

Compilers

Lecture IV—Part 1: Syntactic Analysis

Alessandro Artale

Faculty of Computer Science – Free University of Bolzano

POS Building – Room: 2.03

`artale@inf.unibz.it`

`http://www.inf.unibz.it/~artale/`

Summary of Lecture IV—Part 1

- **Intro to Syntactic Analysis**
- Generating Languages from Grammars
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - Predictive Parsers
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

Intro to Syntactic Analysis

- Every programming language has rules that describe the syntactic structure of *well-formed* programs.
- Context-Free Grammars (or BNF) are used to describe the syntax of programs.
 - **Remark:** Regular Grammars/Expressions are not expressive enough to describe the structure of programs, e.g. a RE cannot recognize balanced open and closed parentheses (since they cannot count).
- From certain classes of grammars we can automatically construct a Parser.
- Imposing a structure to a program is useful for the subsequent translation.
- New programming constructs can be easily added for languages based on grammars.

The Role of the Parser

- The **Parser** stands to a CFG as an Automaton stands to a RE.
- The Parser obtains a sequence of Tokens from the lexical analyzer and verifies that the sequence can be generated by means of a *Derivation* in the CFG of the source program.
- As a result the parser output a (representation of a) **Parse-Tree**.
- Parsers are classified as **Bottom-UP** or **Top-Down** depending whether the parse-tree is built from the leaves or from the root, respectively.

Syntax Error Handling

- Many of the errors are syntactic in nature: much of the error detection and recovery is done during parsing.
- The techniques used to handle errors can vary depending from the compiler design.
- In general, the error handler in a Parser should:
 - Report the presence of errors clearly and accurately;
 - Try to “recover” to be able to detect further errors;
 - It should not slow down the processing of correct programs.

Summary

- Intro to Syntactic Analysis
- **Generating Languages from Grammars**
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - Predictive Parsers
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

Notion of Derivation

- To characterize a Language starting from a Grammar we need to introduce the notion of **Derivation**.
- The notion of Derivation uses Productions to generate a string starting from another string.
- **Direct Derivation (in symbols \Rightarrow)**.

If $\alpha \rightarrow \beta \in P$ and $\gamma, \delta \in V^*$, then, $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$.

- **Derivation (in symbols \Rightarrow^*)**.

If $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \dots, \alpha_{n-1} \Rightarrow \alpha_n$, then, $\alpha_1 \Rightarrow^* \alpha_n$.

Generating Languages from Grammars

Generative Definition of a Language. We say that a Language L is *generated* by the *Grammar* G , in symbols $L(G)$, if:

$$L(G) = \{w \in V_T^* \mid S \Rightarrow^* w\}.$$

Example. Consider the following Grammar for arithmetic expressions:

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

The sequence of Tokens $-(\text{id} + \text{id})$ is a well-formed sentence:

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

Note: Token is a synonym of Terminal Symbol when talking of Grammars for programming languages.

Choices in Derivations

- At each step in a Derivation there are two choices to be made:

1. **Which non-terminal to replace;**
2. **Which Production to use for that non-terminal.**

- W.r.t. point 1. the previous derivation:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id}).$$

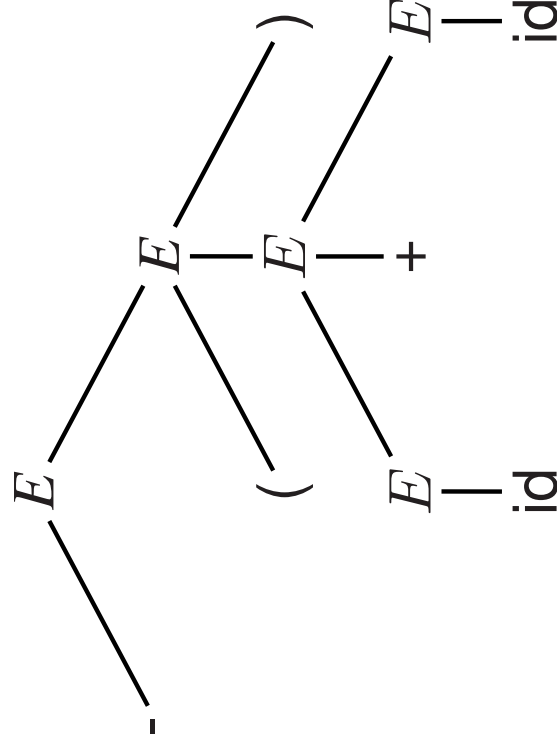
could be:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id}).$$

- **A Parser will consider *Leftmost Derivations*—the leftmost non-terminal is chosen—or *Rightmost Derivations*.**

Parse-Trees and Derivations

- A Parse-Tree is a visualization of a Derivation that ignores variations in the order in which non-terminal are replaced.
- The Parse-Tree associated to the two Derivations in the previous slide is



- Every Parse-Tree is associated with a *unique* leftmost and a *unique* rightmost derivation.
- **Problem of Ambiguity: A sentence can have more than one Parse-Tree.**
Related to point 2 above: **Which Production to use for a given non-terminal**

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- **Ambiguous Grammars**
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - Predictive Parsers
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

Ambiguity

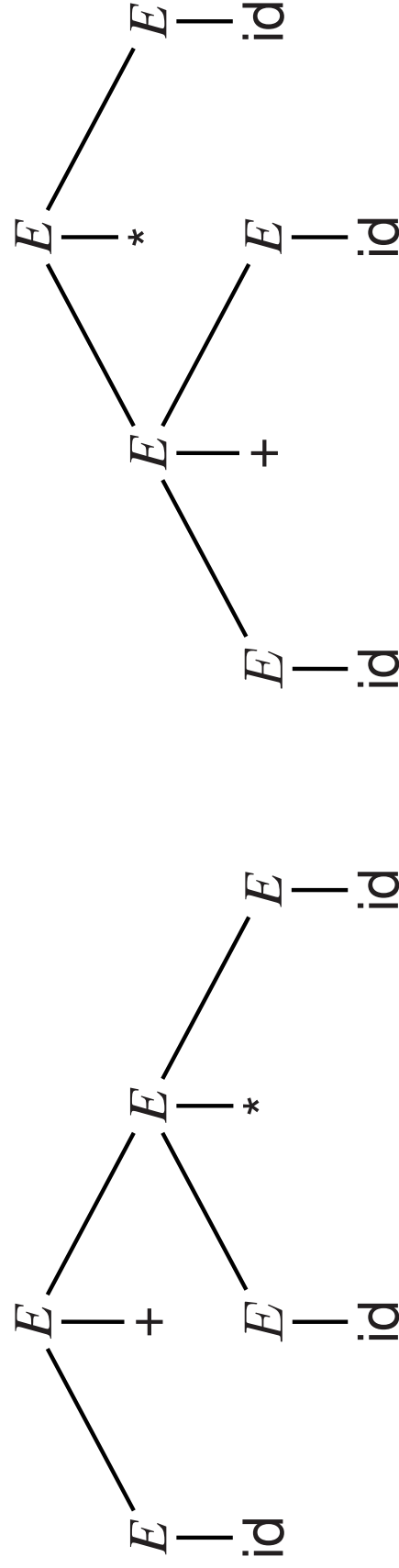
- A grammar is ambiguous if it has more than one Parse-Tree for some string.
 - Equivalently, there is more than one right-most or left-most derivation for some string.
- Ambiguity is bad: Leaves meaning of some programs ill-defined since we cannot decide its syntactical structure uniquely.
- Two alternative solutions:
 1. *Disambiguate the grammar*
 2. *Use extra-grammatical mechanisms, like disambiguating rules, to discard alternative Parse-Trees.*

Ambiguity: Arithmetic Expressions

Consider the Grammar for arithmetic expressions:

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$$

The sequence of Tokens $id + id * id$ has two Parse-Trees



The first Parse-Tree reflects the usual assumption that $*$ takes precedence on $+$.

Eliminating Ambiguity by Disambiguating the Grammar

- Sometime it is possible to eliminate ambiguity by rewriting the Grammar.
- **Example.** Let us rewrite the Grammar for arithmetic expressions:

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

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

- Enforces precedence of * over +;
- Enforces left-associativity of + and *

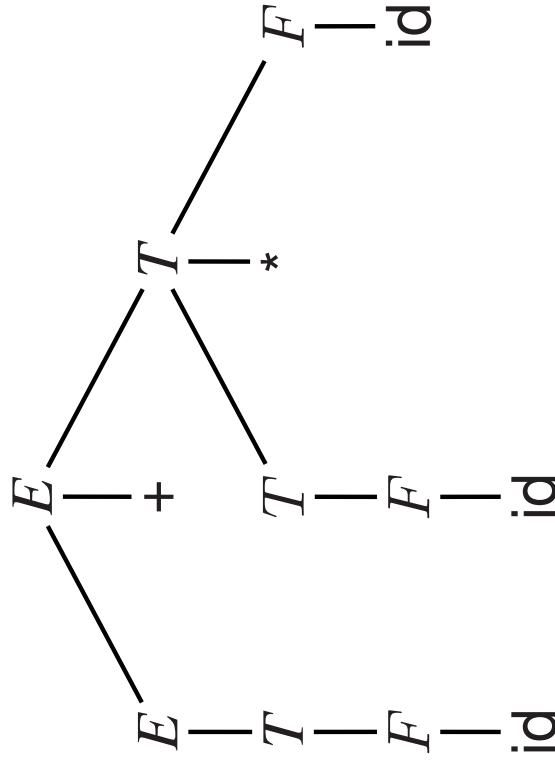
Eliminating Ambiguity: Example

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

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

The sequence of Tokens $\text{id} + \text{id} * \text{id}$ has now only one Parse-Tree



Ambiguity: The Dangling Else

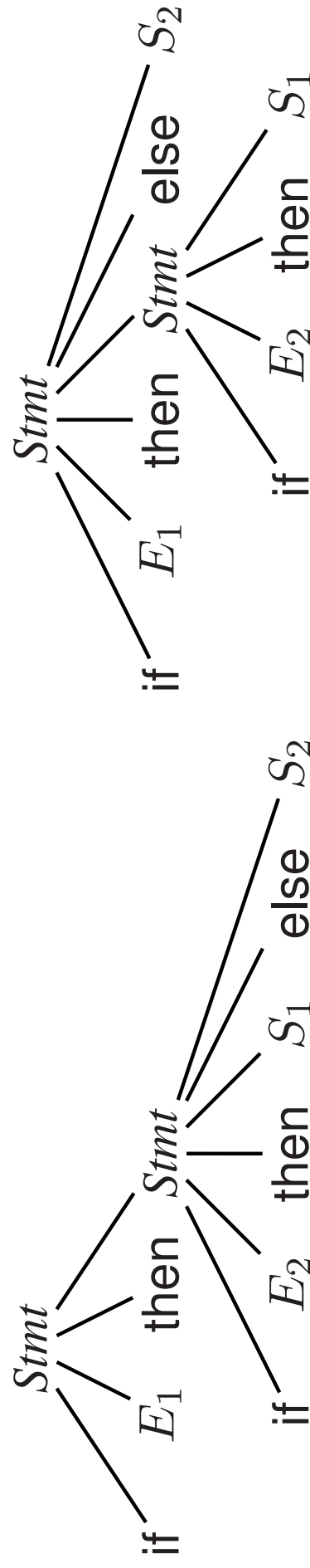
- Consider the Grammar for if-then-else statements:

$$\begin{array}{l} Stmt \rightarrow \text{if } Expr \text{ then } Stmt \\ \quad | \text{if } Expr \text{ then } Stmt \text{ else } Stmt \\ \quad | \text{other} \end{array}$$

- This Grammar is ambiguous.
- **Example.** Consider the statement:
if E_1 then if E_2 then S_1 else S_2 .

The Dangling Else: Example

The statement: `if E_1 then if E_2 then S_1 else S_2` , has two Parse-Trees



- Typically, the first Parse-Tree is preferred.
- **Disambiguating Rule: Match each else with the closest unmatched then.**

Disambiguating Dangling Else

- **Disambiguating Rule: Match each else with the closest unmatched then.**
- The rule can be incorporated into the Grammar if we distinguish between *matched* and *unmatched* statements.
- A statement between a then-else must be *matched*.

$$Stmt \rightarrow Matched_stmt \mid Unmatched_stmt$$

$$Matched_stmt \rightarrow \text{if } Expr \text{ then } Matched_stmt \text{ else } Matched_stmt \\ \mid \text{other}$$

$$Unmatched_stmt \rightarrow \text{if } Expr \text{ then } Stmt$$

$$\mid \text{if } Expr \text{ then } Matched_stmt \text{ else } Unmatched_stmt$$

- This Grammar generates the same set of strings as the previous one but gives just one Parse-Tree for if-then-else statements.

Disambiguating Rules: Precedence and Associativity Declarations

- Instead of rewriting the Grammar:
 1. Use the more natural (ambiguous) Grammar;
 2. Along with disambiguating declarations.
- Most tools (e.g. YACC) allow *precedence* and *associativity* declarations for terminals (e.g. “*” takes precedence over “+”) to disambiguate grammars (see the Book, Sections 4.8-4.9, for more details).

Ambiguity: Summary

- No general techniques for handling ambiguity.
- Impossible to convert automatically an ambiguous Grammar to an unambiguous one.
- Used with care, ambiguity can simplify the Grammar
 - Sometimes ambiguous Grammars allow for more natural definitions
 - But then we need extra-grammatical disambiguation mechanisms.

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- **Top-Down Parsers**
 - Problems: Backtrack and Infinite Loops
 - Predictive Parsers
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

Top-Down Parsing

- The **Top-Down Parsing**—also called *Recursive-Descent Parsing*—builds the Parse-Tree by starting with the root, labeled with the scope, and performing the following two steps:
 1. Select a node labeled with a Non-Terminal, say A ;
 2. Select one production for A and generate as many children of A as symbols on the right-hand side of the production;
- This procedure ends either when all the leaves are labeled with Tokens or we can not apply any production.
- **Note.** To solve point (1), top-down parsers proceed by **left-most derivations**.

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- **Top-Down Parsers**
 - **Problems: Backtrack and Infinite Loops**
 - Predictive Parsers
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

Backtrack in Top-Down Parsing

- In general, the selection of a production for a Non-Terminal can involve **Backtrack: We may need to select another production if the first fails.**
- **Note 1:** A production *fails* if the Parse-Tree can not be completed to match the input string.
- **Note 2:** Backtrack can happen even if the Grammar is not ambiguous.
- **Note 3:** Backtracking is rarely needed to parse programming languages.

Backtrack in Top-Down Parsing: An Example

- Consider the (non ambiguous) Grammar for arithmetic expressions:

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{id} * T \mid \text{id} \mid (E)$$

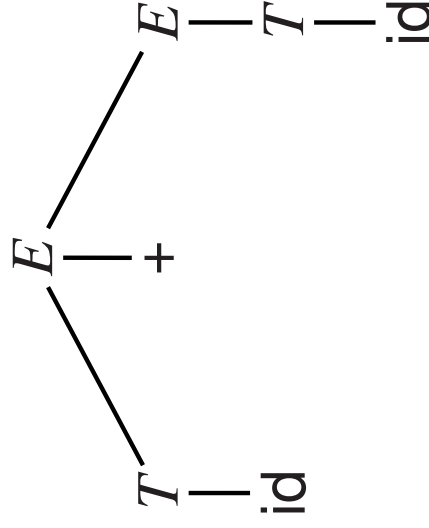
and the input Tokens sequence: `id + id`.

1. Start with the Non-Terminal E as the root of the Parse-Tree.
2. Use the production $E \rightarrow T + E$;
3. Use the production $T \rightarrow \text{id} * T$:
`id` matches the first input Token but `*` does not!
4. **Backtrack:** Use the production $T \rightarrow \text{id}$:
The Token `id` does match! Also the Token `+` matches!!

Backtrack in Top-Down Parsing: An Example (Cont.)

5. We need now to choose a production for the second E : if we still choose $E \rightarrow T+E$ we fail; then we **Backtrack** and choose $E \rightarrow T$.
6. Follow the same step as before (Step 4.) for T and we succeed with $T \rightarrow \text{id}$:
The last Token matches!!

7. The successful Parse-Tree is:



Loops in Top-Down Parsing

- It is possible for a Recursive-Descent Parser to loop forever!
- Since top-down parsers proceed along left-most derivations, looping arises with **Left-Recursive Grammars**.
- **Definition.** A Grammar is said *Left-Recursive* if for a Terminal, A , there is a Derivation, $A \Rightarrow^* A\alpha$, for some $\alpha \in V^*$.
- **Example.** The Grammar with production $E \rightarrow E+T$ is left-recursive.
- **Eliminating Immediate Left Recursion.** If we have a production of the form, $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$, where β_1, \dots, β_m do not begin with A , then an equivalent right-recursive Grammar is:

$$A \rightarrow \beta_1 R \mid \dots \mid \beta_m R$$

$$R \rightarrow \alpha_1 R \mid \dots \mid \alpha_n R \mid \epsilon$$

- In general, even non-immediate left-recursion can be eliminated (see the Book, Section 4.3.3, for more details).

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - **Predictive Parsers**
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

Predictive Parsers

- **Predictive Parsers** avoid backtracking since they can *predict* which production to use by looking at the *current Token* being scanned in the input—called **Lookahead** symbol.
- The Lookahead symbol unambiguously determines which production to use.
- **Example.** Given the following productions:

$$\begin{array}{l} Stmt \rightarrow \text{if } Expr \text{ then } Stmt \text{ else } Stmt \\ \quad | \text{ while } Expr \text{ do } Stmt \\ \quad | \text{ begin } Stmt_list \text{ end} \end{array}$$

Then, depending whether the Lookahead is if, while or begin the Parser will be forced to use just one of the above productions.

Predictive Parsers (Cont.)

- **Criterion for the selection of a production.**
 - If the right side of a production starts with a token then it will be used if such token matches the Lookahead symbol;
 - If the right side of a production starts with a non-terminal then it will be used if the Lookahead symbol can be *generated* from the non-terminal.
- **Predictive Parsing relies on information about what *first* symbol can be generated by the right side of a production.**

- **Definition (FIRST).** Let α be the right side of a production for non-terminal A . Then, $\text{FIRST}(\alpha)$ is the set of Tokens that start a string generated by α :

$$\forall a \in V_T. a \in \text{FIRST}(\alpha) \text{ iff } \alpha \Rightarrow^* a\beta, \text{ with } \alpha, \beta \in V^*.$$

Furthermore, if $\alpha = \epsilon$ or $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$.

Predictive Parsers (Cont.)

- A Predictive Parser decides between two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ by considering the Lookahead symbol;
- If the Lookahead symbol is in $\text{FIRST}(\alpha)$ then $A \rightarrow \alpha$ is used.
- To use a predictive parser it is *necessary* that $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$, for all α, β right side of *alternative* productions—i.e., productions associated to the same non-terminal.
- **Note.** As will be clear in the following slides, the above condition is necessary but not sufficient.

Left-Factoring

- A Grammar must be **Left-Factored** before use for predictive parsing.
- **Main Idea:** If it is not clear which alternative production to use we rewrite the productions to defer the decision until we see enough input to be able to decide.
- **Example.** Consider the following productions:

$$\begin{array}{l}
 Stmt \rightarrow \text{if } Expr \text{ then } Stmt \text{ else } Stmt \\
 \quad \quad \quad | \quad \text{if } Expr \text{ then } Stmt
 \end{array}$$

Then, on seeing the Token if we cannot decide between the two productions above. Left-Factored, this Grammar becomes:

$$\begin{array}{l}
 Stmt \rightarrow \text{if } Expr \text{ then } Stmt \text{ } Stmt' \\
 Stmt' \rightarrow \text{else } Stmt \mid \epsilon
 \end{array}$$

- **Remark.** The left-factored Grammar is still ambiguous: on input `else` it is not clear what production to use for `Stmt'`.

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - **Predictive Parsers**
 - * **LL(k) Grammars**
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables

LL(k) Grammars

- **Predictive parsers accept LL(k) Grammars.**
 - The first L means “left-to-right” scanning of the input;
 - The second L stands for producing a “leftmost derivation”;
 - k means “predict” based on k tokens of lookahead.
- In practice, LL(1) Grammars are used.

LL(1) Grammars

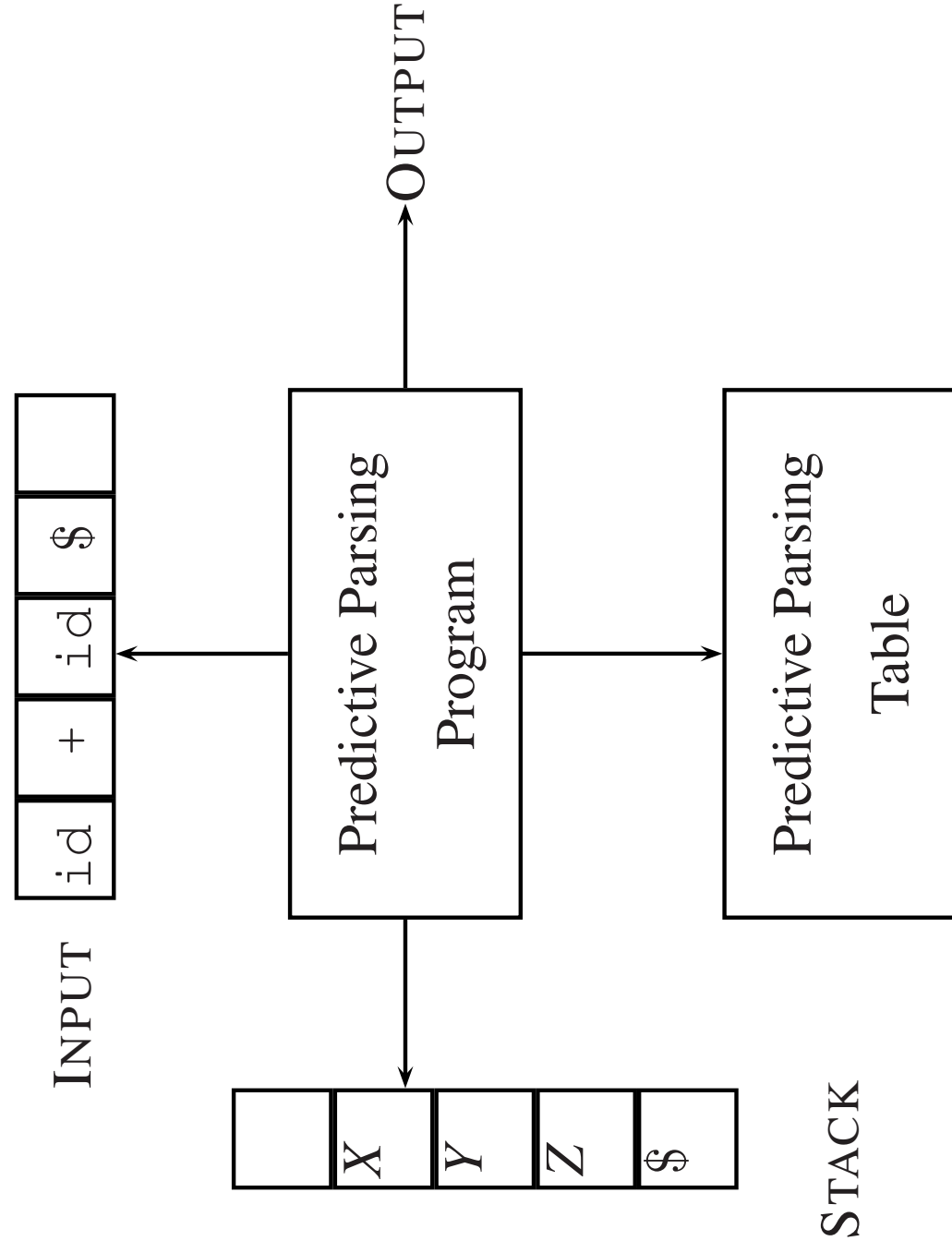
- Given an LL(1) Grammar, then for each non-terminal and Token there is only one production that could lead to success.
- Predictive parsers built on top of LL(1) Grammars can be specified as a two-dimensional table—called the *Parsing Table*, with:
 - One dimension for current non-terminal to expand;
 - One dimension for next Token;
 - Each table entry contains one production.

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - **Predictive Parsers**
 - * LL(k) Grammars
 - * **Predictive Parser Program**
 - * Constructing Predictive Parsing Tables

Predictive Parser Architecture

- A table-driven predictive parser has: an input buffer (Tokens returned from the Lexical Analyzer), a stack (containing Grammar symbols), and a parsing table.



Predictive Parser Program

- We use a stack to keep track of pending non-terminals.
- Initially the stack contains $\$S$ with S , the scope of the Grammar, on top.
- Now, let X the symbol on top of the Stack and a the current Token in the input. There are three possibilities:
 1. If $X = a = \$$, the parser halts successfully;
 2. If $X = a \neq \$$, the parser pops X from the stack and advances the input pointer;
 3. If X is a non-terminal, the parser checks the parser table $M[X, a]$:
 - If $M[X, a] = \text{error}$, then an error recovery is done;
 - If $M[X, a] = \{X \rightarrow UVW\}$, then the parser replaces X on top of the stack by UVW (with U on top for leftmost derivations).

Predictive Parser Program: An Example

- Consider the following LL(1) Grammar, obtained by eliminating the left recursion from the Grammar for arithmetics saw in slide (14):

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Predictive Parser Program: An Example (Cont.)

A predictive table for the Grammar is:

	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'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Where empty entries indicate an error situation.

Predictive Parser Program: An Example (Cont.)

The moves with input $\text{id} + \text{id} * \text{id}$ are:

<i>Stack</i>	<i>Input</i>	<i>Output</i>	<i>Stack</i>	<i>Input</i>	<i>Output</i>
$\$E$	$\text{id} + \text{id} * \text{id}\$$		$\$E'T'\text{id}$	$\text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T$	$\text{id} + \text{id} * \text{id}\$$	$E \rightarrow TE'$	$\$E'T'$	$*\text{id}\$$	
$\$E'T'F$	$\text{id} + \text{id} * \text{id}\$$	$T \rightarrow FT'$	$\$E'T'F*$	$*\text{id}\$$	$T' \rightarrow *FT'$
$\$E'T'\text{id}$	$\text{id} + \text{id} * \text{id}\$$	$F \rightarrow \text{id}$	$\$E'T'F$	$\text{id}\$$	
$\$E'T'$	$+\text{id} * \text{id}\$$		$\$E'T'\text{id}$	$\text{id}\$$	$F \rightarrow \text{id}$
$\$E'$	$+\text{id} * \text{id}\$$	$T' \rightarrow \epsilon$	$\$E'T'$	$\$\$$	
$\$E'T+$	$+\text{id} * \text{id}\$$	$E' \rightarrow +TE'$	$\$E'$	$\$\$$	$T' \rightarrow \epsilon$
$\$E'T$	$\text{id} * \text{id}\$$		$\$\$$	$\$\$$	$E' \rightarrow \epsilon$
$\$E'T'F$	$\text{id} * \text{id}\$$	$T \rightarrow FT'$	$\$\$$	$\$\$$	ACCEPT

Summary

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - **Predictive Parsers**
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * **Constructing Predictive Parsing Tables**

Constructing Predictive Parsing Tables

- To build parsing tables we make use of two functions: FIRST and FOLLOW.

Definition (FIRST). Let α be the right side of a production for non-terminal

A . Then, $\text{FIRST}(\alpha)$ is the set of Tokens that start a string generated by α :

$$\forall a \in V_T. a \in \text{FIRST}(\alpha) \text{ iff } \alpha \Rightarrow^* a\beta, \text{ with } \alpha, \beta \in V^*.$$

Furthermore, if $\alpha = \epsilon$ or $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$.

- Given a non-terminal A , $\text{FOLLOW}(A)$ is the set of Tokens, \mathbf{a} , that can appear immediately to the right of A in some sentential form.

Definition (FOLLOW).

$$\text{FOLLOW}(A) = \{\mathbf{a} \in V_T \mid S \Rightarrow^* \alpha A \mathbf{a} \beta, \text{ with } \alpha, \beta \in V^*\}.$$

- **Note.** There may have been symbols between A and \mathbf{a} , but they derived ϵ and disappeared.

Computing FIRST

$\text{FIRST}(X)$, with $X \in V$: Apply the following rules until no more terminals (or ϵ) can be added:

1. If $X \in V_T$, then $\text{FIRST}(X) = \{X\}$.
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.
3. If $X \rightarrow Y_1, Y_2, \dots, Y_k$ is a production then:
 - Add a to $\text{FIRST}(X)$ if for some i we have:
 - $a \in \text{FIRST}(Y_i)$, and $\epsilon \in \text{FIRST}(Y_1) \cap \dots \cap \text{FIRST}(Y_{i-1})$.
 - If $\epsilon \in \text{FIRST}(Y_j)$, for all $j = 1, \dots, k$, then add ϵ to $\text{FIRST}(X)$.

Note 1. $\text{FIRST}(Y_1) \setminus \{\epsilon\} \subseteq \text{FIRST}(X)$.

Note 2. If $\epsilon \notin \text{FIRST}(Y_1)$, then we add nothing more to $\text{FIRST}(X)$.

Computing FIRST (Cont.)

Given any sequence $X_1 X_2 \dots X_n \in V^*$, we compute $\text{FIRST}(X_1 X_2 \dots X_n)$:

1. Add $\text{FIRST}(X_1) - \{\epsilon\}$ to $\text{FIRST}(X_1 X_2 \dots X_n)$.
2. If $\epsilon \in \text{FIRST}(X_1)$, then:
Add $\text{FIRST}(X_2) - \{\epsilon\}$ to $\text{FIRST}(X_1 X_2 \dots X_n)$; otherwise Stop.
3. If $\epsilon \in \text{FIRST}(X_2)$, then:
Add $\text{FIRST}(X_3) - \{\epsilon\}$ to $\text{FIRST}(X_1 X_2 \dots X_n)$; otherwise Stop.
4. ...
5. If $\epsilon \in \text{FIRST}(X_j)$, for all $j = 1, \dots, n$, then add ϵ to $\text{FIRST}(X_1 X_2 \dots X_n)$.

Computing FOLLOW

Definition. $\text{FOLLOW}(A) = \{a \in V_T \mid S \Rightarrow^* \alpha A a \beta, \text{ with } \alpha, \beta \in V^*\}$.

$\text{FOLLOW}(A)$: For each non-terminal, A , apply the following rules until nothing more can be added:

1. Add $\$$ to $\text{FOLLOW}(S)$ (if S is the scope, and $\$$ is the input right endmarker);
2. For all productions $Y \rightarrow \alpha \mathbf{A} Y_1 \dots Y_n$ (where, $\alpha \in V^*$ and $Y_i \in V$):
 - (a) Add $\text{FIRST}(Y_1) - \{\epsilon\}$ to $\text{FOLLOW}(A)$.
 - (b) If $\epsilon \in \text{FIRST}(Y_1)$, then:
Add $\text{FIRST}(Y_2) - \{\epsilon\}$ to $\text{FOLLOW}(A)$; otherwise Stop.
 - (c) ...
 - (d) If $\epsilon \in \text{FIRST}(Y_{n-1})$, then:
Add $\text{FIRST}(Y_n) - \{\epsilon\}$ to $\text{FOLLOW}(A)$; otherwise Stop.
 - (e) If $\epsilon \in \text{FIRST}(Y_n)$, then:
Add $\text{FOLLOW}(Y)$ to $\text{FOLLOW}(A)$.

FIRST and FOLLOW: An Example

Consider the Grammar for arithmetic expression:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}.$$

$$\text{FIRST}(E') = \{ +, \epsilon \}.$$

$$\text{FIRST}(T') = \{ *, \epsilon \}.$$

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

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

$$\text{FOLLOW}(F) = \{ +, *, \}, \$ \}.$$

Constructing Predictive Parsing Tables (Cont.)

- The production $A \rightarrow \alpha$ with \mathbf{a} in $\text{FIRST}(\alpha)$ is used if \mathbf{a} is the lookahead symbol.
- **Problem.** When $\epsilon \in \text{FIRST}(\alpha)$. Then, the production $A \rightarrow \alpha$ is used if the lookahead is in $\text{FOLLOW}(A)$.

Constructing Predictive Parsing Tables (Cont.)

Algorithm. *Input:* Grammar, G . *Output:* Parsing Table, M .

1. For each production $A \rightarrow \alpha$ in G do:
 - (a) For each terminal \mathbf{a} in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, \mathbf{a}]$.
 - (b) If $\epsilon \in \text{FIRST}(\alpha)$, then for each terminal \mathbf{a} in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \mathbf{a}]$.
 - (c) If $\epsilon \in \text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
2. Make each undefined entry an **error**.

LL(1) Grammars: Final Remarks

- While a parsing table, M , can be constructed for every Grammar, for some Grammar M may have *multiple entries*.
- **Definition. A Grammar whose parsing table has no multiple entries is said to be LL(1).**
- The Grammar for arithmetic expression is LL(1).
- The left-factored Grammar for `if-then-else` is not LL(1): **Ambiguous Grammars are never LL(1).**
- Even the non ambiguous Grammar for `if-then-else` (the one with Matched Vs. Unmatched statements) is not LL(1).
- **General Remark:** There are no universal rules by which a Grammar can be reduced to be LL(1)!!
- We need more powerful parsing techniques than predictive parsers.

Summary of Lecture IV—Part 1

- Intro to Syntactic Analysis
- Generating Languages from Grammars
- Ambiguous Grammars
- Top-Down Parsers
 - Problems: Backtrack and Infinite Loops
 - Predictive Parsers
 - * LL(k) Grammars
 - * Predictive Parser Program
 - * Constructing Predictive Parsing Tables