 2**: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again

Building a Parser in Practice

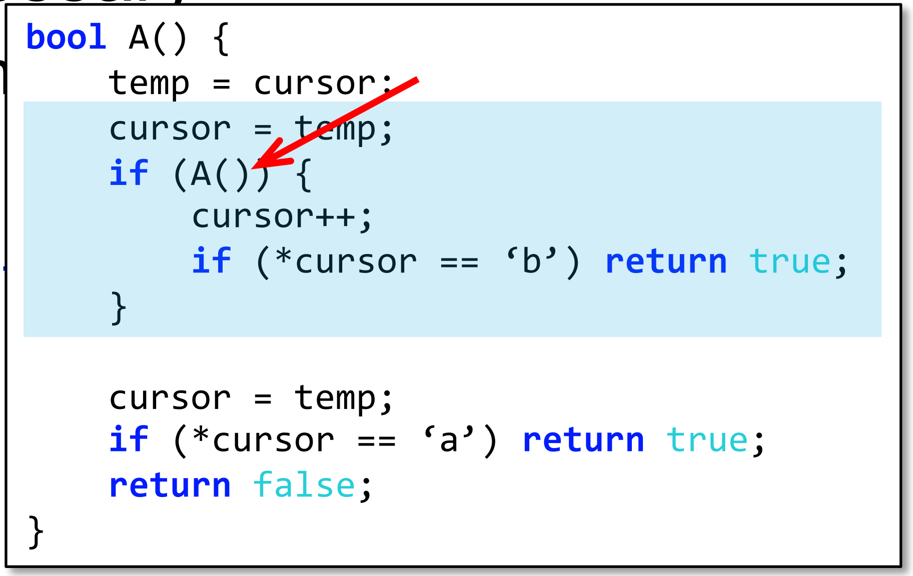
- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1**: Backtracking may be necessary
  - when one derivation does not work, we may try others
- **Problem 2**: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again
- **Example**: $A \to A\ b \mid a$

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1**: Backtracking may be necessary
  - when one derivation does not work, we m
- **Problem 2**: A **left-recursive** grammar
  - when expanding a non-terminal, we may
- **Example**: $A \rightarrow A\ b \mid a$



```
bool A() {
    temp = cursor;
    cursor = temp;
    if (A()) {
        cursor++;
        if (*cursor == ‘b’) return true;
    }

    cursor = temp;
    if (*cursor == ‘a’) return true;
    return false;
}
```

Eliminating Left-Recursion

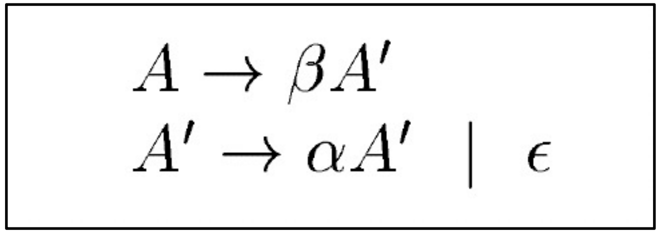
- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive, can be transformed into

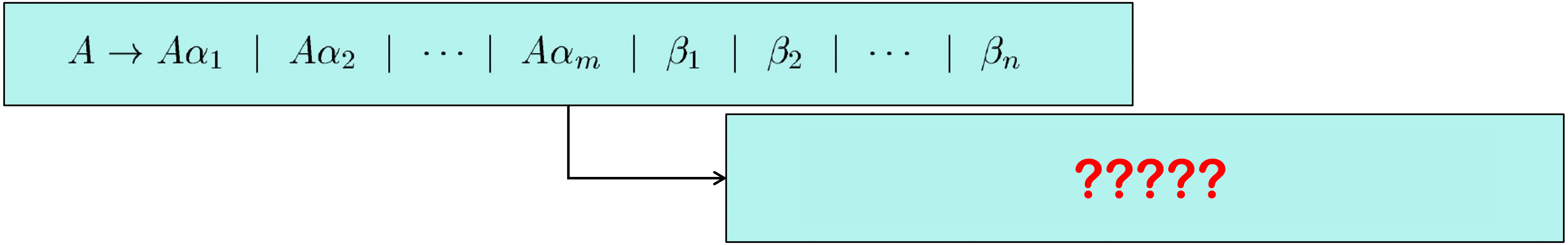


$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Eliminating Left-Recursion

- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive, can be transformed into

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$



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

?????

Eliminating Left-Recursion

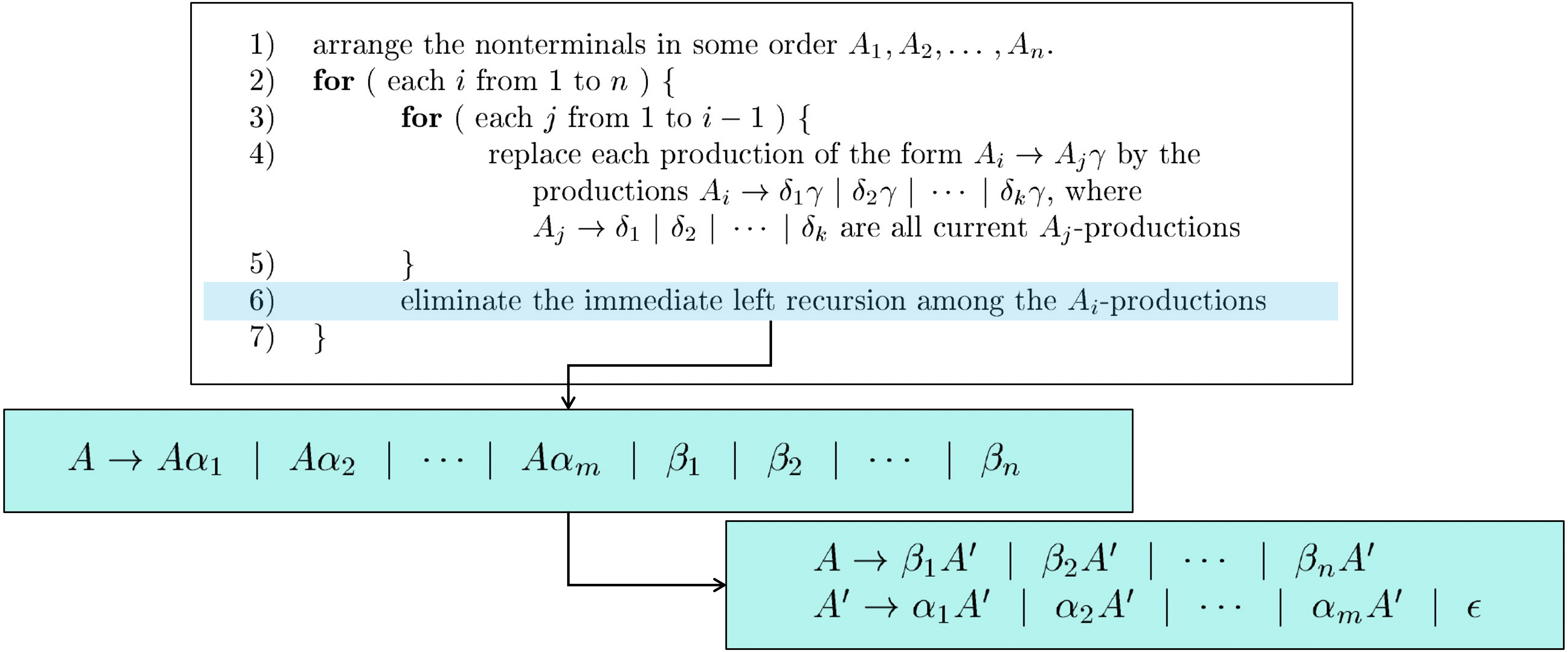
- A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$
- **Example:** $A \rightarrow A\alpha \mid \beta$ is left-recursive, can be transformed into

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

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

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Eliminating 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
}
```

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

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

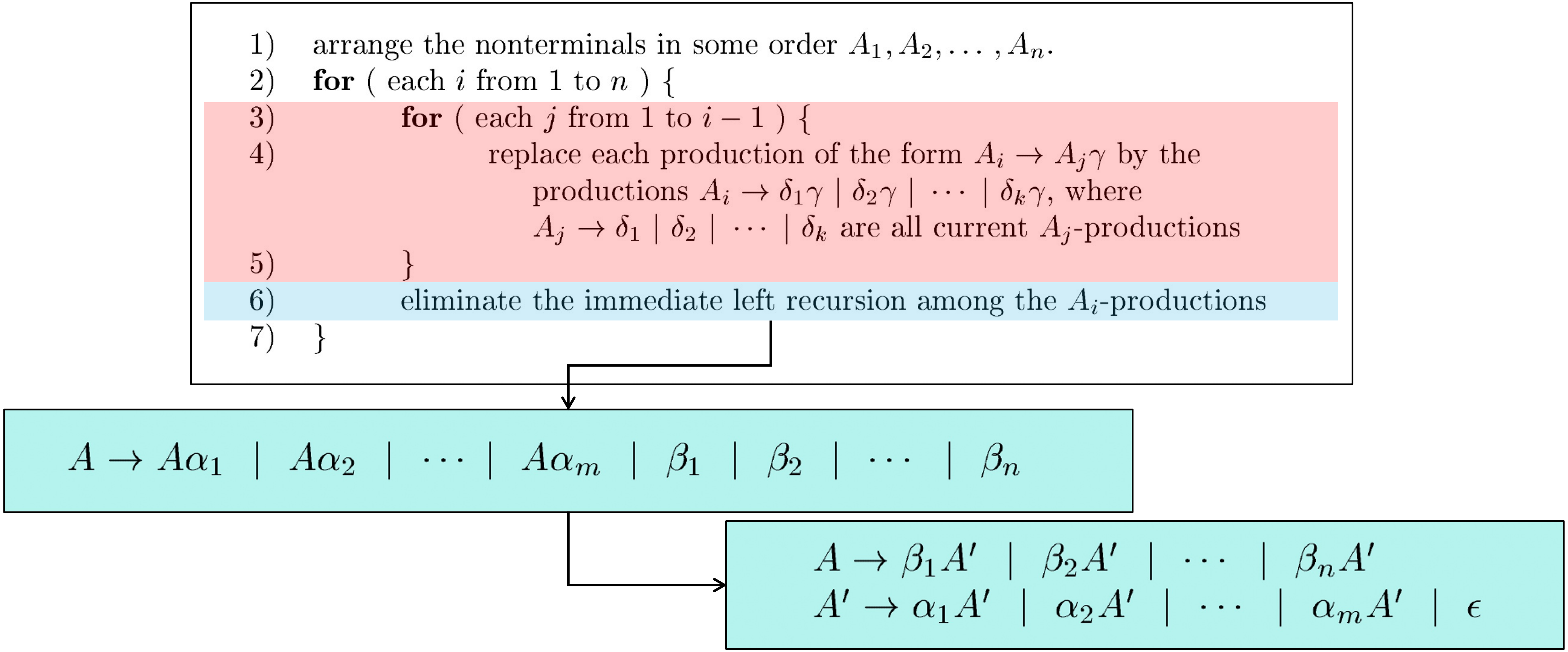
Eliminating Left-Recursion

```
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
}
```

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

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Eliminating Left-Recursion

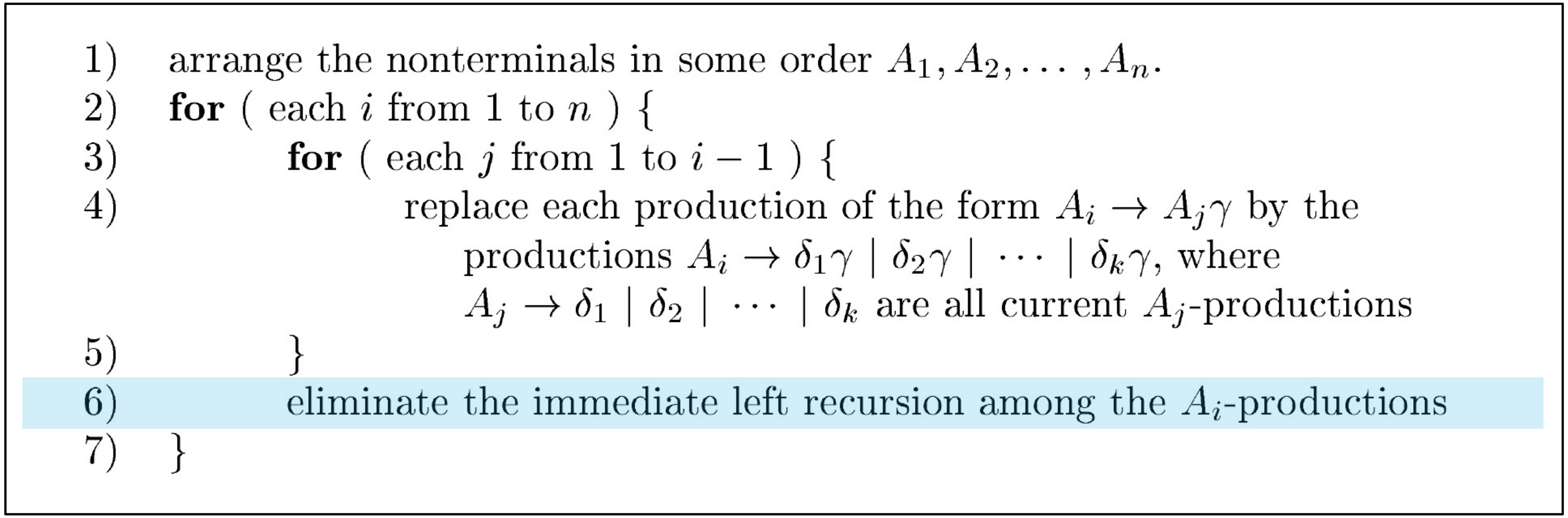


```
arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for ( each $i$ from 1 to $n$ ) {
        for ( each $j$ from 1 to $i-1$ ) {
                replace each production of the form $A_i \rightarrow A_j\gamma$ by the
                    productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
                    $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current $A_j$-productions
        }
        eliminate the immediate left recursion among the $A_i$-productions
}
```

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

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Eliminating Left-Recursion

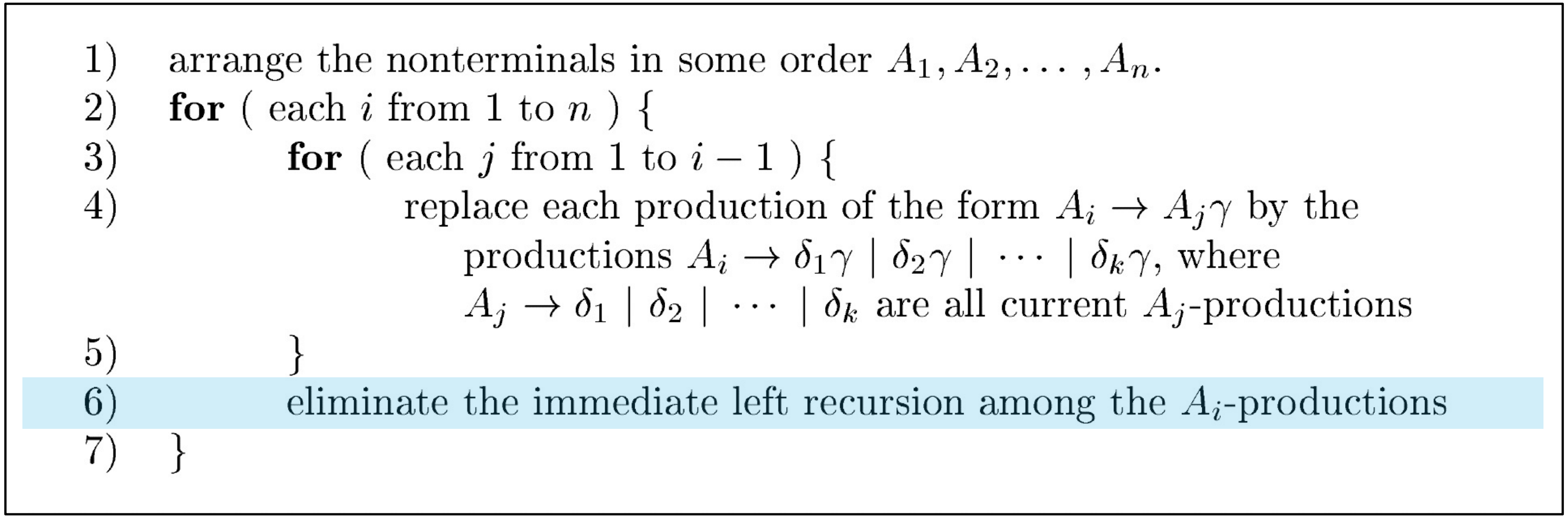


```
arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for ( each $i$ from 1 to $n$ ) {
    for ( each $j$ from 1 to $i-1$ ) {
        replace each production of the form $A_i \to A_j\gamma$ by the
          productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, 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
}
```

This algorithm is guaranteed to work if the input grammar does NOT include

(1) cycles ($A \Rightarrow^+ A$) or (2) ϵ productions

Eliminating 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
}
```

Any grammar can be converted to a grammar that does NOT include

(1) cycles ($A \Rightarrow^+ A$) or (2) ϵ productions*

** with possible exception of the empty string*

Eliminating Left-Recursion

- Example

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow Ac \mid Sd
\end{aligned}
$$

$$
\boxed{A \rightarrow A\alpha \mid \beta} \;\Rightarrow\; \boxed{\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}}
$$

Eliminating Left-Recursion

- Example

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

⇨

$$S \to Aa \mid b$$
$$A \to SdA';\ A' \to cA' \mid \epsilon$$

$$A \to A\alpha \mid \beta$$

⇨

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

Eliminating Left-Recursion

- Example

$$\boxed{\begin{array}{l} S \to Aa \mid b \\ A \to Ac \mid Sd \end{array}} \Rightarrow \boxed{\begin{array}{l} S \to Aa \mid b \\ A \to SdA';\ A' \to cA' \mid \epsilon \end{array}} \Rightarrow \boxed{\begin{array}{l} S \to SdA'a \mid b \\ A \to SdA';\ A' \to cA' \mid \epsilon \end{array}}$$

$$\boxed{A \to A\alpha \mid \beta} \Rightarrow \boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}}$$

Eliminating Left-Recursion

- Example

$$\boxed{\begin{array}{l} S \to Aa \mid b \\ A \to Ac \mid Sd \end{array}} \Rightarrow \boxed{\begin{array}{l} S \to Aa \mid b \\ A \to SdA';\ A' \to cA' \mid \epsilon \end{array}} \Rightarrow \boxed{\begin{array}{l} S \to SdA'a \mid b \\ A \to SdA';\ \ A' \to cA' \mid \epsilon \end{array}} \Rightarrow \boxed{\begin{array}{l} S \to bS';\ \ S' \to dA'aS' \mid \epsilon \\ A \to SdA';\ A' \to cA' \mid \epsilon \end{array}}$$

$$\boxed{A \to A\alpha \mid \beta} \Rightarrow \boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}}$$

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Problem 1**: Backtracking may be necessary
  - when one derivation does not work, we may try others
- ✓ **Problem 2**: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again

Predictive Parsing

- Predictive parsers are recursive descent parser w/o backtracking

Predictive Parsing

- Predictive parsers are recursive descent parser w/o backtracking
- LL(1)
  - L: scanning input from left to right
  - L: leftmost derivation
  - 1: Using one input symbol of lookahead at each step

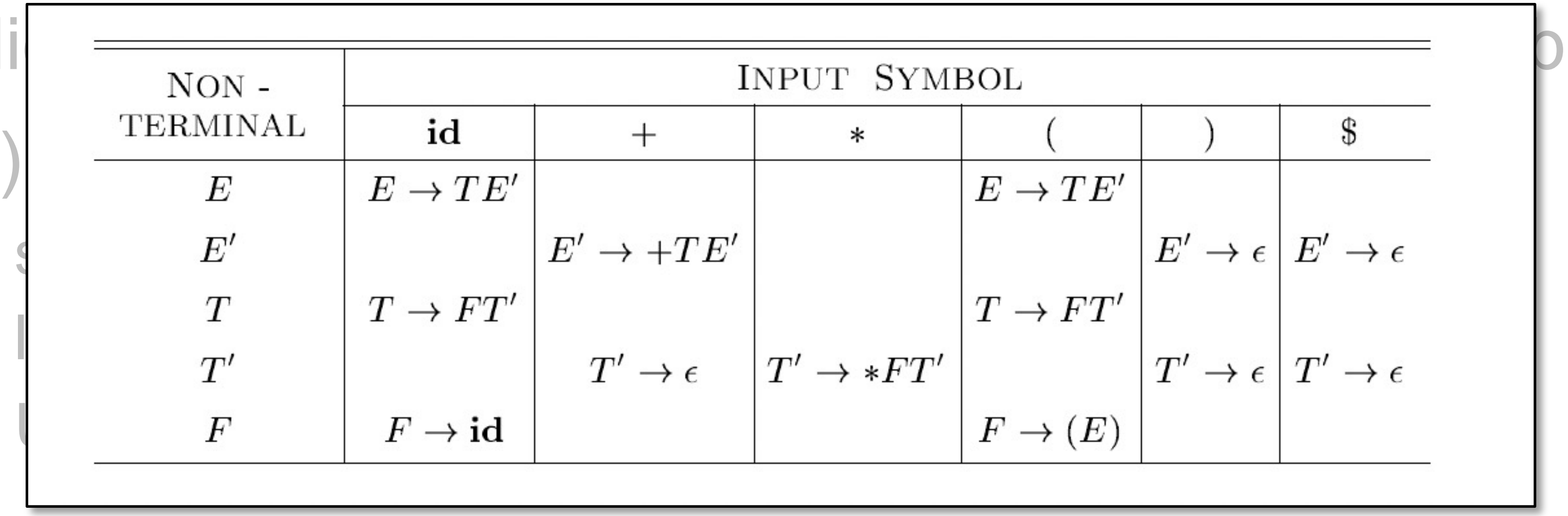
Predictive Parsing

- Predictive parsers are recursive descent parser w/o backtracking
- LL(1)
  - L: scanning input from left to right
  - L: leftmost derivation
  - 1: Using one input symbol of lookahead at each step

- LL(1) grammar (Not ambiguous! Not left-recursive!)
  - Rich enough to cover most programming constructs

Predictive Parsing

- Predi backtracking
- LL(1)
  - L: s
  - L: l
  - 1: U



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

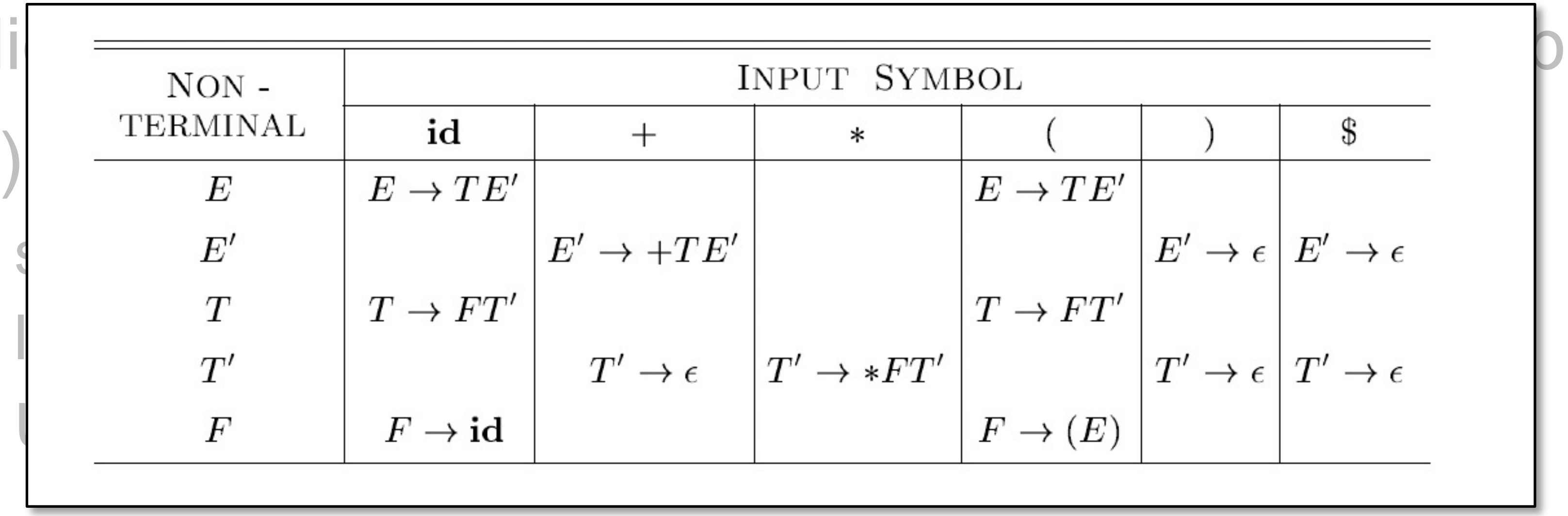
- LL(1) grammar (Not ambiguous! Not left-recursive!)
  - Rich enough to cover most programming constructs

Predictive Parsing

- Predi

backtracking

- LL(1)
  - L: s
  - L: l
  - 1: U



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

- LL(1) grammar (Not ambiguous! Not left-recursive!)
  - Rich enough to cover most programming constructs
  - To build the predictive table, let's define $\mathrm{FIRST}(\alpha)$; $\mathrm{FOLLOW}(\alpha)$

First() and Follow()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with

First() and Follow()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with
- $\mathrm{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α

First()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with
  - If X is a terminal, $\mathrm{FIRST}(X) = \{X\}$

First()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with
  - If X is a terminal, $\mathrm{FIRST}(X) = \{X\}$
  - If $X \to \epsilon$ is a production, $\epsilon \in \mathrm{FIRST}(X)$

First()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with

  - If X is a terminal, $\mathrm{FIRST}(X) = \{X\}$
  - If $X \to \epsilon$ is a production, $\epsilon \in \mathrm{FIRST}(X)$
  - If $X \to Y_1 Y_2 \ldots Y_k$, $\epsilon \in \bigcap_{j=1}^{i-1} \mathrm{FIRST}(Y_j) \land a \in \mathrm{FIRST}(Y_i) \Rightarrow a \in \mathrm{FIRST}(X)$

$X \to Y_1 Y_2 \ldots Y_{i-1} Y_i \ldots Y_k \to Y_i \ldots Y_k$

First()

- $\mathrm{FIRST}(\alpha)$:
  - A set of terminals that α may start with
  - If X is a terminal, $\mathrm{FIRST}(X) = \{X\}$
  - If $X \to \epsilon$ is a production, $\epsilon \in \mathrm{FIRST}(X)$
  - If $X \to Y_1 Y_2 \ldots Y_k$, $\epsilon \in \bigcap_{j=1}^{i-1} \mathrm{FIRST}(Y_j) \land a \in \mathrm{FIRST}(Y_i) \Rightarrow a \in \mathrm{FIRST}(X)$

$X \to Y_1 Y_2 \ldots Y_{i-1} Y_i \ldots Y_k \to Y_i \ldots Y_k$

First()

- $\text{FIRST}(\alpha)$:
  - A set of terminals that α may start with

  - If X is a terminal, $\text{FIRST}(X) = \{X\}$
  - If $X \rightarrow \epsilon$ is a production, $\epsilon \in \text{FIRST}(X)$
  - If $X \rightarrow Y_1 Y_2 \ldots Y_k$, $\epsilon \in \bigcap_{j=1}^{i-1} \text{FIRST}(Y_j) \wedge a \in \text{FIRST}(Y_i) \Rightarrow a \in \text{FIRST}(X)$

$$\epsilon \in \bigcap_{j=1}^{k} \text{FIRST}(Y_j) \Rightarrow \epsilon \in \text{FIRST}(X)$$

$$X \rightarrow Y_1 Y_2 \ldots Y_k \rightarrow \epsilon$$

First()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\text{FIRST}(S) = \{c\}$
- $\text{FIRST}(A) = \{a\}$
- $\text{FIRST}(a) = \{a\}$
- $\text{FIRST}(b) = \{b\}$
- $\text{FIRST}(c) = \{c\}$

First()

- **Exercise**: write First() for all symbols in the following grammar

E → T X

X → + E

X → ε

T → int Y

T → (E)

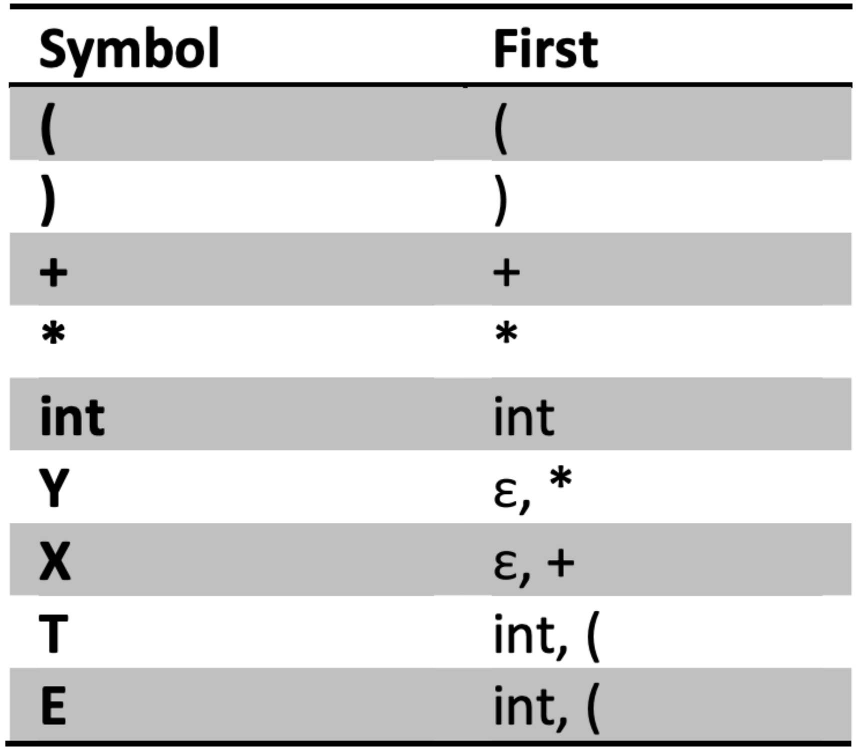
Y → * T

Y → ε

First()

- **Exercise**: write First() for all symbols in the following grammar

E → T X
X → + E
X → ε
T → int Y
T → (E)
Y → * T
Y → ε



Symbol	First
(	(
)	)
+	+
*	*
int	int
Y	ε, *
X	ε, +
T	int, (
E	int, (

Follow()

- $\mathrm{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α

Follow()

- $\text{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α
  - $\$ \in \text{FOLLOW}(S)$, where $\$$ is string's end marker, S the start non-terminal

Follow()

- $\mathrm{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α

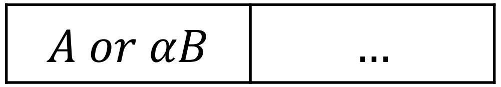
  - $\$ \in \mathrm{FOLLOW}(S)$, where $\$$ is string's end marker, S the start non-terminal
  - $A \rightarrow \alpha B \beta \Rightarrow \mathrm{FIRST}(\beta) \backslash \{\epsilon\} \subseteq \mathrm{FOLLOW}(B)$

αB	β

Follow()

- $\text{FOLLOW}(\alpha)$:
  - A set of terminals that can appear immediately to the right of α

  - $\$ \in \text{FOLLOW}(S)$, where $\$$ is string's end marker, S the start non-terminal
  - $A \to \alpha B \beta \Rightarrow \text{FIRST}(\beta) \backslash \{\epsilon\} \subseteq \text{FOLLOW}(B)$
  - $A \to \alpha B$ or $A \to \alpha B \beta$ where $\epsilon \in \text{FIRST}(\beta) \Rightarrow \text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$



A or αB

…

Follow()

- FOLLOW(α):
  - A set of terminals that can appear immediately to the right of α

  - $\$ \in \mathrm{FOLLOW}(S)$, where $\$$ is string's end marker, S the start non-terminal
  - $A \to \alpha B\beta \Rightarrow \mathrm{FIRST}(\beta)\backslash\{\epsilon\} \subseteq \mathrm{FOLLOW}(B)$
  - $A \to \alpha B$ or $A \to \alpha B\beta$ where $\epsilon \in \mathrm{FIRST}(\beta) \Rightarrow \mathrm{FOLLOW}(A) \subseteq \mathrm{FOLLOW}(B)$

A or αB	…

- **Note: repeat the procedure until fixed point!**

Follow()

- **Example**: $S \rightarrow c\ A\ b;\ \ A \rightarrow a\ b \mid a$
- $\mathrm{FOLLOW}(S) = \{\$\}$
- $\mathrm{FOLLOW}(A) = \{b\}$

Follow()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) = \{b\}$

- **Example**: $S \to c\ A\ A;\ \ A \to a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$

Follow()

- **Example**: $S \rightarrow c\ A\ b;\ \ A \rightarrow a\ b \mid a$
- $\mathrm{FOLLOW}(S) = \{\$\}$
- $\mathrm{FOLLOW}(A) = \{b\}$

- **Example**: $S \rightarrow c\ A\ A;\ \ A \rightarrow a\ b \mid a$
- $\mathrm{FOLLOW}(S) = \{\$\}$
- $\mathrm{FOLLOW}(A) \supseteq \mathrm{FIRST}(A) \backslash \{\epsilon\} = \{a\}$

Follow()

- **Example**: $S \to c\ A\ b;\ \ A \to a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) = \{b\}$

- **Example**: $S \to c\ A\ A;\ \ A \to a\ b \mid a$
- $\text{FOLLOW}(S) = \{\$\}$
- $\text{FOLLOW}(A) \supseteq \text{FIRST}(A) \backslash \{\epsilon\} = \{a\}$
- $\text{FOLLOW}(A) \supseteq \text{FOLLOW}(S) = \{\$\}$

Follow()

- **Exercise**: write Follow() for all symbols in the grammar

E → T X
X → + E
X → ε
T → int Y
T → (E)
Y → * T
Y → ε

Symbol	First
(	(
)	)
+	+
*	*
int	int
Y	ε, *
X	ε, +
T	int, (
E	int, (

Follow()

- **Exercise**: write Follow() for all symbols in the grammar

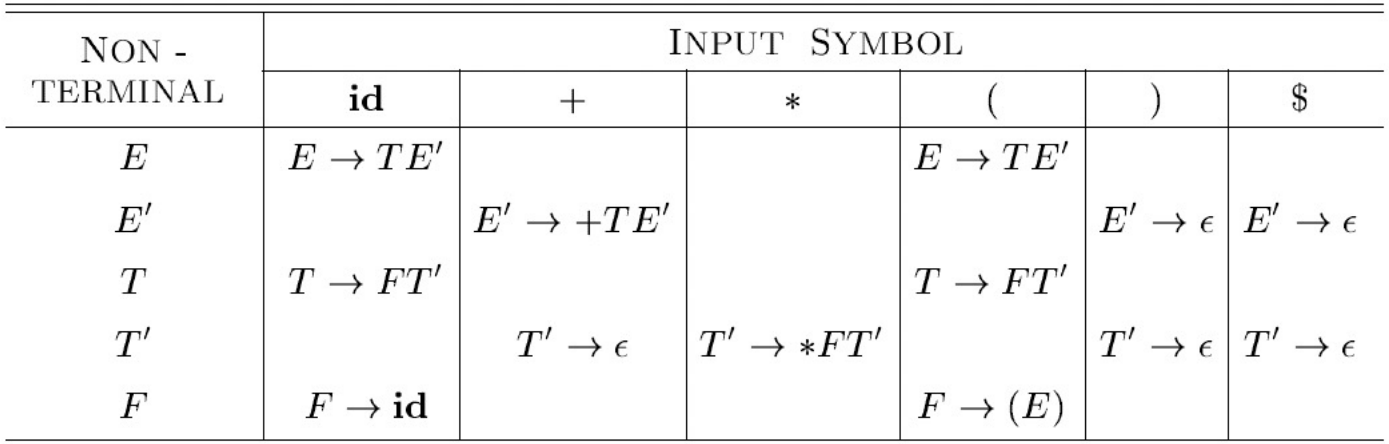
E → T X
X → + E
X → ε
T → int Y
T → (E)
Y → * T
Y → ε

Symbol	First	Follow
(	(	N/A
)	)	
+	+	
*	*	
int	int	
Y	ε, *	), $, +
X	ε, +	), $
T	int, (	), $, +
E	int, (	), $

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \to \alpha$
  - $\forall a \in \text{FIRST}(\alpha)$: $M[A, a] = A \to \alpha$
  - $\epsilon \in \text{FIRST}(\alpha) \Rightarrow \ \forall b \in \text{FOLLOW}(A)$: $M[A, b] = A \to \alpha$

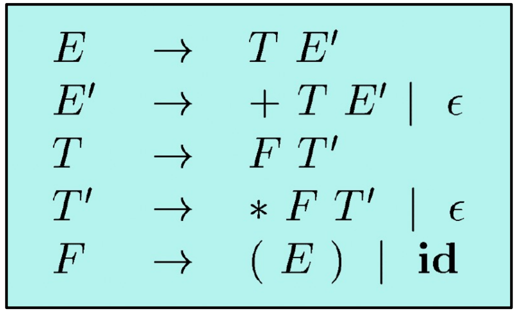
$$\begin{array}{lcl} E & \to & T\ E' \\ E' & \to & +\ T\ E' \ \mid\ \epsilon \\ T & \to & F\ T' \\ T' & \to & *\ F\ T' \ \mid\ \epsilon \\ F & \to & (\ E\ )\ \mid\ \textbf{id} \end{array}$$



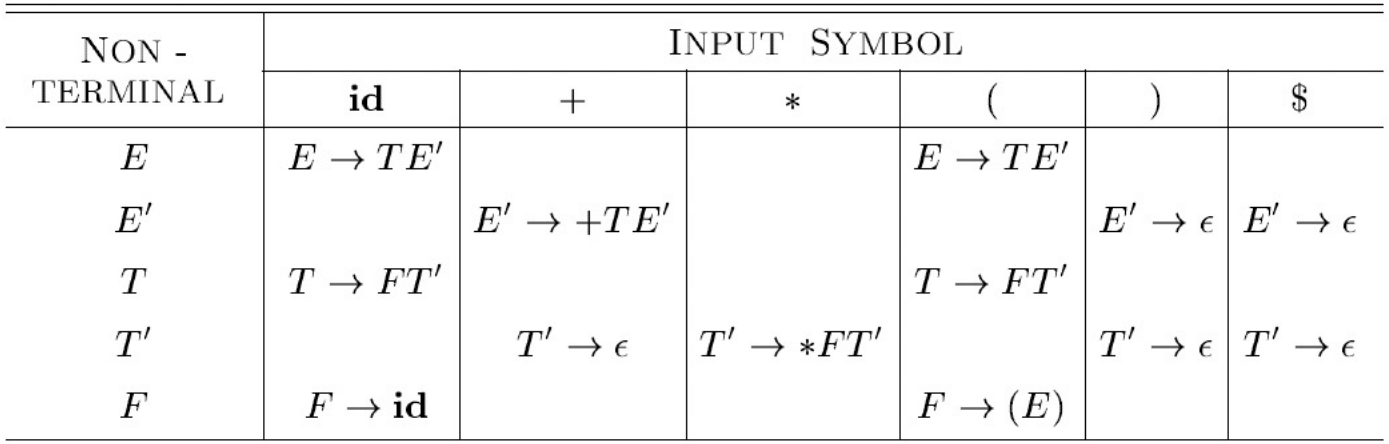
NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \rightarrow \alpha$
  - **$\forall a \in \mathbf{FIRST}(\alpha)$: $M[A, a] = A \rightarrow \alpha$**
  - $\epsilon \in \mathrm{FIRST}(\alpha) \Rightarrow \ \forall b \in \mathrm{FOLLOW}(A)$: $M[A, b] = A \rightarrow \alpha$



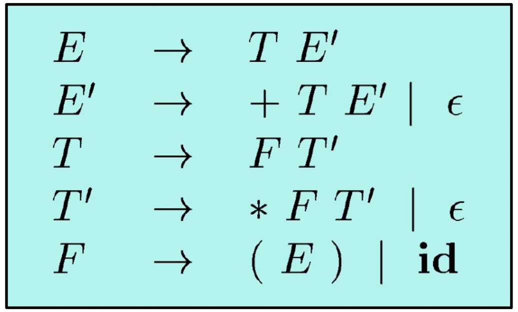
$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$



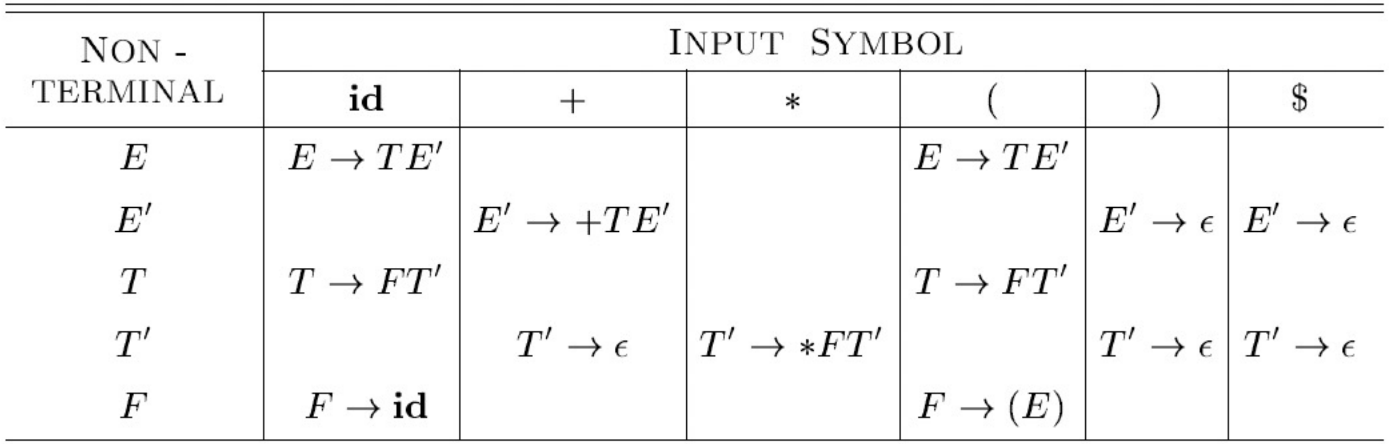
NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$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

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \rightarrow \alpha$
  - $\forall a \in \text{FIRST}(\alpha)$: $M[A, a] = A \rightarrow \alpha$
  - $\boldsymbol{\epsilon \in \text{FIRST}(\alpha) \Rightarrow \ \forall b \in \text{FOLLOW}(A): M[A, b] = A \rightarrow \alpha}$



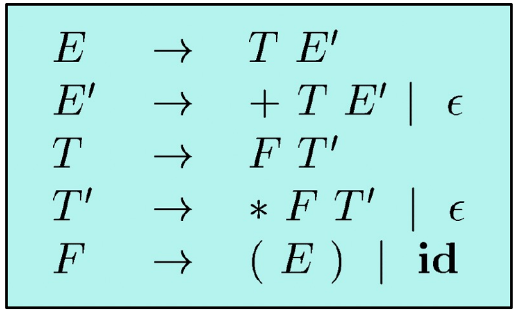
$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \textbf{id} \end{array}$$



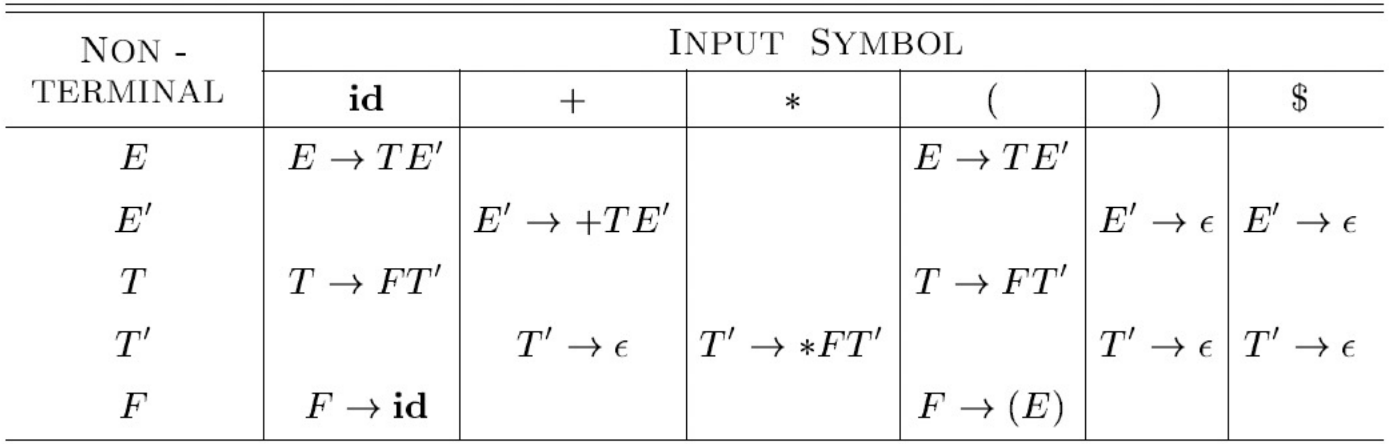
NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T'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

Predictive Parsing Table

- To build a parsing table $M[A, a]$, for each $A \to \alpha$
  - $\forall a \in \text{FIRST}(\alpha)$: $M[A, a] = A \to \alpha$
  - $\epsilon \in \text{FIRST}(\alpha) \Rightarrow \ \forall b \in \text{FOLLOW}(A)$: $M[A, b] = A \to \alpha$



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



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	$F \to \textbf{id}$			$F \to (E)$		

- **Choose the production according to the table, empty means error**

Building a Parser in Practice

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- ✓ **Problem 1**: Backtracking may be necessary
  - when one derivation does not work, we may try others
- ✓ **Problem 2**: A **left-recursive** grammar can cause **infinite loops**
  - when expanding a non-terminal, we may find itself and expand it again

LL(1) Grammar: Formal Definition

- LL(1) grammar (Not ambiguous! Not left-recursive!)
- A grammar is LL(1) if and only if any $A \rightarrow \alpha \mid \beta$ satisfies:
    - (1) For no terminal a do both α and β derive strings starting with a (by left factoring)
    - (2) At most one of α and β derive the empty string
    - (3) If $\beta \Rightarrow^* \epsilon$, α doesn't derive strings starting with terminals in $\mathrm{FOLLOW}(\alpha)$

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

Recursive Predictive Parsing

- Each non-terminal has a procedure or function for parsing
- => Recursive-Descent Parser

- **Example**: $S \rightarrow a\ S\ b \mid \epsilon$

```
bool S() {
    if (*cursor == ‘c’) cursor++;
    else return false;

    if (!S()) return false;

    if (*cursor == ‘b’) cursor++;
    else return false;

    return true;
}
```

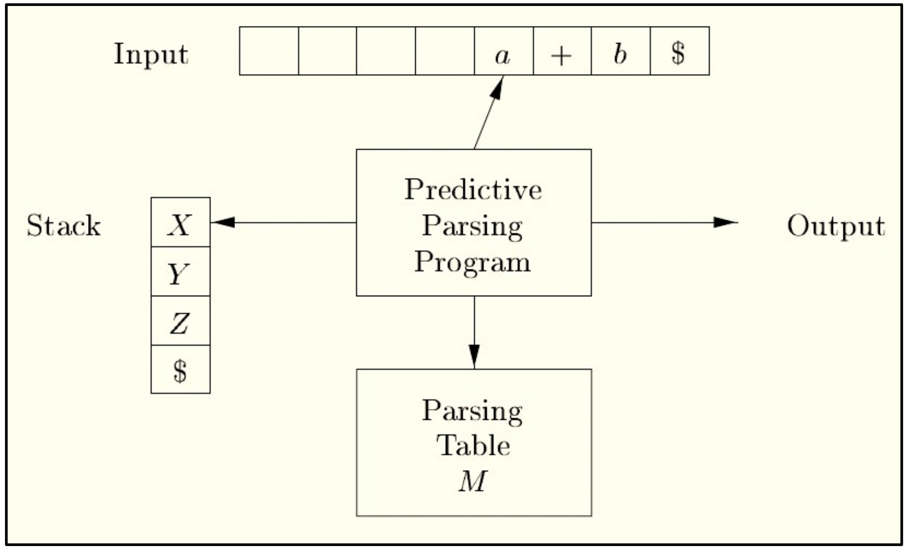
Non-recursive Predictive Parsing

Non-recursive Predictive Parsing

- How can we build a predictive parser without recursion?
- Maintain a stack explicitly!

Non-recursive Predictive Parsing

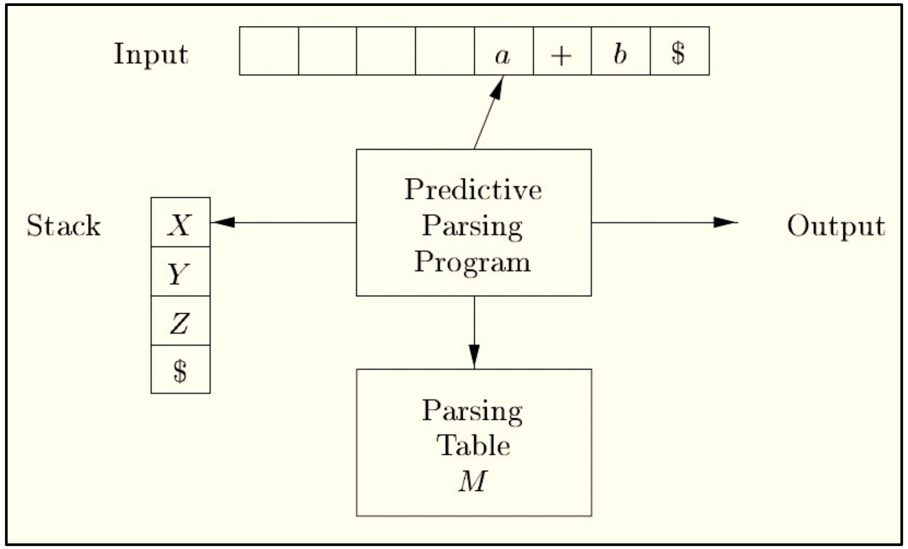
- How can we build a predictive parser without recursion?
- Maintain a stack explicitly!



Non-recursive Predictive Parsing

- How can we build a predictive parser without recursion?
- Maintain a stack explicitly!

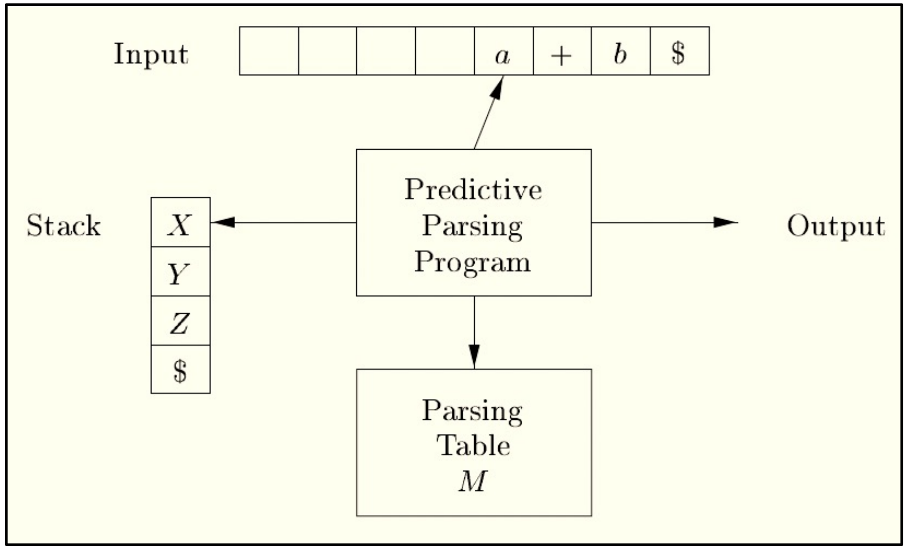
- Initially, we put S in the stack
- When $A \rightarrow \alpha$ is applied, pop A, push α



Non-recursive Predictive Parsing

- How can we build a predictive parser without recursion?
- Maintain a stack explicitly!

- Initially, we put S in the stack
- When $A \to \alpha$ is applied, pop A, push α



- Building the PDA from CFG!
- Parsing table is the set of transition functions of PDA!

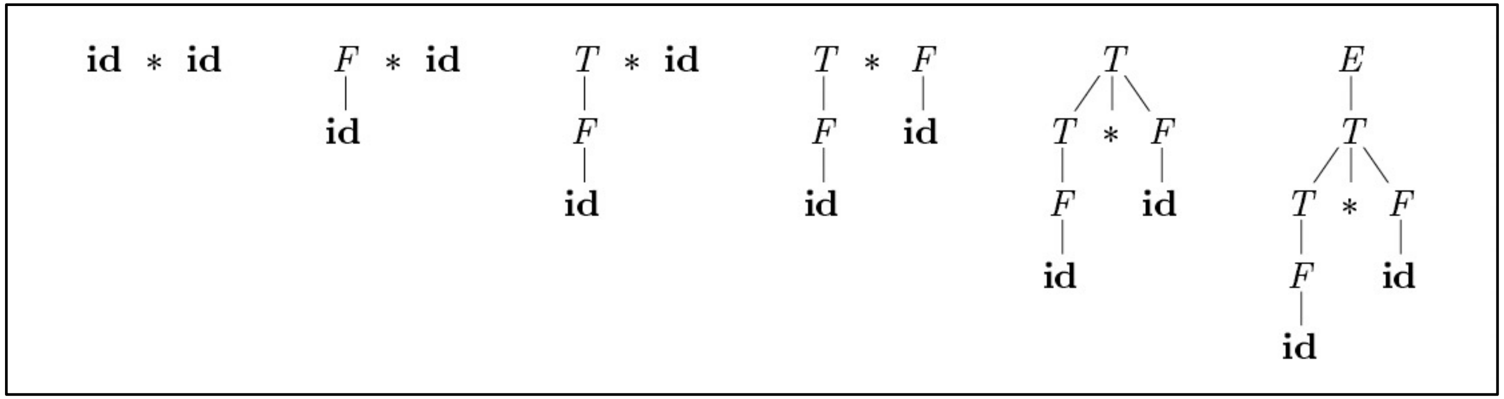
PART II: Bottom-Up Parsing

Bottom-up Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**

Bottom-up Parsing

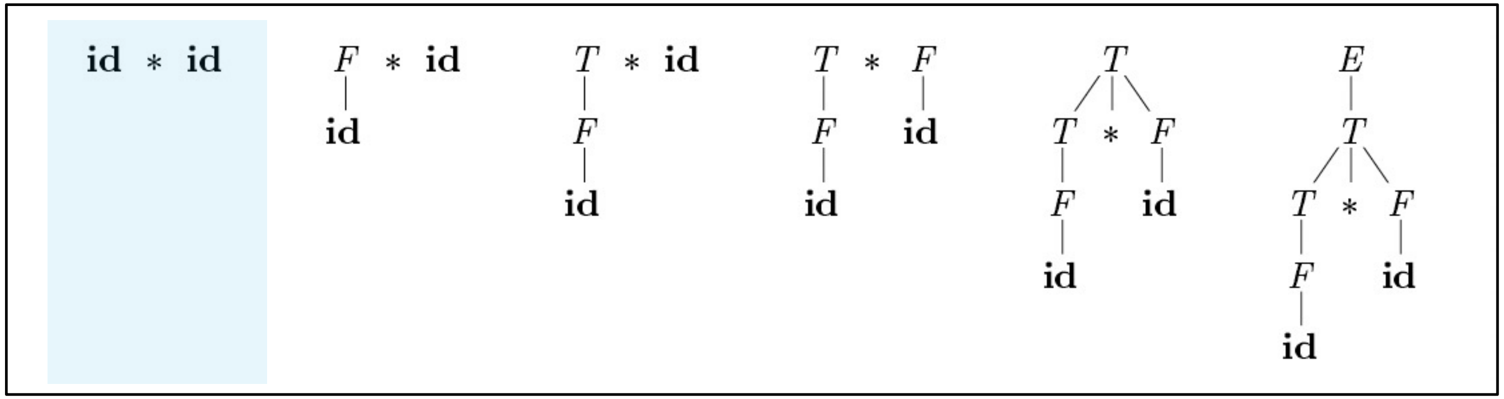
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

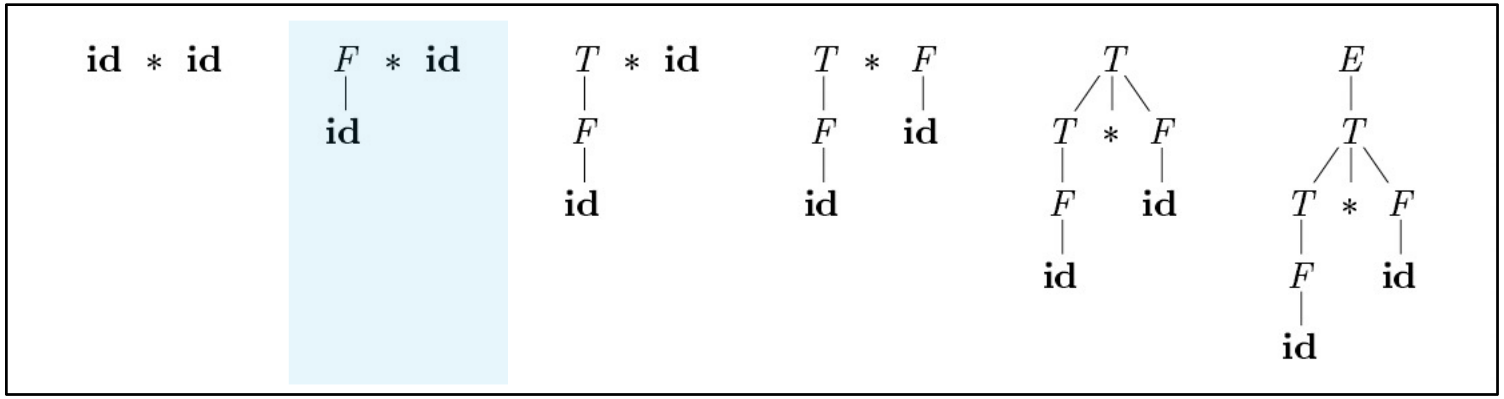
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

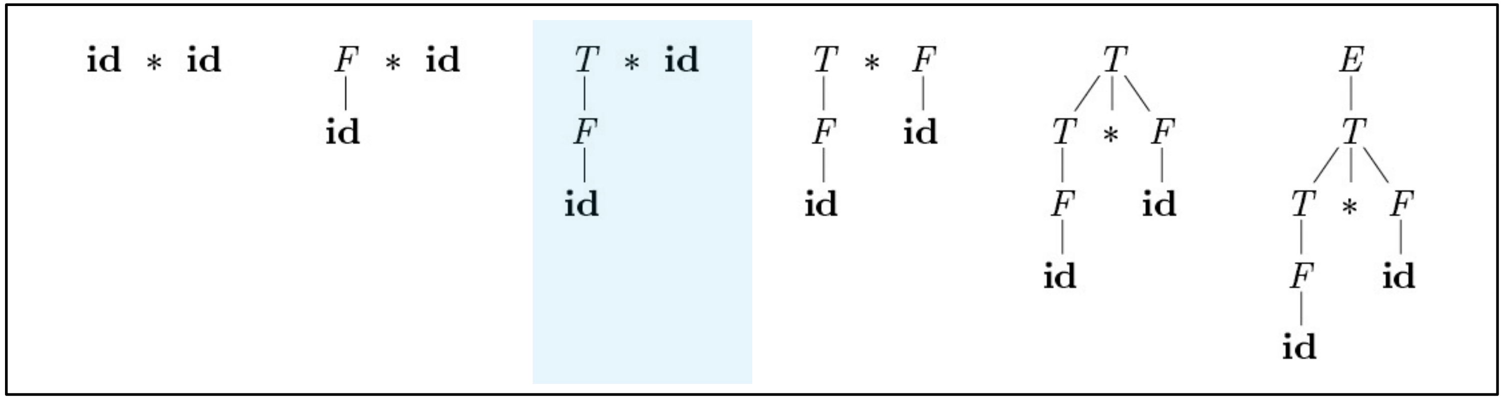
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

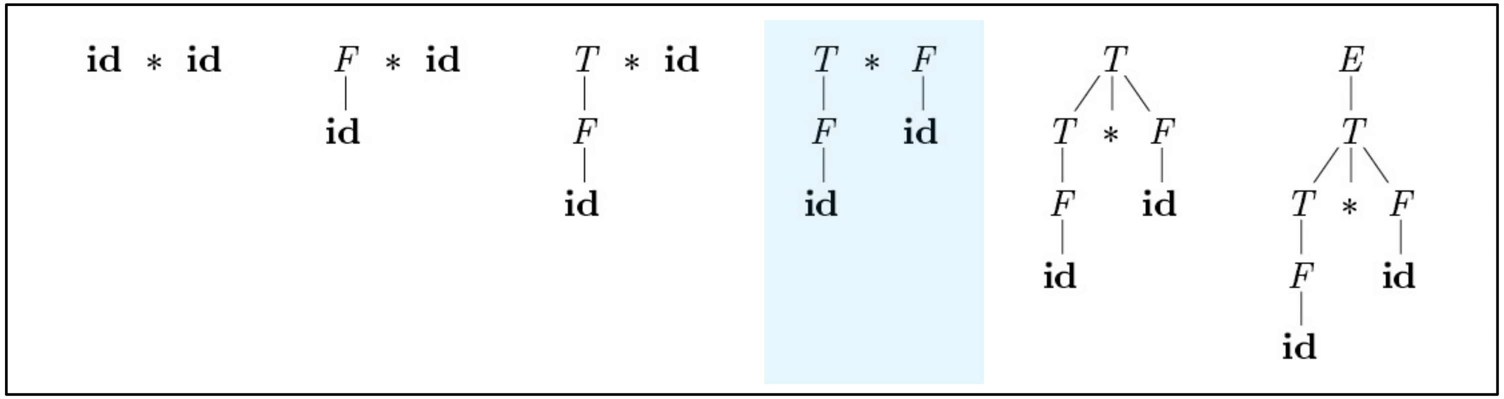
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

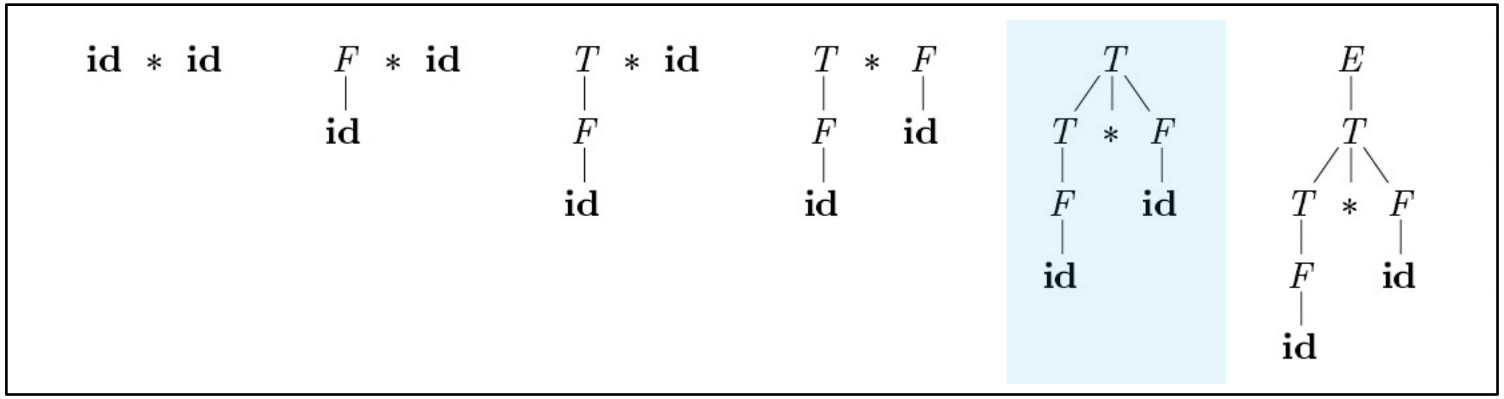
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

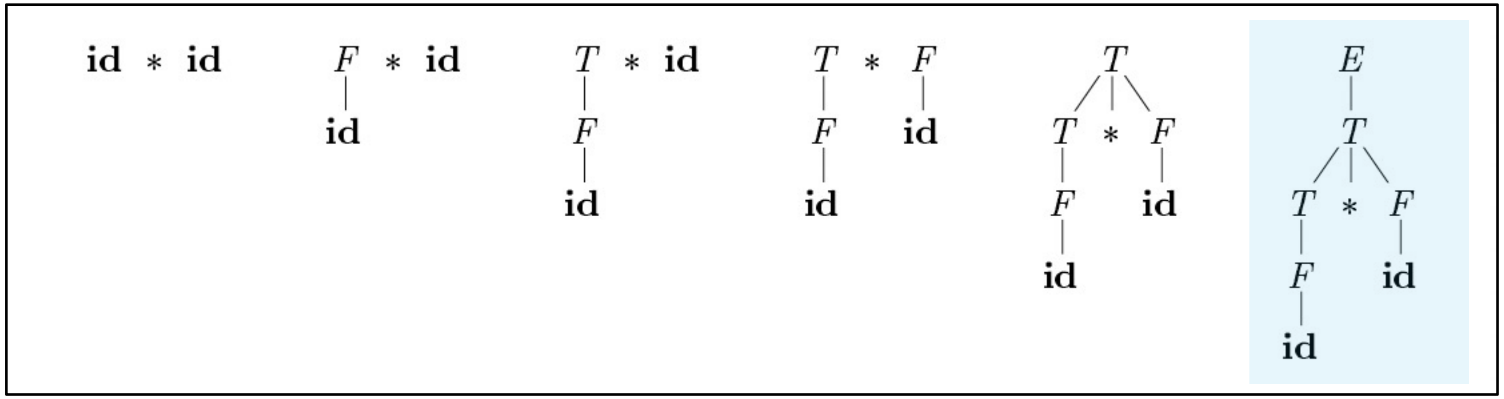
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

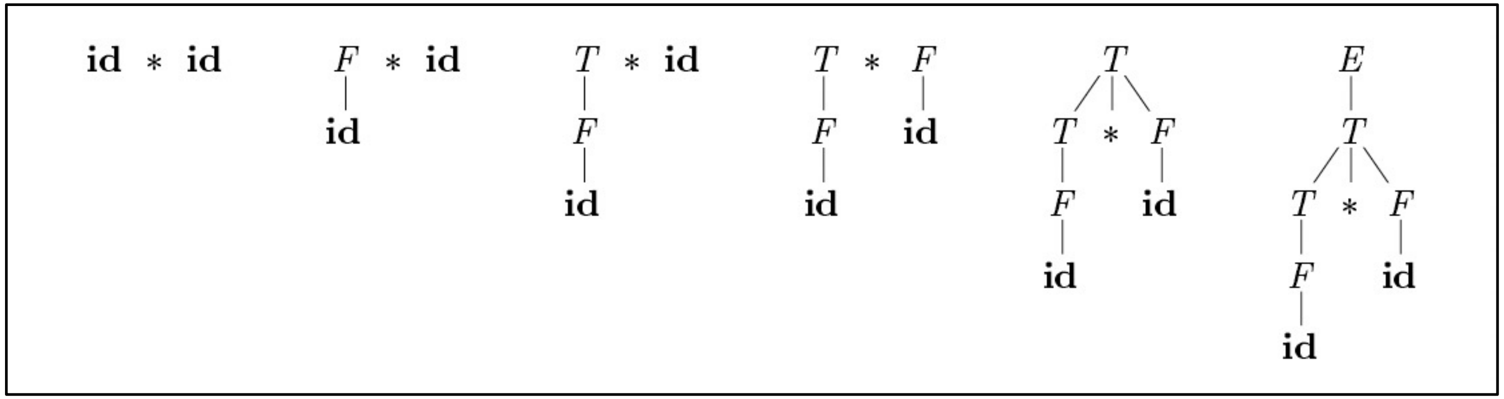
- Top-down parsing can be viewed as the problem of constructing a parse tree **in preorder**
- Bottom-up parsing can be viewed as the problem of constructing a parse tree **in post-order**



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

Bottom-up Parsing

- Keep shifting until we see the right-hand side of a rule
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting



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

Bottom-up Parsing

- Keep shifting until we see the right-hand side of a rule
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting

- Once we have the right-hand side of a rule:
  - Do we reduce right away, or do we keep shifting more symbols?
  - What if there are multiple rules with the same RHS to reduce by?

LR(0)

- LL(1) top-down parsing, we dealt with the tough decisions by just saying
  - **"if we have to make decisions, it's not an LL(1) grammar".**

LR(0)

- LL(1) top-down parsing, we dealt with the tough decisions by just saying
  - **"if we have to make decisions, it's not an LL(1) grammar".**

- We'll start out by looking at LR(0) parsing
  - We only worry about how to handle grammars that **don‘t require us to make decisions during parsing**

LR(0)

- LL(1) top-down parsing, we dealt with the tough decisions by just saying
  - "if we have to make decisions, it's not an LL(1) grammar".

- We'll start out by looking at LR(0) parsing
  - We only worry about how to handle grammars that **don‘t require us to make decisions during parsing**
  - Left-to-right scanning
  - Right-most derivation
  - Zero symbols of lookahead

LR(0)

- Keep shifting until we see the right-hand side of a rule
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting

LR(0)

- Keep shifting until we see the right-hand side of a rule
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting

- If this algorithm ever has to make decisions about which rule to
- reduce by, we give up and say "the grammar is not LR(0)".

LR(0)

- Keep shifting until we see the right-hand side of a rule
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting

- If this algorithm ever has to make decisions about which rule to
- reduce by, we give up and say "the grammar is not LR(0)".
  - LR(1), SLR(1), …
  - Refer to Chapter 4, the Dragon book!

LR(0)

???

- Keep shifting until **we see the right-hand side of a rule**
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting

- If this algorithm ever has to make decisions about which rule to
- reduce by, we give up and say "the grammar is not LR(0)".
  - LR(1), SLR(1), …
  - Refer to Chapter 4, the Dragon book!

LR(0)

??? DFA/NFA!!!

- Keep shifting until **we see the right-hand side of a rule**
- Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting

- If this algorithm ever has to make decisions about which rule to
- reduce by, we give up and say "the grammar is not LR(0)".
  - LR(1), SLR(1), …
  - Refer to Chapter 4, the Dragon book!

Recognizing the RHS

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

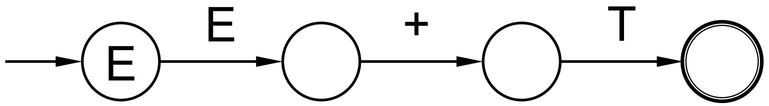
$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

Create DFAs for the RHS of each rule and mark the initial states with the LHS.

Recognizing the RHS

$E \to E + T$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

$E \to T$

$T \to T * F$

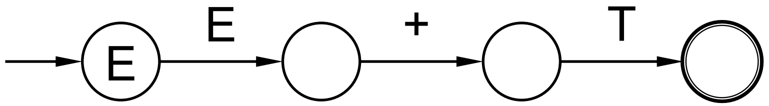
$T \to F$

$F \to (E)$

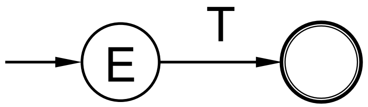
$F \to \textbf{id}$

Recognizing the RHS

$E \rightarrow E + T$



$E \rightarrow T$



$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

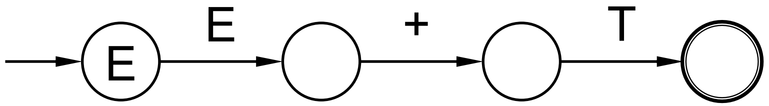
$F \rightarrow \mathbf{id}$

Create DFAs for the RHS of each rule and mark the initial states with the LHS.

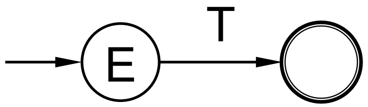
Recognizing the RHS

Create DFAs for the RHS of each rule and mark the initial states with the LHS.

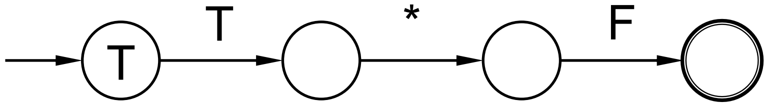
$E \rightarrow E + T$



$E \rightarrow T$



$T \rightarrow T * F$



$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

Recognizing the RHS

$E \to E + T$

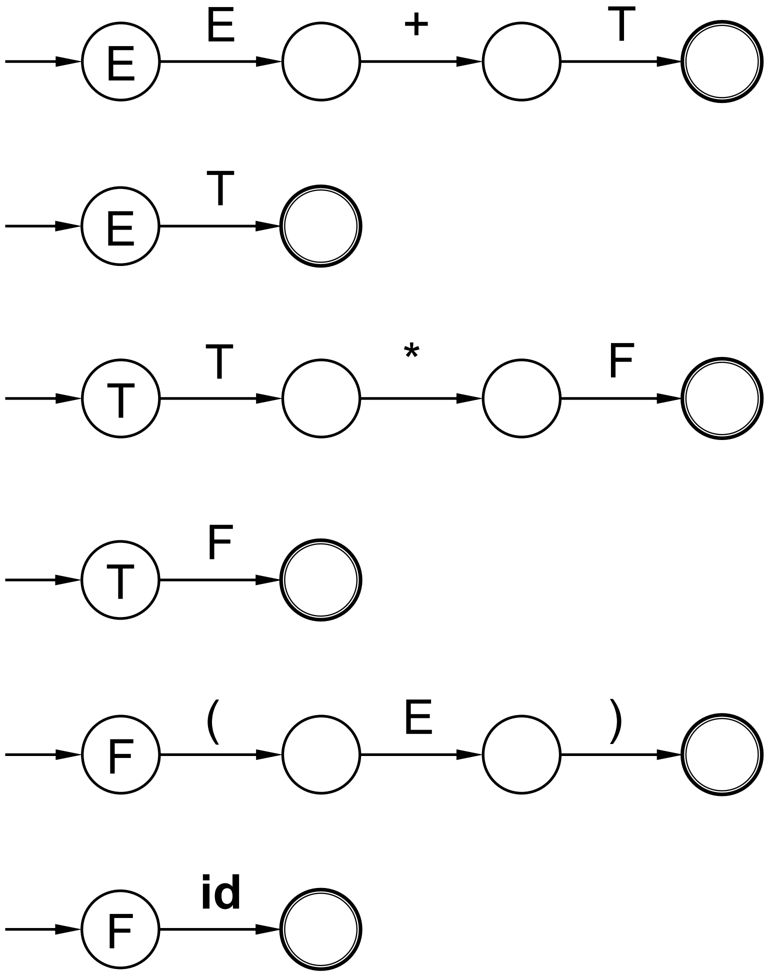
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

Recognizing the RHS

$E \to E + T$

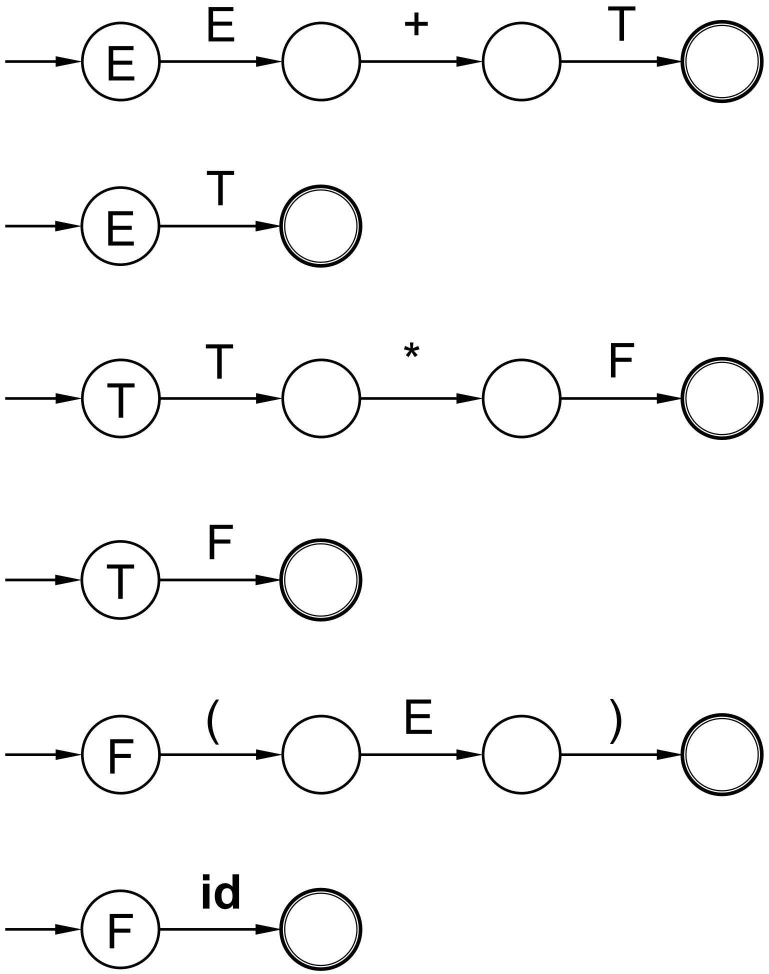
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

Recognizing the RHS

$E \rightarrow E + T$

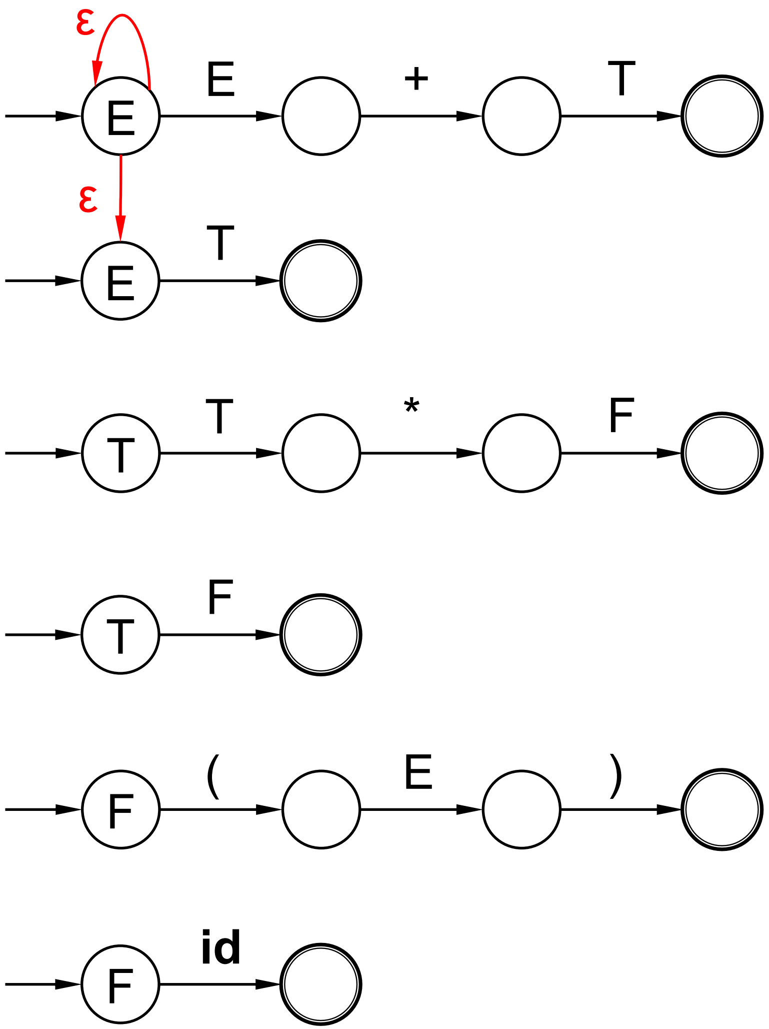
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

Recognizing the RHS

$E \to E + T$

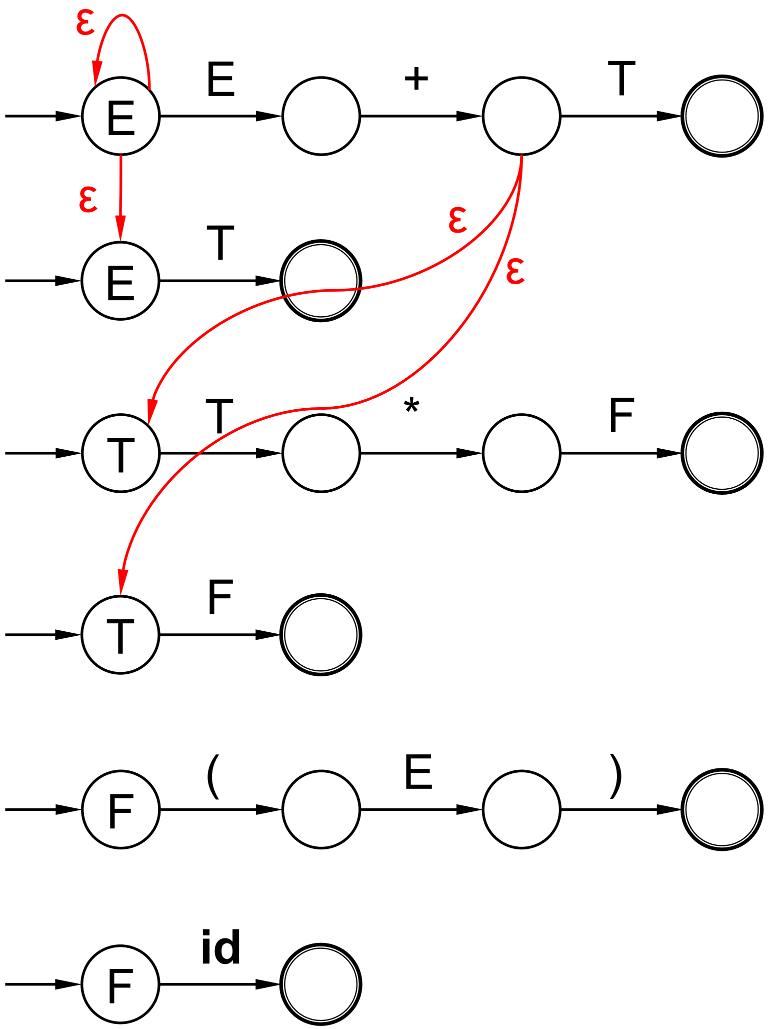
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

Recognizing the RHS

$E \to E + T$

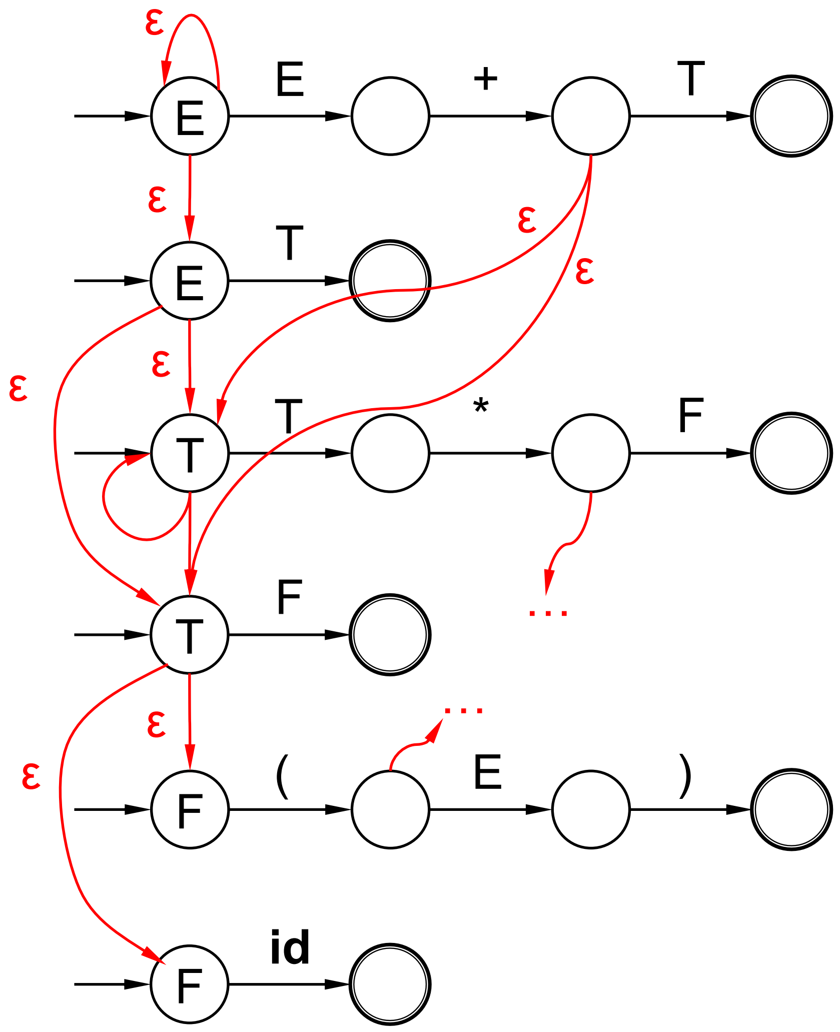
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \textbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

Recognizing the RHS

$E \to E + T$

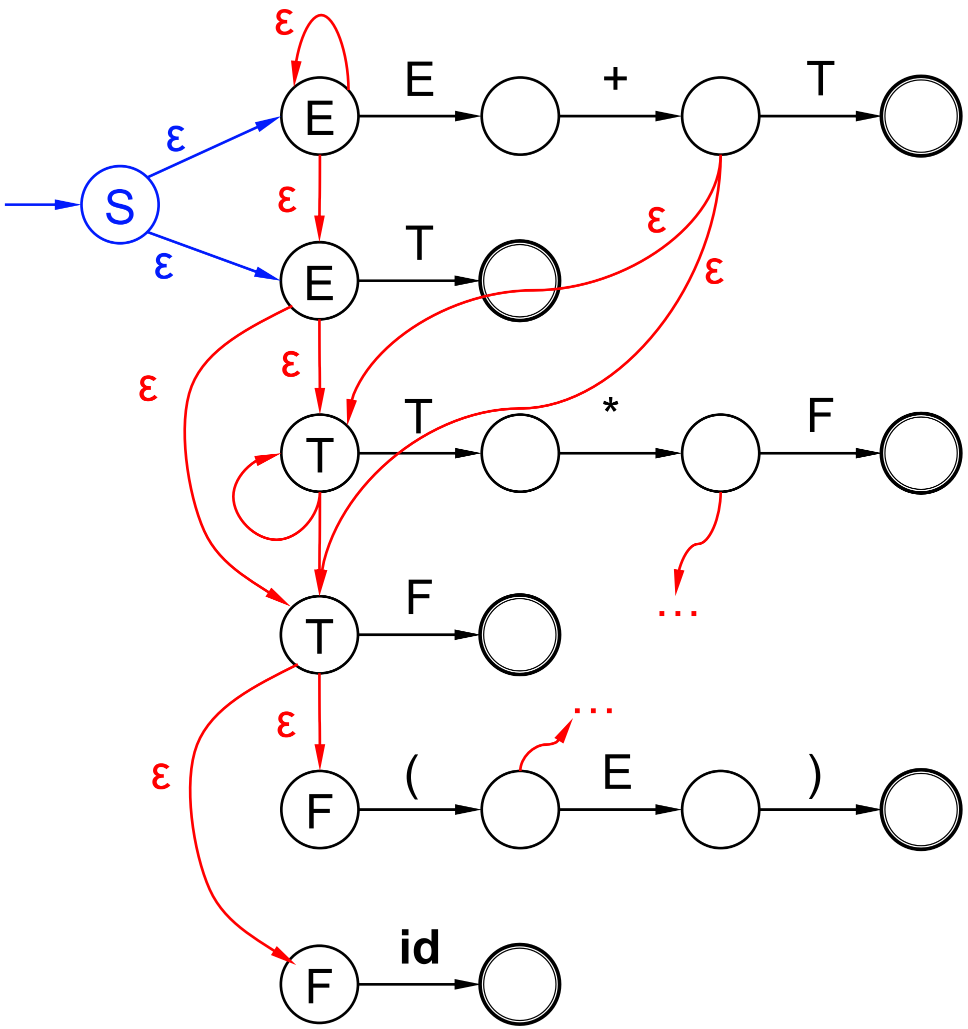
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

Recognizing the RHS

$E \to E + T$

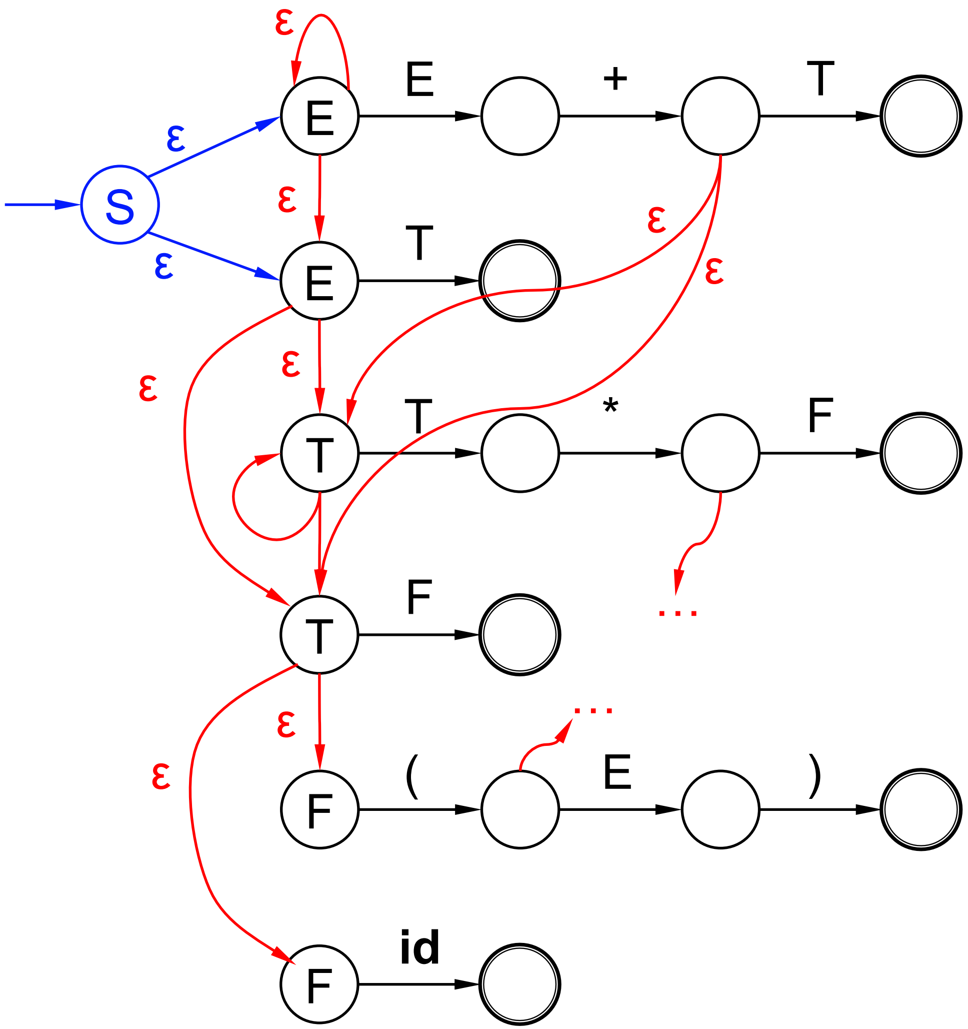
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

We can transform it into a DFA

Recognizing the RHS

$E \rightarrow E + T$

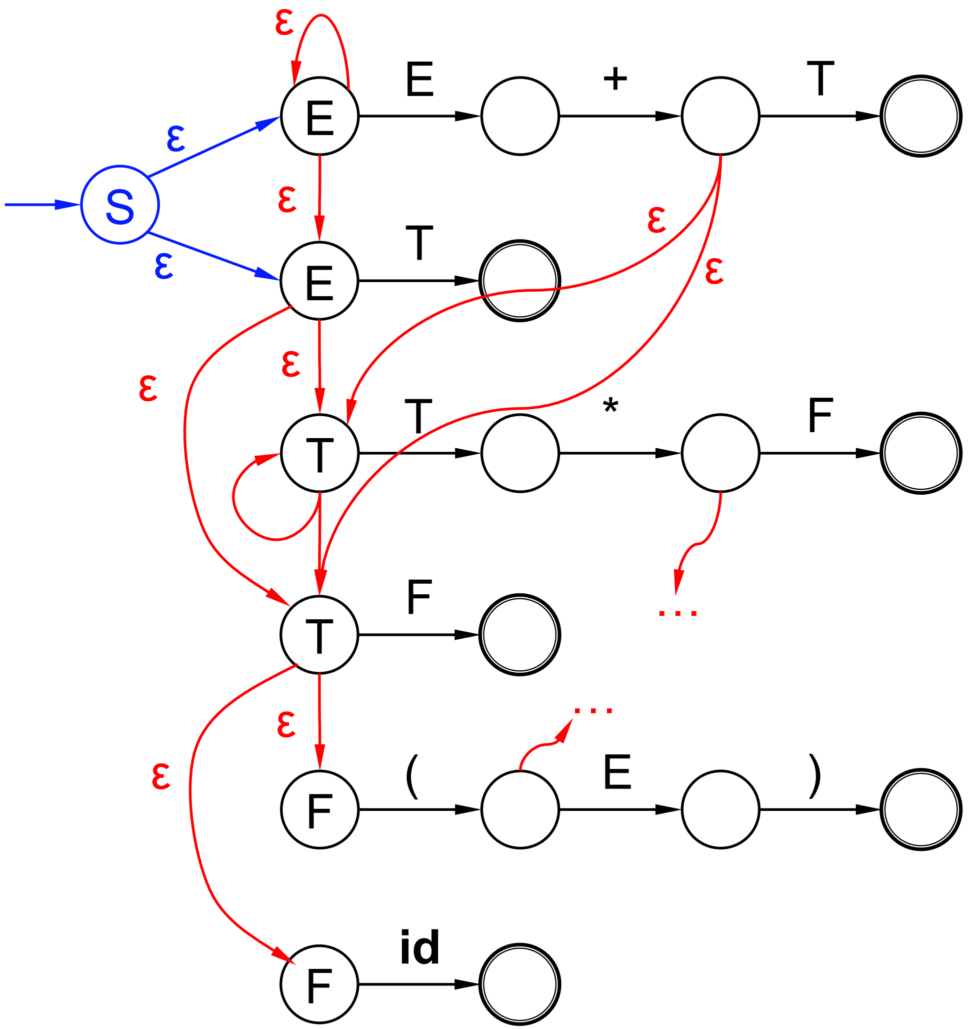
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \textbf{id}$



Create DFAs for the RHS of each rule and mark the initial states with the LHS.

For each state with a transition leading **outwards on a nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

We can transform it into a DFA

When shifting during LR(0) parsing, if we reach a final state, use the corresponding non-terminal to reduce.

Recognizing the RHS

$E \rightarrow E + T$

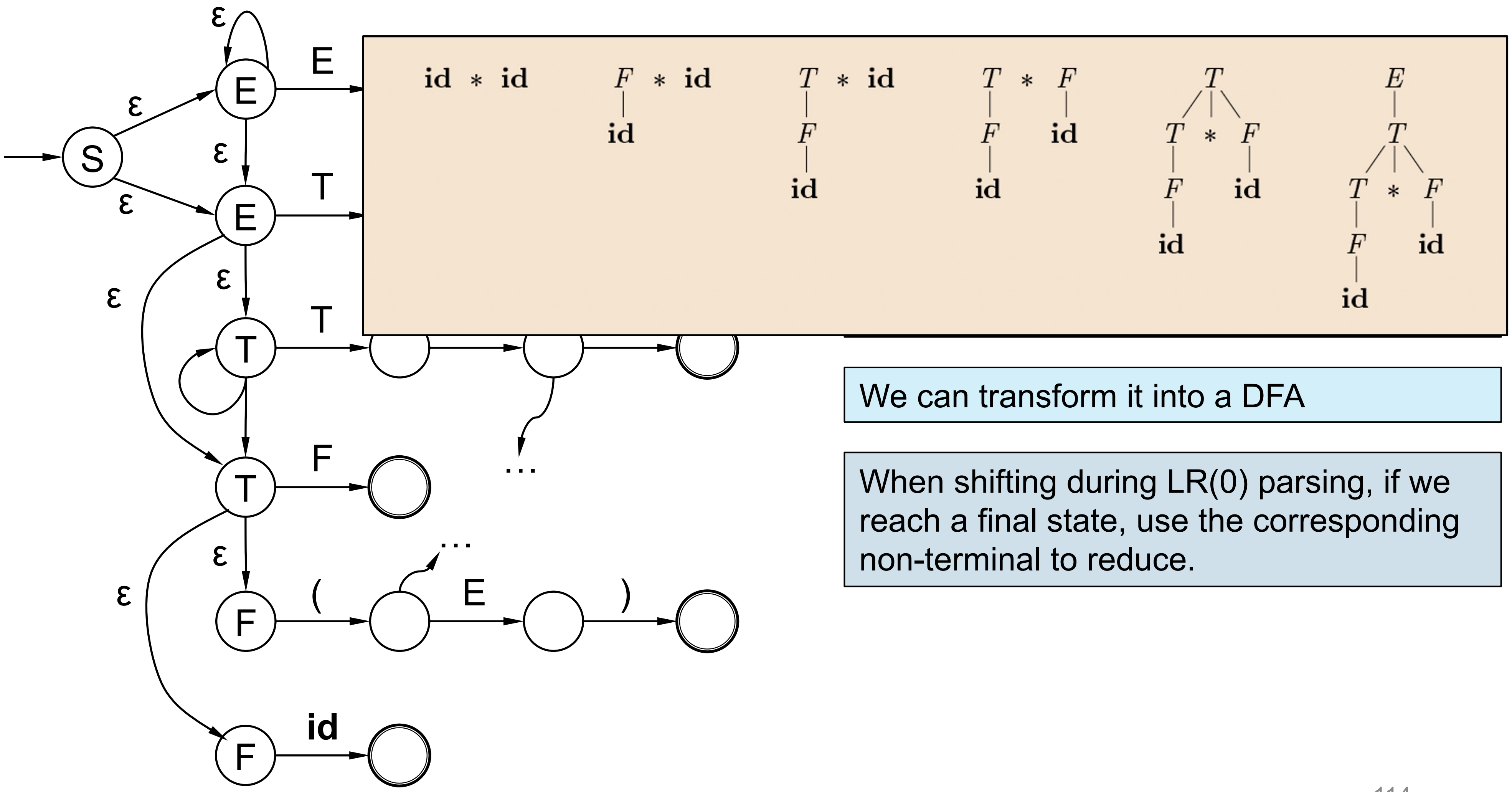
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$



We can transform it into a DFA

When shifting during LR(0) parsing, if we reach a final state, use the corresponding non-terminal to reduce.

Recognizing the RHS

$E \rightarrow E + T$

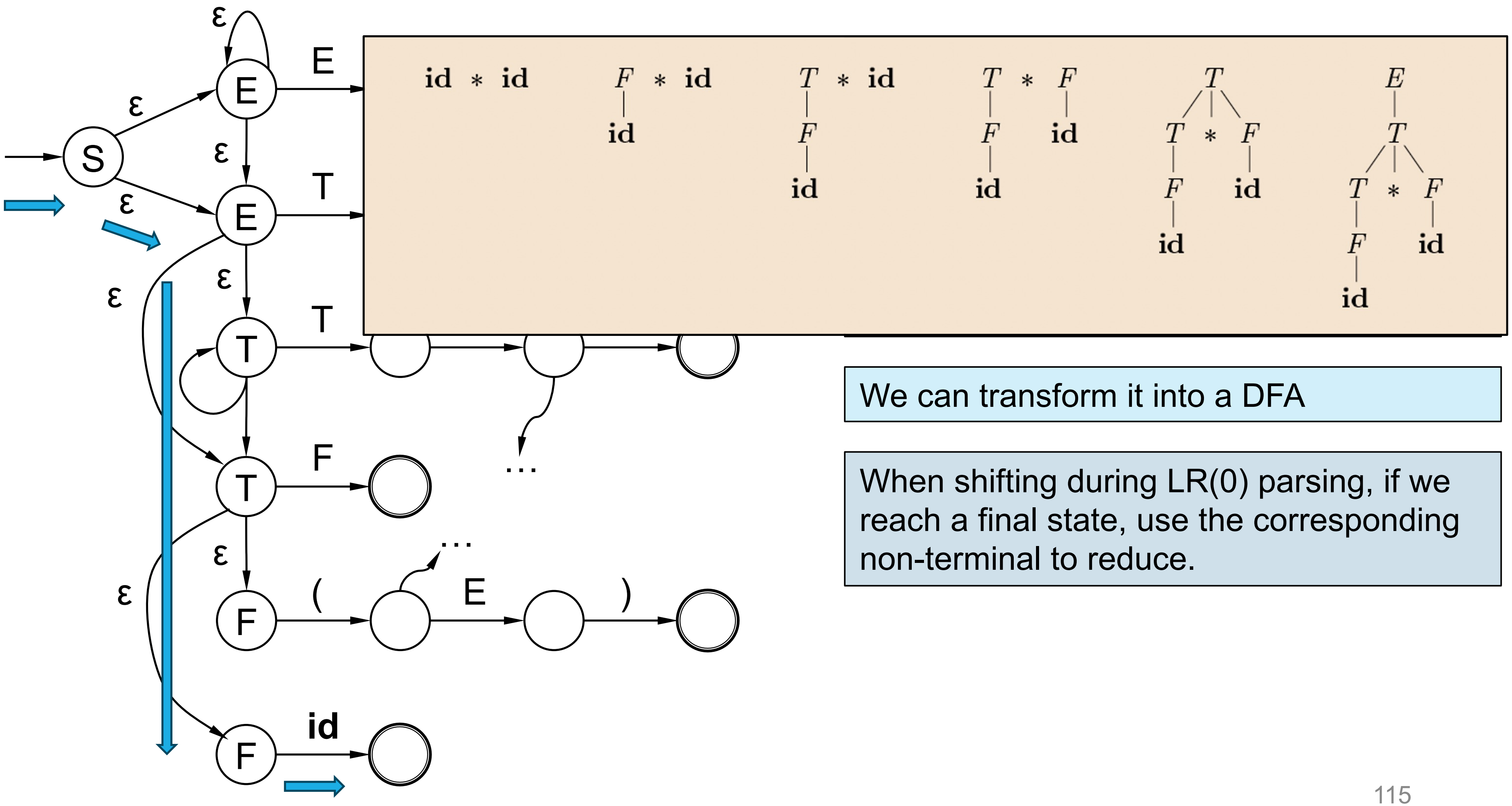
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \textbf{id}$



We can transform it into a DFA

When shifting during LR(0) parsing, if we reach a final state, use the corresponding non-terminal to reduce.

Recognizing the RHS

$E \to E + T$

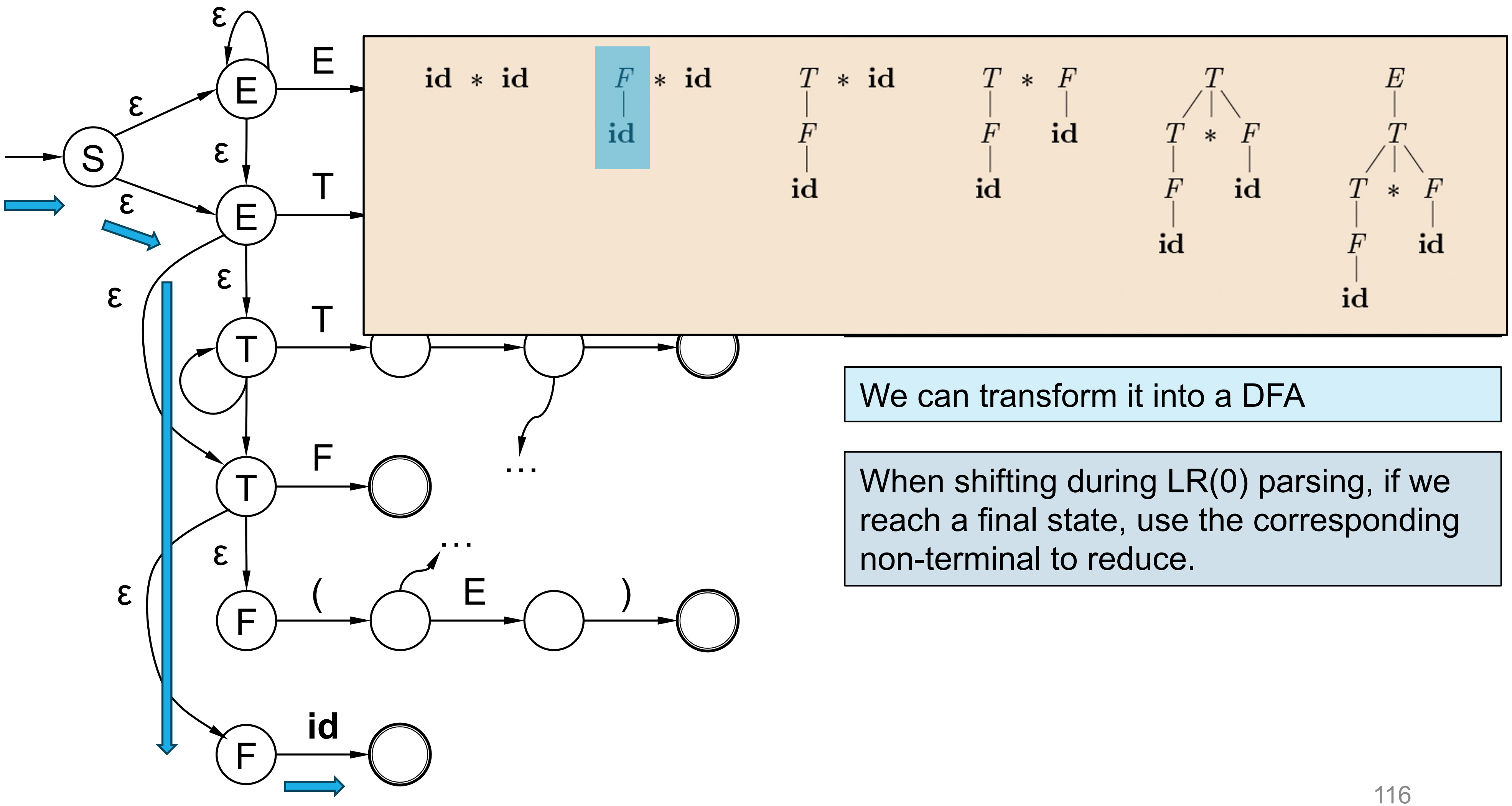
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to \mathbf{id}$



We can transform it into a DFA

When shifting during LR(0) parsing, if we reach a final state, use the corresponding non-terminal to reduce.

Recognizing the RHS

$E \rightarrow E + T$

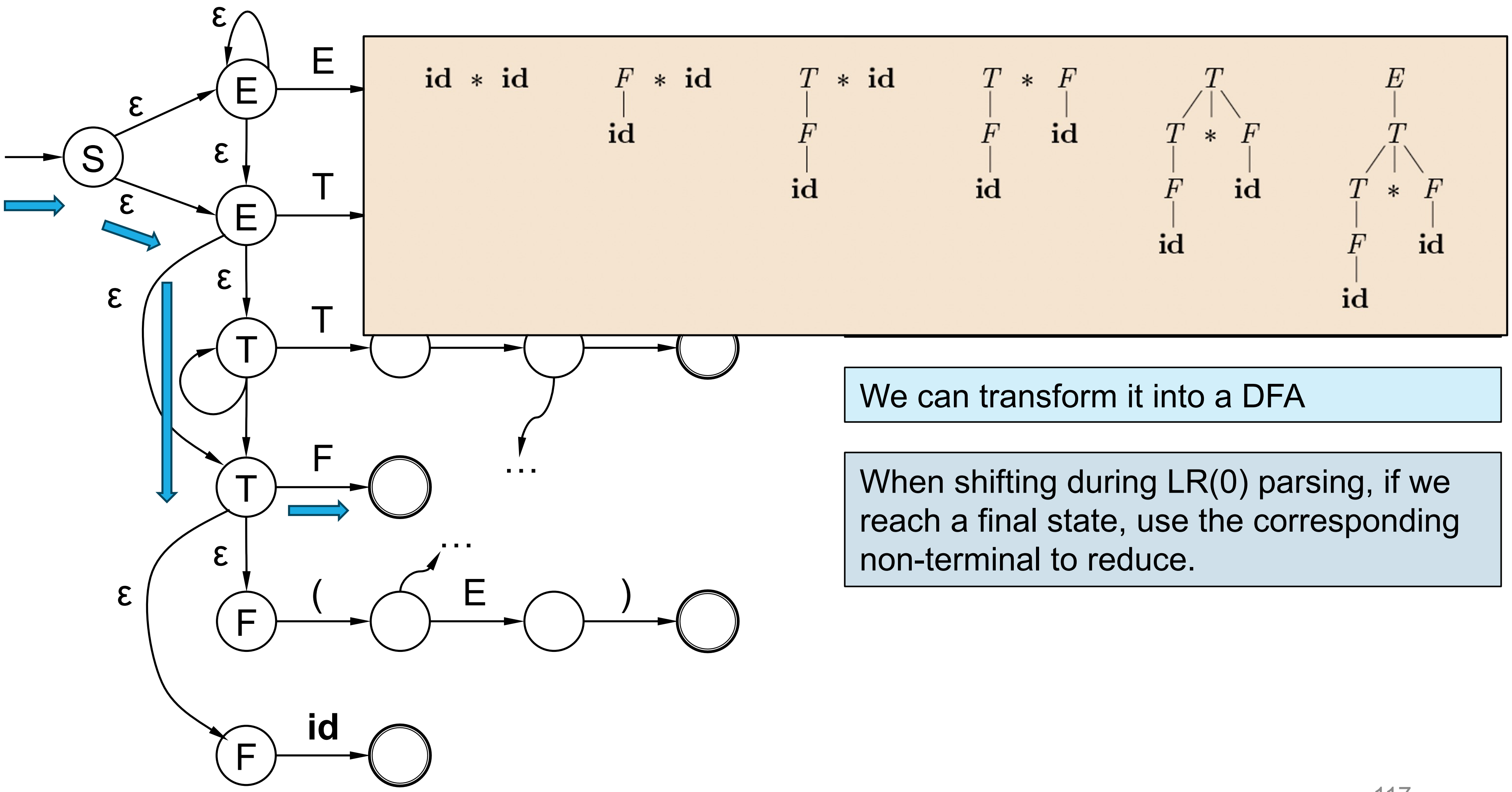
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$



We can transform it into a DFA

When shifting during LR(0) parsing, if we reach a final state, use the corresponding non-terminal to reduce.

Recognizing the RHS

$E \to E + T$

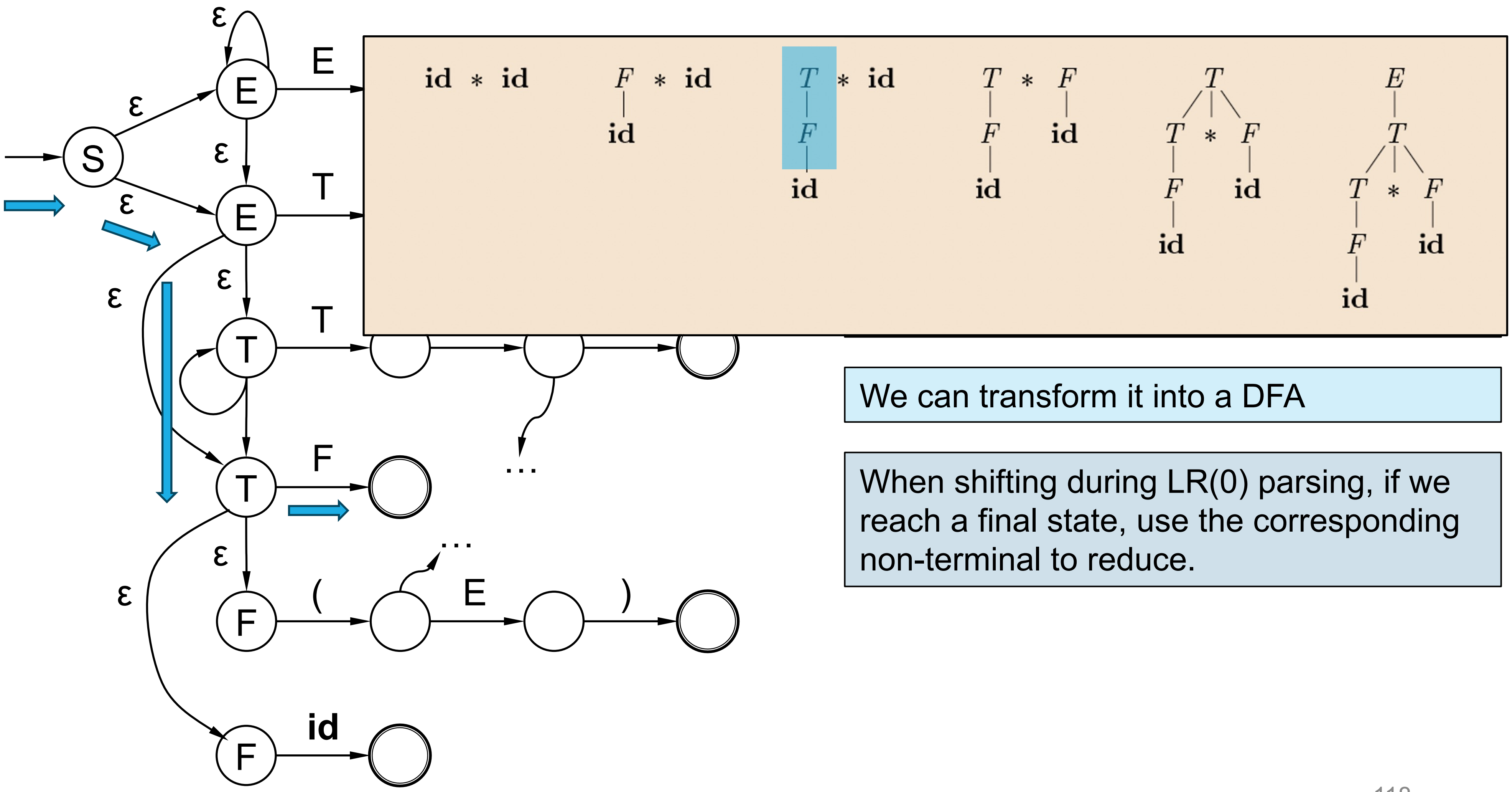
$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

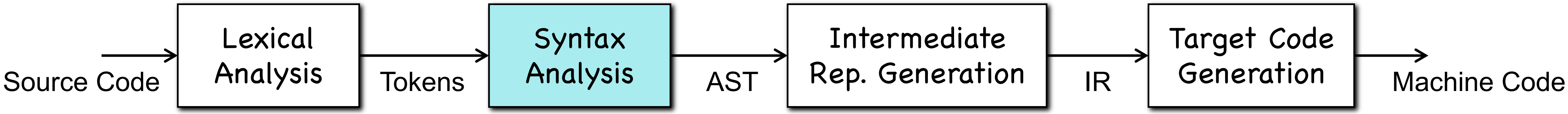
$F \to \mathbf{id}$



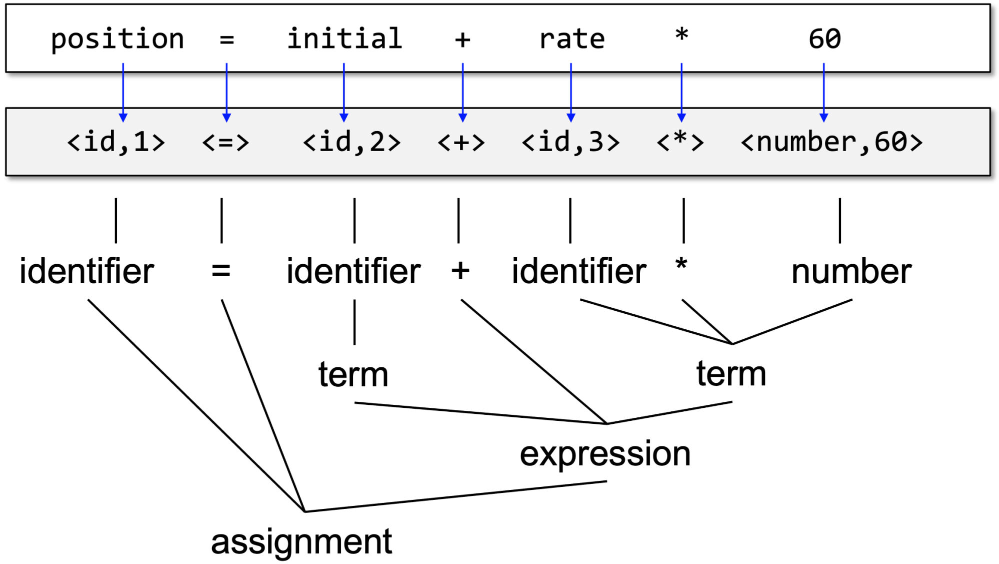
We can transform it into a DFA

When shifting during LR(0) parsing, if we reach a final state, use the corresponding non-terminal to reduce.

Summary



- Syntax analysis is a procedure of building the parse/syntax tree
  - Top-down parsing and LL parsing
    - Recursive-descent parsers
    - Eliminating left-recursive
    - Predictive parsers, LL(1) parsers
    - Non-recursive predictive parser vs. PDA
  - Bottom-up parsing and LR parsing
    - LR(0) parser
    - Refer to § 4.5, Chapter 4, the Dragon book!


position = initial + rate * 60
<id,1> <=> <id,2> <+> <id,3> <*> <number,60>
identifier
=
identifier
+
identifier
*
number
term
term
expression
assignment

THANKS!