Chapter 4
Syntax Analysis

Part 1
TOP-DOWN PARSING

Overview

mansoori@shirazu.ac.ir

- Parsing methods in typical compilers
  - Techniques suitable for hand implementation
  - Algorithms used in automated tools
  - Extensions of parsing methods for recovery from errors
- The syntax of a construct, specified by CFG:
  - Gives precise and easy-to-understand specification of constructs
  - For certain classes of CFGs, efficient parser can be constructed
  - A properly-designed CFG for structures is useful for translating and detecting errors
  - A CFG allows a language to be evolved iteratively

The Parser

mansoori@shirazu.ac.ir

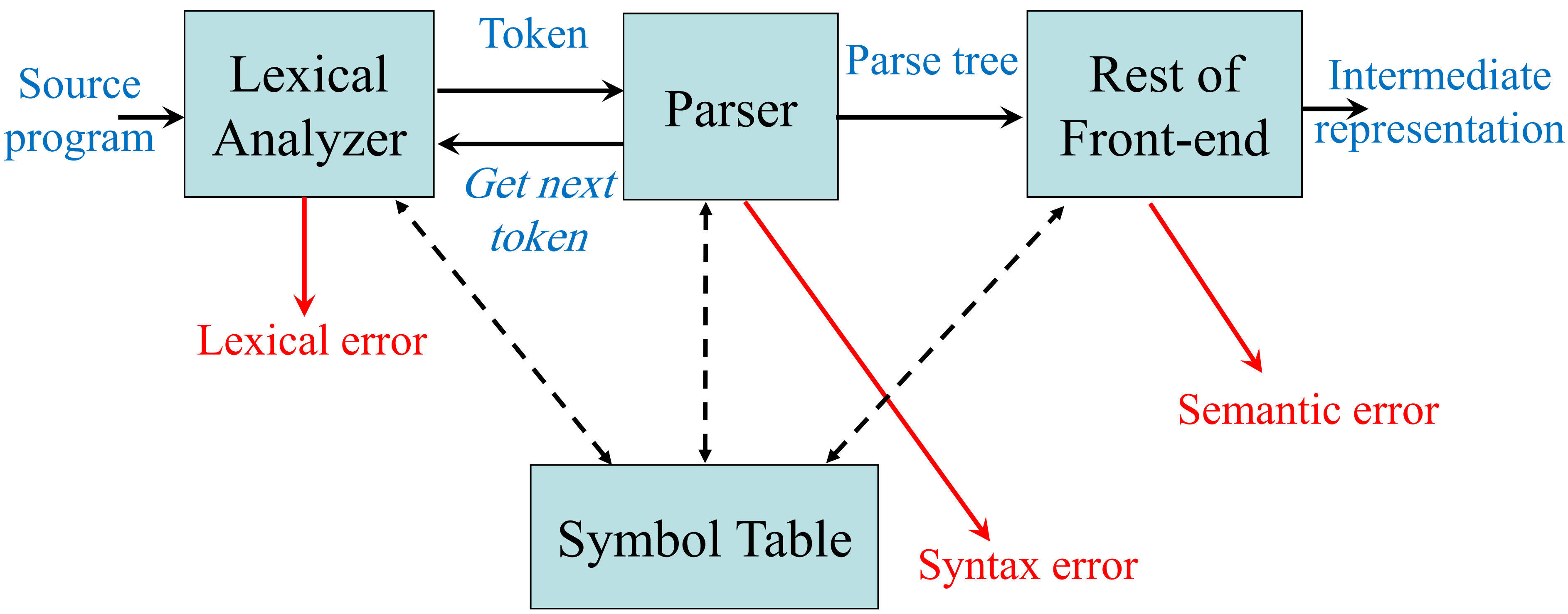
- A parser implements a CFG

- The roles of the parser:
  1. To check syntax (as string recognizer), and
     - To report syntax errors accurately

  2. To invoke semantic actions
     - For static semantics checking, (type checking of expressions, functions, …)
     - For syntax-directed translation of the source code to an intermediate representation

Position of Parser in Compiler Model

mansoori@shirazu.ac.ir



Syntax Error Handling

mansoori@shirazu.ac.ir

- A good compiler should assist in:
  - Identifying and locating syntax errors, and
  - Recovering from almost all of them

- Viable-prefix property of parsers allows early detection of syntax errors
  - Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

Prefix
```
…
for (;)
…
```
Error is detected here

Prefix
```
…
DO 10 I = 1;0
…
```
Error is detected here

Error Recovery Strategies

mansoori@shirazu.ac.ir

- Panic mode
  - Discard input tokens until a token in a set of designated synchronizing tokens is found
- Phrase-level recovery
  - Perform local correction on input tokens to repair syntax error
- Error productions
  - Augment grammar with productions for erroneous constructs
- Global correction
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Three General Types of Parsers

mansoori@shirazu.ac.ir

1. Universal parsers: CYK and Earley's algorithms
   - Can parse any grammar
   - Too inefficient to use in compiler production

2. Top-down parsers: LL parsers (for small set of CFGs)
   - Build parse trees from the top (root) to the bottom (leaves)
   - LL parsers are often implemented by hand

3. Bottom-up parsers: LR parsers (for large set of CFGs)
   - Start from the leaves and work their way up to the root
   - LR parsers are usually constructed using automated tools

- Input to parser is scanned from left to right, one token at a time

Top-Down Parsing

mansoori@shirazu.ac.ir

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

$E \rightarrow T + T$

$T \rightarrow (\ E\ )$

$T \rightarrow -\ E$

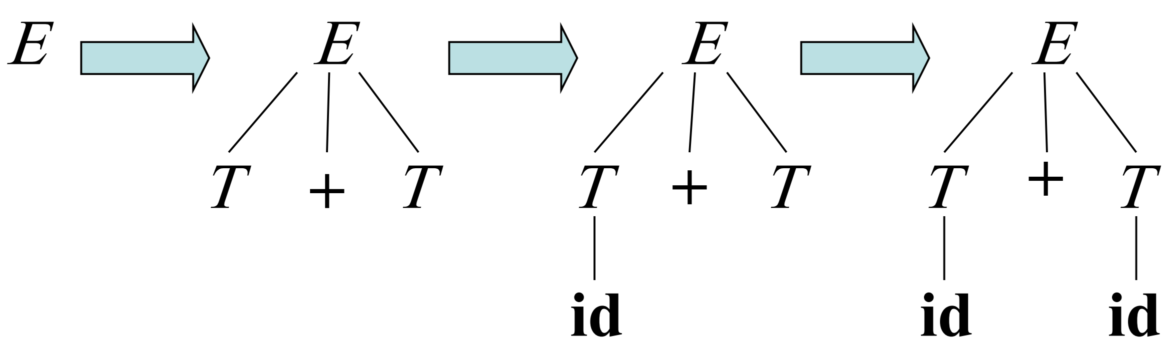
$T \rightarrow \mathbf{id}$

Leftmost derivation:

$E \Rightarrow_{lm} T + T$

$\Rightarrow_{lm} \mathbf{id} + T$

$\Rightarrow_{lm} \mathbf{id} + \mathbf{id}$



Grammars (Recap)

mansoori@shirazu.ac.ir

- Context-free grammar (CFG) is a 4-tuple $G = (N, T, P, S)$ where
  - T: a finite set of tokens (*terminals*)
  - N: a finite set of *nonterminals*
  - P: a finite set of *productions* of the form $A \to \beta$ where $A \in N$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start nonterminal*

Example Grammar

mansoori@shirazu.ac.ir

CFG defines simple arithmetic expressions

- $T = \{ +, -, *, /, (,), \textbf{id}, \textbf{num} \}$
- $N = \{ \textit{expression}, \textit{term}, \textit{factor} \}$
- $S = \textit{expression}$
- $P = \{$
  $\textit{expression} \rightarrow \textit{expression} + \textit{term}$
  $\textit{expression} \rightarrow \textit{expression} - \textit{term}$
  $\textit{expression} \rightarrow \textit{term}$
  $\textit{term} \rightarrow \textit{term} * \textit{factor}$
  $\textit{term} \rightarrow \textit{term} \,/\, \textit{factor}$
  $\textit{term} \rightarrow \textit{factor}$
  $\textit{factor} \rightarrow (\textit{expression})$
  $\textit{factor} \rightarrow \textbf{id}$
  $\textit{factor} \rightarrow \textbf{num} \quad \}$

In notational conventions:

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

Derivations (Recap)

mansoori@shirazu.ac.ir

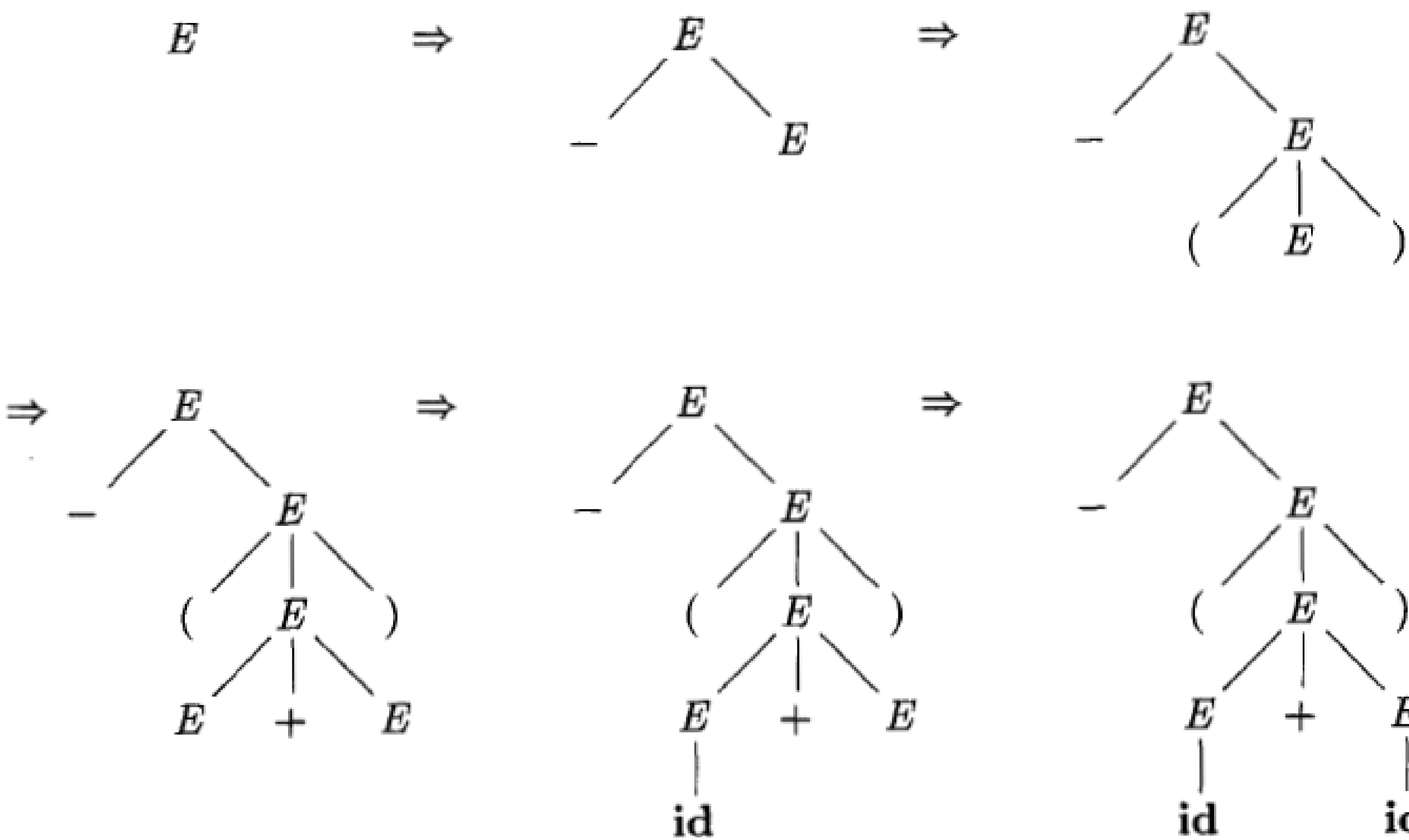
- The *one-step derivation* is defined by
  $$\alpha A \beta \Rightarrow \alpha \gamma \beta$$
  where $A \rightarrow \gamma$ is a production in the grammar
- In addition, we define
  - *Leftmost derivation* ($\Rightarrow_{lm}$) if $\alpha \in T^*$
  - *Rightmost derivation* ($\Rightarrow_{rm}$) if $\beta \in T^*$
  - Transitive closure ($\Rightarrow^*$): zero or more steps in derivation
  - Positive closure ($\Rightarrow^+$): one or more steps in derivation
- The *language* generated by G is defined as:
  $$L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}$$

Derivations and Parse Trees

mansoori@shirazu.ac.ir

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

Parse tree for $- (\textbf{id} + \textbf{id})$



mansoori@shirazu.ac.ir

Ambiguity

- A grammar is *ambiguous* if
  - It produces more than one parse tree for some sentence, or
  - It produces more than one leftmost derivation or more than one rightmost derivation for the same sentence

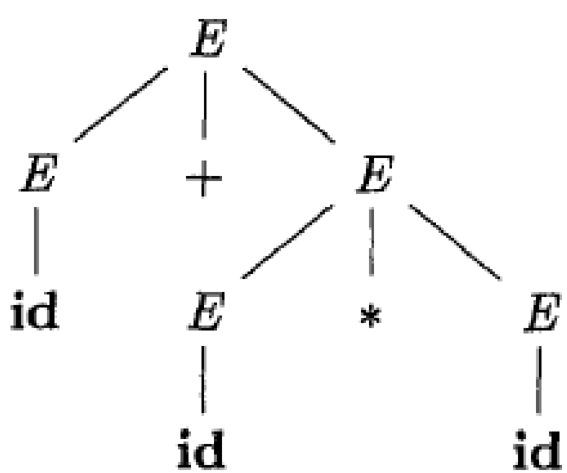
mansoori@shirazu.ac.ir

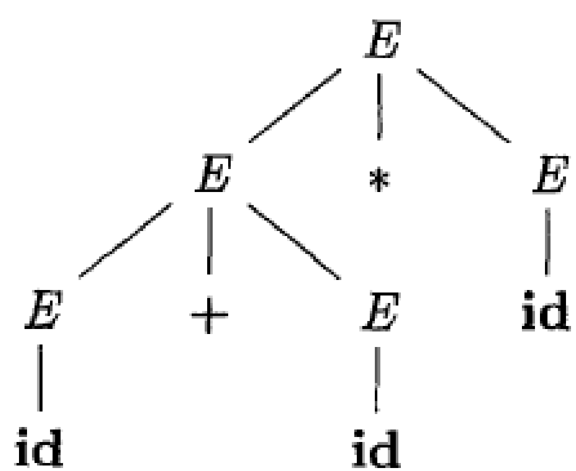
Ambiguity Example 1

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

Two distinct leftmost derivations for **id** + **id** * **id**

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





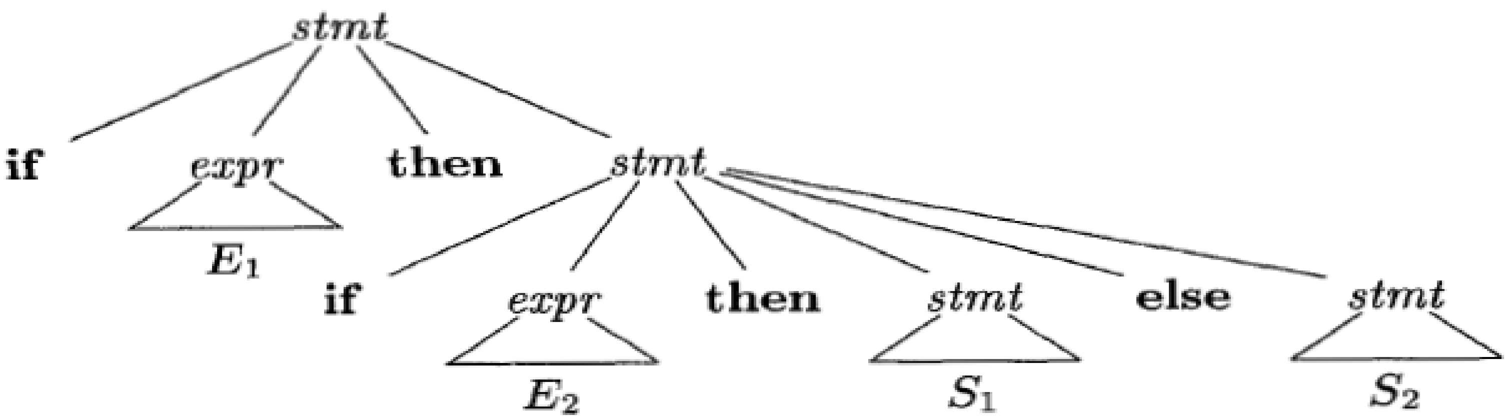
Ambiguity Example 2

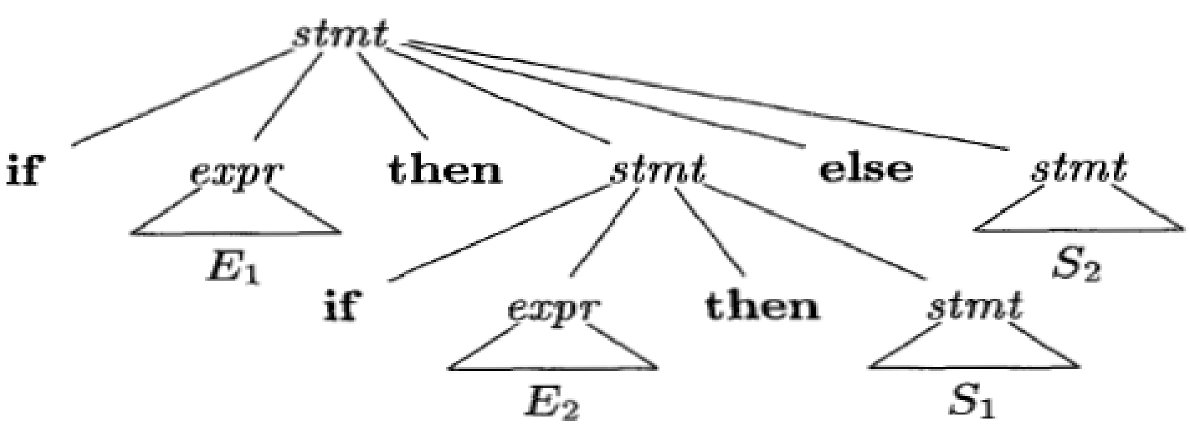
mansoori@shirazu.ac.ir

- *Dangling else* grammar:

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

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





Eliminating Ambiguity

mansoori@shirazu.ac.ir

- Sometimes, an ambiguous grammar can be rewritten to eliminate the ambiguity

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

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

Left Recursion

mansoori@shirazu.ac.ir

- Left-recursion: $A \overset{+}{\Rightarrow} A\alpha$
- Immediate left-recursion: $A \rightarrow A\alpha$
- When one of the productions in a grammar is left-recursive then a top-down parser (predictive parser) loops forever on certain inputs

Immediate Left Recursion Elimination

mansoori@shirazu.ac.ir

1. Group the productions as:

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

where no β_i begins with an A

2. Replace the A-productions by:

$$A \rightarrow \beta_1 R \mid \beta_2 R \mid \ \ldots \mid \beta_n R$$
$$R \rightarrow \alpha_1 R \mid \alpha_2 R \mid \ \ldots \mid \alpha_m R \mid \varepsilon$$

where R is a new nonterminal

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

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

General Left Recursion Elimination

mansoori@shirazu.ac.ir

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 $A_i \to A_j\, \gamma$

        by $A_i \to \delta_1\, \gamma \mid \delta_2\, \gamma \mid \ldots \mid \delta_k\, \gamma$

        where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$

    }

    Eliminate the immediate left recursion in A_i

}

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

$S,\ A$

$$A \to A\ c \ \mid\ A\ a\ d \ \mid\ b\ d \ \mid\ \epsilon$$

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

Example Left Recursion Elimination

mansoori@shirazu.ac.ir

$$\left.\begin{array}{l} A \to B\,C \mid \mathbf{a} \\ B \to C\,A \mid A\,\mathbf{b} \\ C \to A\,B \mid C\,C \mid \mathbf{a} \end{array}\right\}$$ Choose arrangement: A, B, C

$i = 1$: nothing to do

$i = 2, j = 1$: $B \to C\,A \mid \underline{A}\,\mathbf{b}$

$\Rightarrow \quad B \to C\,A \mid \underline{B\,C}\,\mathbf{b} \mid \underline{\mathbf{a}}\,\mathbf{b}$

$\Rightarrow_{\text{(imm)}} \quad B \to C\,A\,R \mid \mathbf{a}\,\mathbf{b}\,R$

$R \to C\,\mathbf{b}\,R \mid \varepsilon$

$i = 3, j = 1$: $C \to \underline{A}\,B \mid C\,C \mid \mathbf{a}$

$\Rightarrow \quad C \to \underline{B\,C}\,B \mid \underline{\mathbf{a}}\,B \mid C\,C \mid \mathbf{a}$

$i = 3, j = 2$: $C \to \underline{B}\,C\,B \mid \mathbf{a}\,B \mid C\,C \mid \mathbf{a}$

$\Rightarrow \quad C \to \underline{C\,A\,R}\,C\,B \mid \underline{\mathbf{a}\,\mathbf{b}\,R}\,C\,B \mid \mathbf{a}\,B \mid C\,C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}} \quad C \to \mathbf{a}\,\mathbf{b}\,R\,C\,B\,S \mid \mathbf{a}\,B\,S \mid \mathbf{a}\,S$

$S \to A\,R\,C\,B\,S \mid C\,S \mid \varepsilon$

Example Left Recursion Elimination

mansoori@shirazu.ac.ir

$$A \rightarrow B\,C \mid \mathbf{a}$$
$$B \rightarrow C\,A \mid A\,\mathbf{b}$$
$$C \rightarrow A\,B \mid C\,C \mid \mathbf{a}$$

$$A \rightarrow B\,C \mid \mathbf{a}$$
$$B \rightarrow C\,A\,R \mid \mathbf{a}\,\mathbf{b}\,R$$
$$R \rightarrow C\,\mathbf{b}\,R \mid \varepsilon$$
$$C \rightarrow \mathbf{a}\,\mathbf{b}\,R\,C\,B\,S \mid \mathbf{a}\,B\,S \mid \mathbf{a}\,S$$
$$S \rightarrow A\,R\,C\,B\,S \mid C\,S \mid \varepsilon$$

Left Factoring

mansoori@shirazu.ac.ir

- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not suitable for top-down (predictive) parsing

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

Left Factoring a Grammar

mansoori@shirazu.ac.ir

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

- Replace all A-productions

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

where γ represents all alternatives not begin with α

with

$$A \rightarrow \alpha\, R \mid \gamma$$
$$R \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where R is a new nonterminal

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$
$$E \rightarrow b$$

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

Non-Context-Free Constructs

mansoori@shirazu.ac.ir

- A few syntactic constructs found in typical programming languages cannot be specified by grammars

- Checking declaration of identifiers before their use

$$L_1 = \{wcw \mid w \in \{a, b\}^*\}$$

Semantic-analysis phase checks that identifiers are declared before they are used

- Checking agreement of number of formal parameters in declaration with number of actual parameters in use

$$L_2 = \{a^n b^m c^n d^m \mid n, m \geq 1\}$$

Semantic-analysis phase checks that the number of parameters in a call is correct

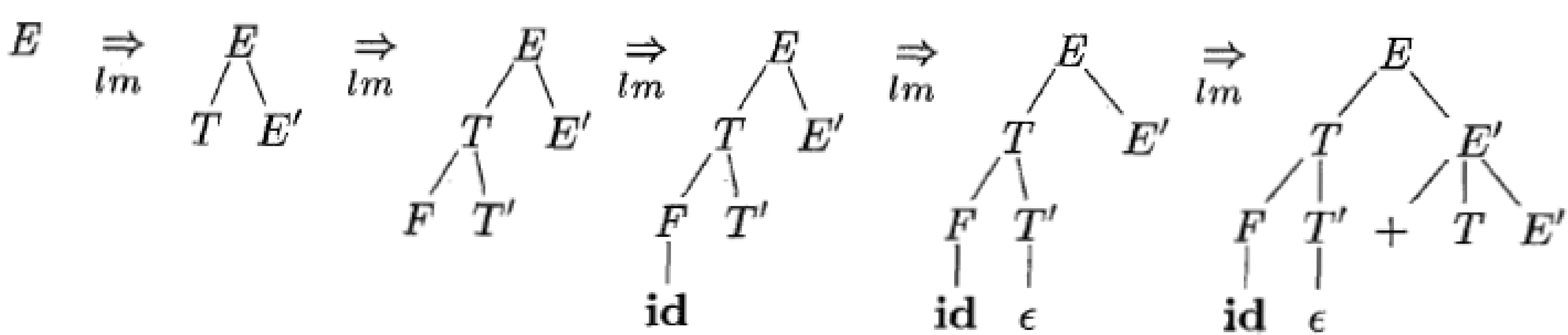
Top-Down Parsing

mansoori@shirazu.ac.ir

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

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

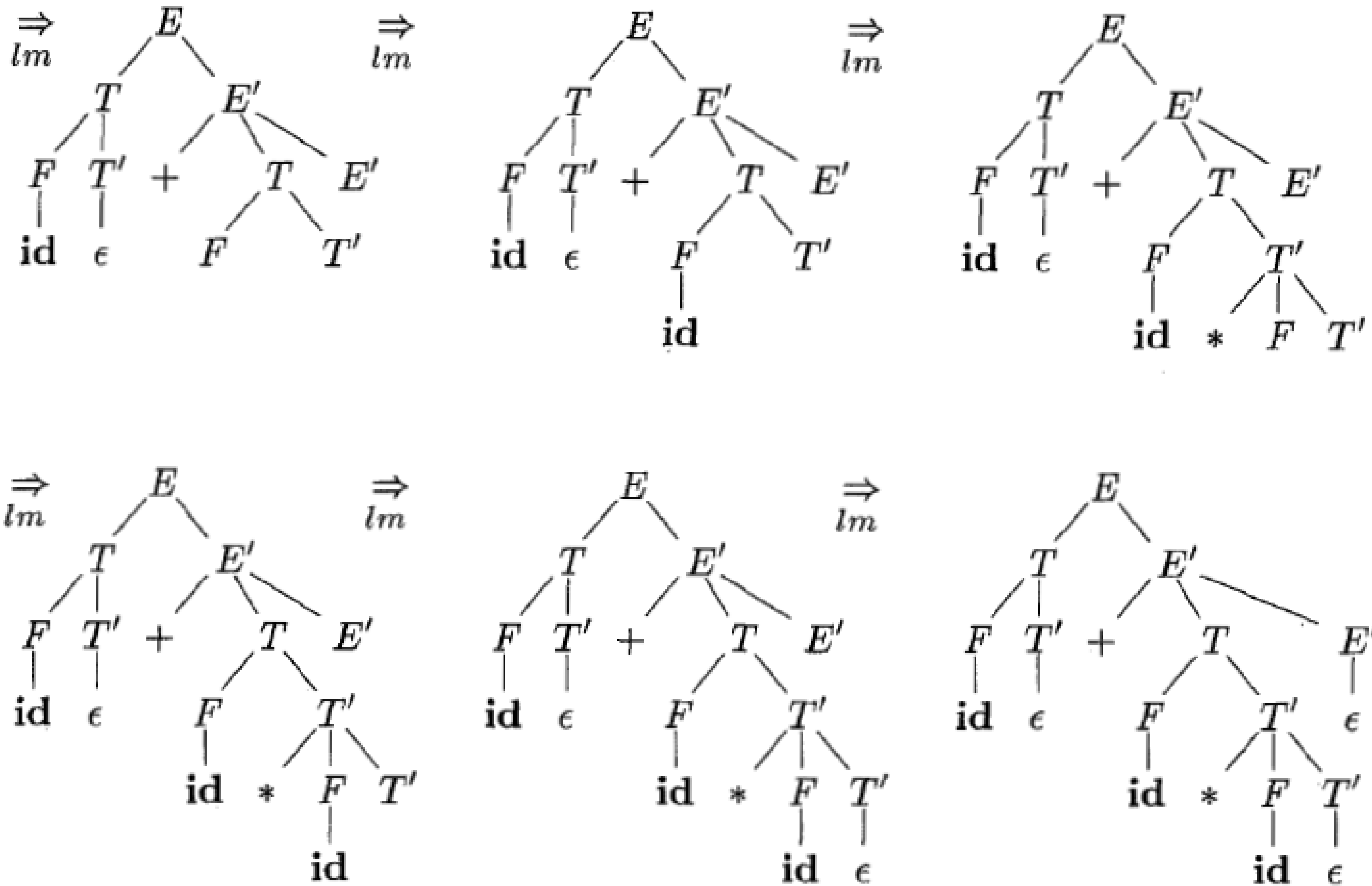
Top-down parse for: **id + id * id**



Top-Down Parsing

mansoori@shirazu.ac.ir

Top-down parse for: **id + id * id**



Top-Down Parsing Methods

mansoori@shirazu.ac.ir

- Recursive-descent parsing
  - A general form of top-down parsing
  - May require backtracking to find the correct *A*-production
- Predictive parsing
  - A special case of recursive-descent parsing
  - No backtracking is required
  - Chooses the correct *A*-production by looking ahead the next token
  - Suitable for LL(1) class of grammars

Recursive-Descent Parser

mansoori@shirazu.ac.ir

- Consists of a set of procedures, one for each nonterminal
- Execution begins with procedure of start nonterminal, which halts and announces success if its body scans entire input

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

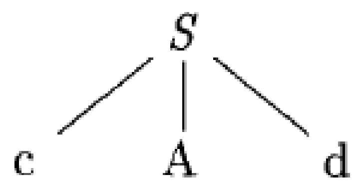
Suggests to return and try another *A*-production

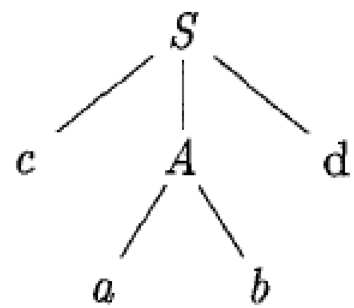
Recursive-Descent Parser

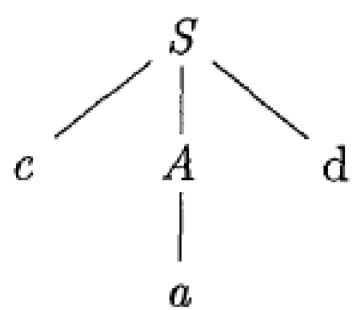
mansoori@shirazu.ac.ir

$$
\begin{aligned}
S &\rightarrow c\ A\ d \\
A &\rightarrow a\ b \mid a
\end{aligned}
$$

parsing: **c a d**







mansoori@shirazu.ac.ir

Predictive Parser

- Rewrite the grammar to eliminate the ambiguity, if any
- Eliminate left-recursion from grammar
- Do left-factoring on the grammar
- Compute FIRST and FOLLOW sets

- Two variants of predictive parser:
  - Recursive: uses recursive calls
  - Non-recursive: table-driven parser

FIRST and FOLLOW Sets

mansoori@shirazu.ac.ir

- Vital in construction of top-down and bottom-up parsers
- In top-down parsing, FIRST and FOLLOW allow to choose which production to apply, based on the next token
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens

FIRST Function

mansoori@shirazu.ac.ir

FIRST(α) = {terminals that begin all strings derived from α}

FIRST(a) = $\{a\}$ if $a \in T$

FIRST(ε) = $\{\varepsilon\}$

FIRST(A) = $\cup_{A\to\alpha}$ FIRST(α) for $A\to\alpha \in P$

FIRST($X_1X_2\ldots X_k$) =

if for all $j = 1, \ldots, i\text{-}1$: $\varepsilon \in$ FIRST(X_j) **then**

add non-ε in FIRST(X_i) to FIRST($X_1X_2\ldots X_k$)

if for all $j = 1, \ldots, k$: $\varepsilon \in$ FIRST(X_j) **then**

add ε to FIRST($X_1X_2\ldots X_k$)

FOLLOW Function

mansoori@shirazu.ac.ir

FOLLOW(A) = {terminals immediately follow nonterminal A}

FOLLOW(A) =

for all ($B \rightarrow \alpha\, A\, \beta$) $\in P$ **do**

add FIRST(β)\\{ε} to FOLLOW(A)

for all ($B \rightarrow \alpha\, A\, \beta$) $\in P$ and $\varepsilon \in$ FIRST(β) **do**

add FOLLOW(B) to FOLLOW(A)

for all ($B \rightarrow \alpha\, A$) $\in P$ **do**

add FOLLOW(B) to FOLLOW(A)

if A is the start nonterminal **then**

add **$** to FOLLOW($A$)

Example FIRST and FOLLOW

mansoori@shirazu.ac.ir

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

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

FIRST(E') = {+, **ε**} FIRST(T') = {*, **ε**}

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

FOLLOW(T') = FOLLOW(T) = {+,), $}

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

LL(1) Grammar

mansoori@shirazu.ac.ir

- 1^{st} L: scanning the input from left to right
  2^{nd} L: producing a leftmost derivation
  1: using one lookahead token at each step of decision

- A grammar G is LL(1) if it is not left-recursive and for each collection of productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ for nonterminal A, the following holds:

- $\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \varnothing$ for all $i \neq j$
- if $\alpha_i \Rightarrow^* \varepsilon$ then

  $\alpha_j \not\Rightarrow^* \varepsilon$ for all $i \neq j$

  $\mathrm{FIRST}(\alpha_j) \cap \mathrm{FOLLOW}(A) = \varnothing$ for all $i \neq j$

LL(1) Examples

mansoori@shirazu.ac.ir

$$
\begin{array}{lll}
stmt & \rightarrow & \textbf{if} \ (\ expr \ ) \ stmt \ \textbf{else} \ stmt \\
 & | & \textbf{while} \ (\ expr \ ) \ stmt \\
 & | & \{ \ stmt\_list \ \}
\end{array}
$$

Grammar	Not LL(1) because:
$S \rightarrow S\,\mathbf{a} \mid \mathbf{a}$	Is left recursive
$S \rightarrow \mathbf{a}\ S \mid \mathbf{a}$	$\text{FIRST}(\mathbf{a}\ S) \cap \text{FIRST}(\mathbf{a}) = \{\mathbf{a}\} \neq \varnothing$
$S \rightarrow \mathbf{a}\ R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow \mathbf{a}\ R\ \mathbf{a}$ $R \rightarrow S \mid \varepsilon$	For R: $\text{FIRST}(S) \cap \text{FOLLOW}(R) = \{\mathbf{a}\} \neq \varnothing$

Recursive Predictive Parsing

mansoori@shirazu.ac.ir

- Grammar must be LL(1)

- Every nonterminal has one (recursive) procedure responsible for parsing the syntactic category of input tokens corresponding to that nonterminal

- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input lookahead information

Writing a Recursive Predictive Parser

mansoori@shirazu.ac.ir

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{term}\ \mathit{rest} \\
\mathit{rest} &\rightarrow +\ \mathit{term}\ \mathit{rest} \\
&\mid -\ \mathit{term}\ \mathit{rest} \\
&\mid \varepsilon \\
\mathit{term} &\rightarrow \mathbf{id}
\end{aligned}
$$

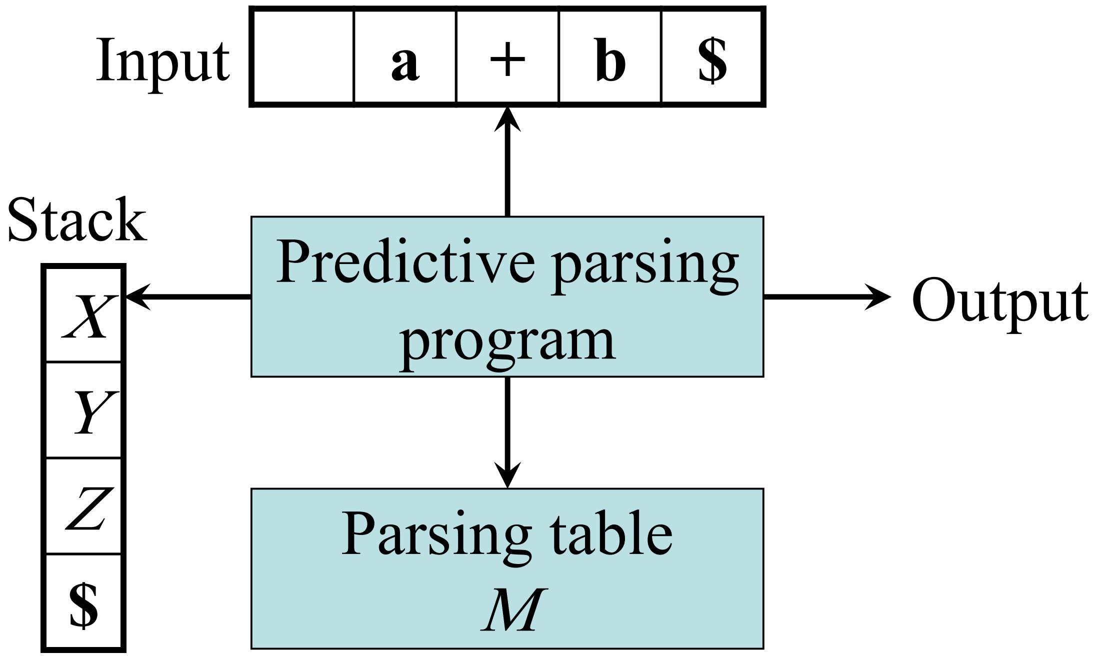
```
void rest() {
    if ( lookahead in FIRST(+ term rest) ) {
        match('+'); term(); rest(); }
    else if ( lookahead in FIRST(- term rest) ) {
        match('-'); term(); rest(); }
    else if ( lookahead in FOLLOW(rest) )
        return;
    else error();
}
```

FIRST(+ *term rest*) = { + }
FIRST(- *term rest*) = { - }
FOLLOW(*rest*) = { **$** }

Non-recursive Predictive Parsing

mansoori@shirazu.ac.ir

- Given an LL(1) grammar $G = (N, T, P, S)$
  – Construct a table $M[A, a]$ for $A \in N$, $a \in T$, and
  – Use a predictive parsing program with a *stack*



Constructing Predictive Parsing Table

mansoori@shirazu.ac.ir

for (each production $A \rightarrow \alpha$) {

for (each terminal $a \in \text{FIRST}(\alpha)$)

add $A \rightarrow \alpha$ to $M[A,a]$

if ($\varepsilon \in \text{FIRST}(\alpha)$) {

for (each terminal $b \in \text{FOLLOW}(A)$)

add $A \rightarrow \alpha$ to $M[A,b]$

if ($\$ \in \text{FOLLOW}(A)$)

add $A \rightarrow \alpha$ to $M[A,\$]$

}

}

Mark each undefined entry in M as error (empty entry)

Example Predictive Parsing Table

$E \rightarrow T\,E'$
$E' \rightarrow +\ T\,E' \mid \varepsilon$
$T \rightarrow F\,T'$
$T' \rightarrow *\ F\,T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \textbf{id}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\,E'$	**(id**	**$)**
$E' \rightarrow +\ T\,E'$	+	**$)**
$E' \rightarrow \varepsilon$	ε	**$)**
$T \rightarrow F\,T'$	**(id**	+ **$)**
$T' \rightarrow *\ F\,T'$	*	+ **$)**
$T' \rightarrow \varepsilon$	ε	+ **$)**
$F \rightarrow (\ E\ )$	**(**	* + **$)**
$F \rightarrow \textbf{id}$	**id**	* + **$)**

	id	+	*	**(**	**)**	**$**
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$		
E'		$E' \rightarrow +\ T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$			$T \rightarrow F\,T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *\ F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (\ E\ )$		

mansoori@shirazu.ac.ir

Example Predictive Parsing Table

mansoori@shirazu.ac.ir

Ambiguous grammar:

$S \to \mathbf{i}\ E\ \mathbf{t}\ S\ S' \mid \mathbf{a}$
$S' \to \mathbf{e}\ S \mid \varepsilon$
$E \to \mathbf{b}$

$A \to \alpha$	FIRST(α)	FOLLOW(A)
$S \to \mathbf{i}\ E\ \mathbf{t}\ S\ S'$	$\mathbf{i}$	$\mathbf{e}$ \$
$S \to \mathbf{a}$	$\mathbf{a}$	$\mathbf{e}$ \$
$S' \to \mathbf{e}\ S$	$\mathbf{e}$	$\mathbf{e}$ \$
$S' \to \varepsilon$	ε	$\mathbf{e}$ \$
$E \to \mathbf{b}$	$\mathbf{b}$	$\mathbf{t}$

	a	b	e	i	t	\$
S	$S \to \mathbf{a}$			$S \to \mathbf{i}\ E\ \mathbf{t}\ S\ S'$		
S'			$S' \to \varepsilon$ $S' \to \mathbf{e}\ S$			$S' \to \varepsilon$
E		$E \to \mathbf{b}$				

Error: duplicate table entry

Predictive Parsing Program

mansoori@shirazu.ac.ir

push(\$); // stack bottom-marker

push(S); // grammar start nonterminal

a = lookahead; X = pop();

while ($X \neq \$$) { // stack is not empty

    if (X is a terminal)

        if ($X = a$) { a = next lookahead; } else error();

    else if ($M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$) {

        output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$

        push($Y_k, Y_{k-1}, \ldots, Y_2, Y_1$) // Y_1 on top

    }

    else error();

    X = pop();

}

if ($a \neq \$$) error();

Example Table-Driven Parsing

mansoori@shirazu.ac.ir

STACK	INPUT	ACTION
$E\$$	**id**+**id*****id**$	output $E \to T\,E'$
$T\,E'\$$	**id**+**id*****id**$	output $T \to F\,T'$
$F\,T'\,E'\$$	**id**+**id*****id**$	output $F \to$ **id**
id $T'\,E'\$$	**id**+**id*****id**$	match **id**
$T'\,E'\$$	+**id*****id**$	output $T' \to \varepsilon$
$E'\$$	+**id*****id**$	output $E' \to +\,T\,E'$
$+\,T\,E'\$$	+**id*****id**$	match +
$T\,E'\$$	**id*****id**$	output $T \to F\,T'$
$F\,T'\,E'\$$	**id*****id**$	output $F \to$ **id**
id $T'\,E'\$$	**id*****id**$	match **id**
$T'\,E'\$$	***id**$	output $T' \to *\,F\,T'$
$*\,F\,T'\,E'\$$	***id**$	match *
$F\,T'\,E'\$$	**id**$	output $F \to$ **id**
id $T'\,E'\$$	**id**$	match **id**
$T'\,E'\$$	$	output $T' \to \varepsilon$
$E'\$$	$	output $E' \to \varepsilon$
$	$	

	id	+	*	(	)	$
E	$E \to T\,E'$			$E \to T\,E'$		
E'		$E' \to +\,T\,E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F\,T'$			$T \to F\,T'$		
T'		$T' \to \varepsilon$	$T' \to *\,F\,T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**			$F \to (\,E\,)$		

Panic Mode Error Recovery

mansoori@shirazu.ac.ir

Add synchronizing actions to undefined entries based on FOLLOW

Pro: can be automated
Cons: error messages are needed

FOLLOW(E) = {), \$ }
FOLLOW(E') = {), \$ }
FOLLOW(T) = { +,), \$ }
FOLLOW(T') = { +,), \$ }
FOLLOW(F) = { +, *,), \$ }

	id	**+**	*****	**(**	**)**	**\$**
E	$E \to T E'$			$E \to T E'$	***synch***	***synch***
E'		$E' \to + T E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F T'$	***synch***		$T \to F T'$	***synch***	***synch***
T'		$T' \to \varepsilon$	$T' \to * F T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**	***synch***	***synch***	$F \to (E)$	***synch***	***synch***

empty $M[A,a]$: parsing program skips current token a

synch in $M[A,a]$: parsing program pops current nonterminal A

Example Panic Mode Error Recovery

mansoori@shirazu.ac.ir

STACK	INPUT	ACTION
$E\,\$$	$+\mathbf{id}*+\mathbf{idid}\$$	error, skip +
$E\,\$$	$\mathbf{id}*+\mathbf{idid}\$$	
$T\,E'\$$	$\mathbf{id}*+\mathbf{idid}\$$	
$F\,T'\,E'\$$	$\mathbf{id}*+\mathbf{idid}\$$	
$\mathbf{id}\;T'\,E'\$$	$\mathbf{id}*+\mathbf{idid}\$$	
$T'\,E'\$$	$*+\mathbf{idid}\$$	
$*\,F\,T'\,E'\$$	$*+\mathbf{idid}\$$	
$F\,T'\,E'\$$	$+\mathbf{idid}\$$	error, pop F
$T'\,E'\$$	$+\mathbf{idid}\$$	
$E'\$$	$+\mathbf{idid}\$$	
$+\;T\,E'\$$	$+\mathbf{idid}\$$	
$T\,E'\$$	$\mathbf{idid}\$$	
$F\,T'\,E'\$$	$\mathbf{idid}\$$	
$\mathbf{id}\;T'\,E'\$$	$\mathbf{idid}\$$	
$T'\,E'\$$	$\mathbf{id}\$$	error, skip **id**
$T'\,E'\$$	$\$$	
$E'\$$	$\$$	
$\$$	$\$$	

	id	+	*	(	)	$
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$	synch	synch
E'		$E' \rightarrow +\,T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$	synch		$T \rightarrow F\,T'$	synch	synch
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *\,F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$	synch	synch	$F \rightarrow (\,E\,)$	synch	synch

Phrase-Level Error Recovery

mansoori@shirazu.ac.ir

Change input stream by inserting missing tokens

For example: **id id** is changed into **id * id**

Can then continue here

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T E_R$			$E \to T E'$		
E'		$E' \to + T E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F T'$			$T \to F T'$		
T'	***insert ****	$T' \to \varepsilon$	$T' \to * F T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**			$F \to (E)$		

insert *: parsing program inserts missing * and retries the production

Pro: can be automated
Cons: recovery not always intuitive

Error Productions for Error Recovery

mansoori@shirazu.ac.ir

Add *error production*: $T' \rightarrow F\ T'$ to ignore missing *, e.g.: **id id**

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E'$		
E'		$E' \rightarrow +\ T\ E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T'	$T' \rightarrow F\ T'$	$T' \rightarrow \varepsilon$	$T' \rightarrow *\ F\ T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (\ E\ )$		

Pro: powerful recovery method
Cons: cannot be automated