Formal Languages and Compilers
Lecture VI—Part 1: Syntactic Analysis

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

2020/21 – Second Semester

Summary of Lecture VI—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 arbitrary 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 $\alpha \to \beta \in \mathbf{P}$ and $\gamma, \delta \in \mathbf{V}^*$, then, $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$.
- **Derivation (in symbols $\Rightarrow^*$).**
  If $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{n-1} \Rightarrow \alpha_n$, 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**$= (\mathbf{V}_T, \mathbf{V}_N, \mathbf{S}, \mathbf{P})$, in symbols L(G), if:

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

Example. Consider the following CF Grammar for arithmetic expressions:

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

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

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

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

Derivation Trees for Context-Free Grammars

- Derivation Trees, called also **Parse Trees**, are a visual method of describing any derivation in a context-free grammar.
- Let **G**= (**V**$_T$,**V**$_N$,**S**,**P**) be a CFG. A tree is a *derivation tree* for **G** if:
  1. Every node has a *label*, which is a symbol of **V**;
  2. The label of the root is **S**;
  3. If a node, n, labeled with A has at least one descendant, then A must be in $\mathbf{V}_N$;
  4. If nodes $n_1, n_2, \ldots, n_k$ are direct descendants of node n, with labels $A_1, A_2, \ldots, A_k$, respectively, then:
  $$A \to A_1, A_2, \ldots, A_k$$
  must be a production in **P**.

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), we have two derivations for $-(\text{id} + \text{id})$:

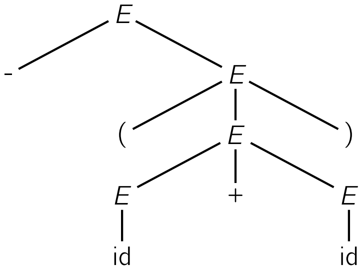
$$1. \quad E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$

$$2. \quad E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$$

- A Parser will consider either *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—point (1) above.
- 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, and viceversa.
- 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

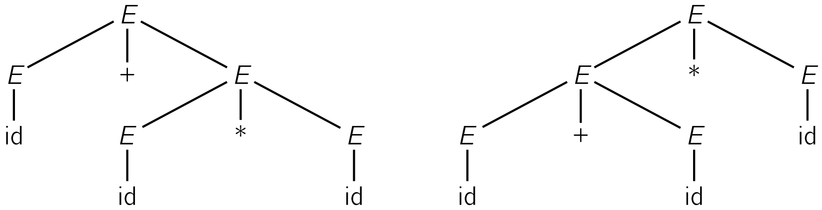
Ambiguity

- 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.
- Ambiguity is a property of Grammars, not of Languages.
- Two alternative solutions:
    1. *Disambiguate* the grammar
    2. Use extra-grammatical mechanisms, like *disambiguating rules*, to discard alternative Parse-Trees.

Consider the Grammar for arithmetic expressions:

$$E \to E + E \mid E * E \mid (E) \mid -E \mid \mathrm{id}$$

The sequence of Tokens $\mathrm{id} + \mathrm{id} * \mathrm{id}$ 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:

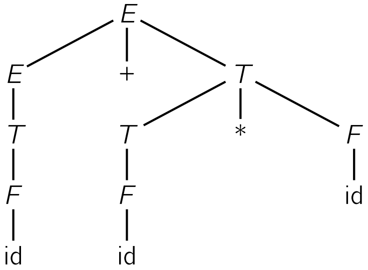
$$
\begin{array}{lcl}
E & \rightarrow & E+T \mid T \\
T & \rightarrow & T*F \mid F \\
F & \rightarrow & (E) \mid \mathrm{id}
\end{array}
$$

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

Eliminating Ambiguity: Example

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

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



Ambiguity: The Dangling Else

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

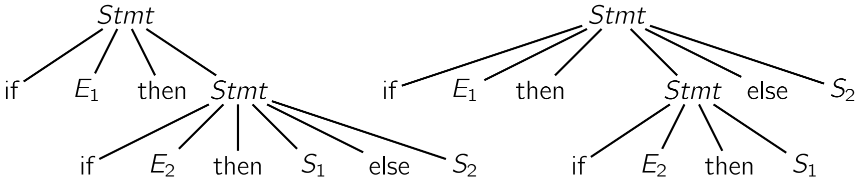
$$
\begin{array}{lcl}
Stmt & \rightarrow & \text{if } \; Expr \; \text{ then } \; Stmt \\
 & | & \text{if } \; Expr \; \text{ then } \; Stmt \; \text{ else } \; Stmt \\
 & | & \text{other}
\end{array}
$$

- 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.

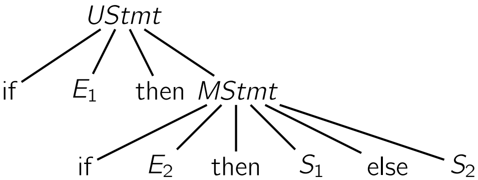
Disambiguating Dangling Else

- **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*.

$$
\begin{array}{rcl}
\texttt{Stmt} & \rightarrow & \texttt{Matched\_stmt} \mid \texttt{Unmatched\_stmt} \\
\texttt{Matched\_stmt} & \rightarrow & \text{if } \texttt{Expr} \text{ then } \texttt{Matched\_stmt} \text{ else } \texttt{Matched\_stmt} \\
 & \mid & \texttt{Other-Stmt} \\
\texttt{Unmatched\_stmt} & \rightarrow & \text{if } \texttt{Expr} \text{ then } \texttt{Stmt} \\
 & \mid & \text{if } \texttt{Expr} \text{ then } \texttt{Matched\_stmt} \text{ else } \texttt{Unmatched\_stmt}
\end{array}
$$

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

The statement: if E_1 then if E_2 then S_1 else S_2, has now a unique Parse-Tree.



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).

Inherent Ambiguity

- It would be nice if for every ambiguous grammar, there were some way to "fix" the ambiguity.
- Unfortunately, certain CFLs are inherently ambiguous, meaning that every grammar for the language is ambiguous.

Example: Inherent Ambiguity

- The language $\{0^i 1^j 2^k \mid i = j \text{ or } j = k, i, j, k \geq 1\}$ is inherently ambiguous.
- Intuitively, at least the strings of the form $0^n 1^n 2^n$ can be generated by two different parse trees, one based on checking the 0's and 1's, the other based on checking the 1's and 2's.

One Possible Ambiguous Grammar

$S \rightarrow AB \mid CD$	
$A \rightarrow 0A1 \mid 01$	A generates equal 0's and 1's
$B \rightarrow 2B \mid 2$	B generates any sequence of 2's
$C \rightarrow 0C \mid 0$	C generates any sequence of 0's
$D \rightarrow 1D2 \mid 12$	D generates equal 1's and 2's

There are two derivations of every string with equal numbers of 0's, 1's, and 2's. E.g.:

$$S \Rightarrow AB \Rightarrow 01B \Rightarrow 012$$

$$S \Rightarrow CD \Rightarrow 0D \Rightarrow 012$$

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:

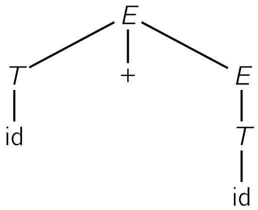
$$\begin{array}{lcl} \mathtt{E} & \rightarrow & \mathtt{T{+}E} \mid \mathtt{T} \\ \mathtt{T} & \rightarrow & \mathsf{id}{*}\mathtt{T} \mid \mathsf{id} \mid (\mathtt{E}) \end{array}$$

  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 $\mathtt{E} \rightarrow \mathtt{T{+}E}$;
3. Use the production $\mathtt{T} \rightarrow \mathsf{id}{*}\mathtt{T}$:
   id matches the first input Token but $*$ does not!
4. **Backtrack:** Use the production $\mathtt{T} \rightarrow \mathsf{id}$:
   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 $\mathtt{E} \rightarrow \mathtt{T+E}$ we fail; then we **Backtrack** and choose $\mathtt{E} \rightarrow \mathtt{T}$.
6. Follow the same step as before (Step 4.) for $\mathtt{T}$ and we succeed with $\mathtt{T} \rightarrow \text{id}$: The last Token matches!!!
7. The successful Parse-Tree is:



Loops in Top-Down Parsing — Left-Recursive Grammars

- 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 Non-Terminal, A, there is a Derivation, $A \Rightarrow^* A\alpha$, for some $\alpha \in \mathbf{V}^*$.
- **Example.** The Grammar with production $E \rightarrow E+T$ is left-recursive.
- **Eliminating Immediate Left Recursion.** If we have a production of the form, $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$, where $\beta_1, \ldots, \beta_m$ do not begin with A, then an equivalent right-recursive Grammar is:

$$\begin{array}{lcl} A & \rightarrow & \beta_1 R \mid \ldots \mid \beta_m R \\ R & \rightarrow & \alpha_1 R \mid \ldots \mid \alpha_n R \mid \epsilon \end{array}$$

- 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:

$$
\begin{array}{lcl}
\texttt{Stmt} & \rightarrow & \textit{if}\ \texttt{Expr}\ \textit{then}\ \texttt{Stmt}\ \textit{else}\ \texttt{Stmt} \\
 & \mid & \textit{while}\ \texttt{Expr}\ \textit{do}\ \texttt{Stmt} \\
 & \mid & \textit{begin}\ \texttt{Stmt\_list}\ \textit{end}
\end{array}
$$

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, first(α) is the set of Tokens that start a string generated by α:

$$\forall a \in \mathbf{V}_T . a \in \text{first}(\alpha) \quad \textit{iff} \quad \alpha \Rightarrow^* a\beta, \text{ with } \alpha, \beta \in \mathbf{V}^*.$$

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

Predictive Parsers (Cont.)

- A Predictive Parser decides between two productions $A \to \alpha$ and $A \to \beta$ by considering the Lookahead symbol;
- If the Lookahead symbol is in $\text{first}(\alpha)$ then $A \to \alpha$ is used.
- **Important!** 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:

$$\begin{array}{lcl} \mathit{Stmt} & \to & \texttt{if}\ \ \mathit{Expr}\ \ \texttt{then}\ \ \mathit{Stmt}\ \ \texttt{else}\ \ \mathit{Stmt} \\ & \mid & \texttt{if}\ \ \mathit{Expr}\ \ \texttt{then}\ \ \mathit{Stmt} \end{array}$$

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

$$\begin{array}{lcl} \mathit{Stmt} & \to & \texttt{if}\ \ \mathit{Expr}\ \ \texttt{then}\ \ \mathit{Stmt}\ \ \mathit{Stmt}' \\ \mathit{Stmt}' & \to & \texttt{else}\ \ \mathit{Stmt} \mid \epsilon \end{array}$$

- **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

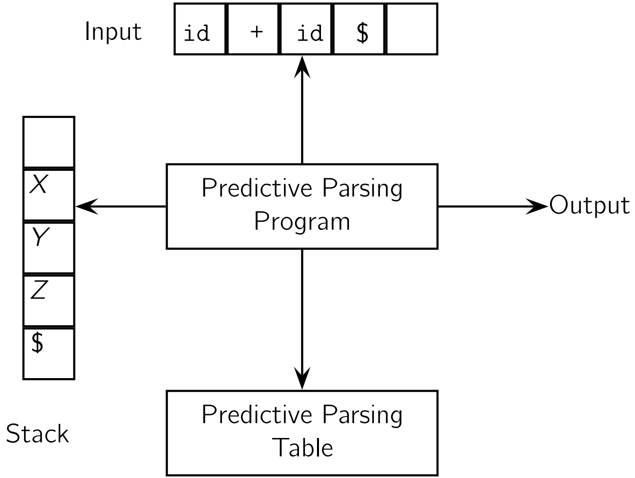
- Given an LL(1) Grammar, then for each non-terminal and Token (Lookahead) there is only one production that could lead to success.
- Predictive parsers built on top of LL(1) Grammars can be specified as a two-dimensional table—called the *Parsing Table*, with:
  - One dimension for current non-terminal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

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, and \$ the input right-end marker.
- Now, let X the symbol on top of the Stack and a the current Token in the input. There are three possibilities:
  1. If $X = \mathrm{a} = \$$, the parser halts successfully;
  2. If $X = \mathrm{a} \neq \$$, 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 $M[X, \mathrm{a}]$:
     - If $M[X, \mathrm{a}] =$ **error**, then an error recovery is done;
     - If $M[X, \mathrm{a}] = \{X \to UVW\}$, 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 non-ambiguos Grammar for arithmetic expressions:

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

Predictive Parser Program: An Example (Cont.)

A predictive table for the Grammar is:

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

Where empty entries indicate an error situation.

Predictive Parser Program: An Example (Cont.)

Predictive Parser Program: An Example (Cont.)

The moves with input "id + id * id" are:

Stack	*Input*	*Output*
$\$E$	$\text{id} + \text{id} * \text{id}\$$	
$\$E'T$	$\text{id} + \text{id} * \text{id}\$$	$E \to TE'$
$\$E'T'F$	$\text{id} + \text{id} * \text{id}\$$	$T \to FT'$
$\$E'T'\text{id}$	$\text{id} + \text{id} * \text{id}\$$	$F \to \text{id}$
$\$E'T'$	$+\text{id} * \text{id}\$$	
$\$E'$	$+\text{id} * \text{id}\$$	$T' \to \epsilon$
$\$E'T+$	$+\text{id} * \text{id}\$$	$E' \to +TE'$
$\$E'T$	$\text{id} * \text{id}\$$	
$\$E'T'F$	$\text{id} * \text{id}\$$	$T \to FT'$

Stack	*Input*	*Output*
$\$E'T'\text{id}$	$\text{id} * \text{id}\$$	$F \to \text{id}$
$\$E'T'$	$*\text{id}\$$	
$\$E'T'F*$	$*\text{id}\$$	$T' \to *FT'$
$\$E'T'F$	$\text{id}\$$	
$\$E'T'\text{id}$	$\text{id}\$$	$F \to \text{id}$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \to \epsilon$
$\$$	$\$$	$E' \to \epsilon$
$\$$	$\$$	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

- 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, first(α) is the set of Tokens that start a string generated by α:
  $$\forall a \in \mathbf{V}_T . a \in \text{first}(\alpha) \quad \textit{iff}\, \alpha \Rightarrow^* a\beta, \text{ with } \alpha, \beta \in \mathbf{V}^*.$$
  Furthermore, if $\alpha = \epsilon$ or $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{first}(\alpha)$.
- Given a non-terminal A, follow(A) is the set of Tokens, a, that can appear immediately to the right of A in some sentential form.
  Definition (follow).
  $$\text{follow}(A) = \{\text{a} \in \mathbf{V}_T \mid S \Rightarrow^* \alpha A \text{a} \beta, \text{with } \alpha, \beta \in \mathbf{V}^*\}.$$
  - **Note.** There may have been symbols between A and a, but they derived ϵ and disappeared.

Computing first

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

1. If $X \in \mathbf{V}_T$, then $\text{first}(X) = \{X\}$.
2. If $X \to \epsilon$ is a production, then add ϵ to $\text{first}(X)$.
3. If $X \to Y_1 Y_2 \dots Y_k$ 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 $X_1 X_2 \ldots X_n \in \mathbf{V}^*$, we compute **first**$(X_1 X_2 \ldots X_n)$:

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

Computing follow

Definition. **follow**$(A) = \{a \in \mathbf{V}_T \mid S \Rightarrow^* \alpha A a \beta, \mathrm{with}\ \alpha, \beta \in \mathbf{V}^*\}$.

follow(A): For each non-terminal, A, apply the following rules until nothing more can be added:

1. Add \$ to follow(S) (if S is the scope, and \$ is the input right endmarker);
2. For all productions $Y \to \alpha A Y_1 \dots Y_n$ (where, $\alpha \in \mathbf{V}^*$ and $Y_i \in \mathbf{V}$):
    1. Add first$(Y_1) - \{\epsilon\}$ to follow(A).
    2. If $\epsilon \in$ first(Y_1), then:
       Add first$(Y_2) - \{\epsilon\}$ to follow(A); otherwise Stop.
    3. ...
    4. If $\epsilon \in$ first(Y_{n-1}), then:
       Add first$(Y_n) - \{\epsilon\}$ to follow(A); otherwise Stop.
    5. If $\epsilon \in$ first(Y_n), then:
       Add follow(Y) to follow(A).

Important: ϵ never belongs to follow!

first and follow: An Example

Consider the Grammar for arithmetic expression:

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

first(E) =first(T) =first(F) $= \{(, \mathrm{id}\}$.
first(E') $= \{+, \epsilon\}$.
first(T') $= \{*, \epsilon\}$.

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

Constructing Predictive Parsing Tables (Cont.)

- The production $A \to \alpha$, with a in first(α), is used if a is the lookahead symbol.
- **Problem.** When $\epsilon \in$ first(α). Then, the production $A \to \alpha$ is used if the lookahead is in follow(A).

Constructing Predictive Parsing Tables (Cont.)

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

1. For each production $A \to \alpha$ in **G** do:
    1. For each terminal a in first(α), add $A \to \alpha$ to $M[A, \mathrm{a}]$.
    2. If $\epsilon \in$ first(α), then for each terminal a in follow(A), add $A \to \alpha$ to $M[A, \mathrm{a}]$.
    3. If $\epsilon \in$ first(α) and \$ is in follow(A), add $A \to \alpha$ 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 predictive parsing table has no multiple entries is said to be LL(1).
- The Grammar for arithmetic expressions (once factored) 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 VI—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