Syntax Analysis: Context-Free Grammars

Lecture 3

Objectives

By the end of this lecture you should be able to:

1. Identify the role of syntax analysis in a compiler.
2. Construct derivations and parse trees of strings from context-free grammars.
3. Design context-free grammars.
4. Prove that simple context-free grammars are correct.
5. Construct unambiguous grammars (sometimes).
6. Eliminate left-recursion from a grammar.
7. Left-factor a grammar.

Outline

Outline

What It Does

Main Function

1. Determine whether the program is grammatical.
2. Generate a parse tree for a stream of tokens.
   - A parse tree may not be explicitly generated.

Auxiliary Function

1. Identify syntax errors.
2. Recover from common errors to continue processing the input.

Side Effects

- Update the symbol table.

What It Does

Main Function

1. Determine whether the program is grammatical.
2. Generate a parse tree for a stream of tokens.
    - A parse tree may not be explicitly generated.

Auxiliary Function

1. Identify syntax errors.
2. Recover from common errors to continue processing the input.

Side Effects

- Update the symbol table.

What It Does

Main Function

1. Determine whether the program is grammatical.
2. Generate a parse tree for a stream of tokens.
    - A parse tree may not be explicitly generated.

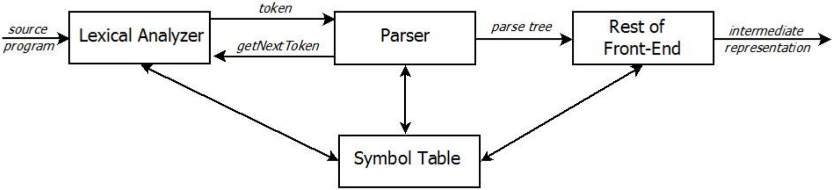
Auxiliary Function

1. Identify syntax errors.
2. Recover from common errors to continue processing the input.

Side Effects

- Update the symbol table.

Connection to the Rest of the System



Associated Concepts: Grammars

Grammars

- A grammar is a formal device used to specify the syntactic structure of a language.
- Regular languages may be specified using so-called regular grammars.
- Most interesting languages are not regular.
- Programming languages are mostly context-free; they are specified using context-free grammars.

Associated Concepts: Grammars

Grammars

- A grammar is a formal device used to specify the syntactic structure of a language.
- Regular languages may be specified using so-called regular grammars.
- Most interesting languages are not regular.
- Programming languages are mostly context-free; they are specified using context-free grammars.

Associated Concepts: Grammars

Grammars

- A grammar is a formal device used to specify the syntactic structure of a language.
- Regular languages may be specified using so-called regular grammars.
- Most interesting languages are not regular.
- Programming languages are mostly context-free; they are specified using context-free grammars.

Associated Concepts: Grammars

Grammars

- A grammar is a formal device used to specify the syntactic structure of a language.
- Regular languages may be specified using so-called regular grammars.
- Most interesting languages are not regular.
- Programming languages are mostly context-free; they are specified using context-free grammars.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Associated Concepts: Parsers

Parsers

- A parser is an algorithm which, given a grammar, produces a parse tree for an input string.
- Context-free parsing may be carried out by the CYK algorithm or the Early algorithm, appropriately modified to produce a parse tree.
- Unfortunately, this is not very efficient.
- Parsers used in compilers are of two main types:
  1. top-down—parse trees are constructed from the root down to the leaves.
  2. bottom-up—parse trees are constructed from the leaves up to the root.
- Some kind of mixed-mode parsing is often adopted though.

Outline

What is a Context-Free Grammar?

Definition

A context-free grammar (CFG) is a 4-tuple (V, Σ, R, S), where

1. V is an alphabet, whose symbols are referred to as variables or non-terminals,
2. Σ, is an alphabet, disjoint from V, whose symbols are called terminals,
3. $R \subseteq V \times (V \cup \Sigma)^*$ is a non-empty finite set of production rules, and
4. $S \in V$ is the start variable.

Displaying CFGs

- Typically only rules are displayed (using an $\longrightarrow$), each on a separate line.
- Non-terminals are the symbols which appear on the left side of at least one rule; terminals are the other symbols.
- The start variable is the variable appearing on the left side of the top rule.
- Multiple rules with the same left side are combined in one line, using $|$ to separate the right sides.

Displaying CFGs

- Typically only rules are displayed (using an $\longrightarrow$), each on a separate line.
- Non-terminals are the symbols which appear on the left side of at least one rule; terminals are the other symbols.
- The start variable is the variable appearing on the left side of the top rule.
- Multiple rules with the same left side are combined in one line, using $|$ to separate the right sides.

Displaying CFGs

- Typically only rules are displayed (using an $\longrightarrow$), each on a separate line.
- Non-terminals are the symbols which appear on the left side of at least one rule; terminals are the other symbols.
- The start variable is the variable appearing on the left side of the top rule.
- Multiple rules with the same left side are combined in one line, using $|$ to separate the right sides.

Displaying CFGs

- Typically only rules are displayed (using an $\longrightarrow$), each on a separate line.
- Non-terminals are the symbols which appear on the left side of at least one rule; terminals are the other symbols.
- The start variable is the variable appearing on the left side of the top rule.
- Multiple rules with the same left side are combined in one line, using $|$ to separate the right sides.

Example G_1

Example (Arithmetic Expressions 1)

$$E \quad \longrightarrow \quad E + E \mid E * E \mid (\, E \,) \mid \mathbf{id} \mid \mathbf{number}$$

Note that terminals are names of lexical categories.

Terminology

- α is a sentential form of $G = (V, \Sigma, R, S)$ if $\alpha \in (V \cup \Sigma)^*$.
- w is a sentence of $G = (V, \Sigma, R, S)$ if $w \in \Sigma^*$.

Derivations

Let $G = (V, \Sigma, R, S)$ be a CFG. Let α, β, and γ be sentential forms of G.

- If $(A \longrightarrow \gamma) \in R$, then $\alpha A \beta$ yields $\alpha\gamma\beta$, written $\alpha A \beta \Rightarrow \alpha\gamma\beta$.
- α derives β, written $\alpha \overset{*}{\Rightarrow} \beta$, if
  1. $\alpha = \beta$, or
  2. there is a sequence $\alpha_1, \alpha_2, \ldots, \alpha_k$ for $k > 0$ such that

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k \Rightarrow \beta$$

- The language of G is the set

$$L(G) = \{w \in \Sigma^* | \, S \overset{*}{\Rightarrow} w\}$$

- Show that $\mathbf{id}+\mathbf{id}*(\mathbf{number}) \in L(G_1)$.

Derivations

Let $G = (V, \Sigma, R, S)$ be a CFG. Let α, β, and γ be sentential forms of G.

- If $(A \longrightarrow \gamma) \in R$, then $\alpha A\beta$ yields $\alpha\gamma\beta$, written $\alpha A\beta \Rightarrow \alpha\gamma\beta$.
- α derives β, written $\alpha \overset{*}{\Rightarrow} \beta$, if
  1. $\alpha = \beta$, or
  2. there is a sequence $\alpha_1, \alpha_2, \ldots, \alpha_k$ for $k > 0$ such that
  $$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k \Rightarrow \beta$$
- The language of G is the set
  $$L(G) = \{w \in \Sigma^* | \, S \overset{*}{\Rightarrow} w\}$$
- Show that $\mathbf{id}+\mathbf{id}*(\mathbf{number}) \in L(G_1)$.

Special Derivations

- A derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$ is a leftmost derivation if, for every $1 \leq i < k$, there is some $(A_i \longrightarrow \gamma_i) \in R$, some sentence w_i, and some sentential form β_i such that $\alpha_i = w_i A_i \beta_i$ and $\alpha_{i+1} = w_i \gamma_i \beta_i$.
- A derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$ is a rightmost derivation if, for every $1 \leq i < k$, there is some $(A_i \longrightarrow \gamma_i) \in R$, some sentence w_i, and some sentential form β_i such that $\alpha_i = \beta_i A_i w_i$ and $\alpha_{i+1} = \beta_i \gamma_i w_i$.
- Give leftmost and rightmost derivations of **id**+**id**$*$(**number**) from E in G_1.

Special Derivations

- A derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$ is a leftmost derivation if, for every $1 \leq i < k$, there is some $(A_i \longrightarrow \gamma_i) \in R$, some sentence w_i, and some sentential form β_i such that $\alpha_i = w_i A_i \beta_i$ and $\alpha_{i+1} = w_i \gamma_i \beta_i$.
- A derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$ is a rightmost derivation if, for every $1 \leq i < k$, there is some $(A_i \longrightarrow \gamma_i) \in R$, some sentence w_i, and some sentential form β_i such that $\alpha_i = \beta_i A_i w_i$ and $\alpha_{i+1} = \beta_i \gamma_i w_i$.
- Give leftmost and rightmost derivations of $\mathbf{id}+\mathbf{id}*(\mathbf{number})$ from E in G_1.

Special Derivations

- A derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$ is a leftmost derivation if, for every $1 \leq i < k$, there is some $(A_i \longrightarrow \gamma_i) \in R$, some sentence w_i, and some sentential form β_i such that $\alpha_i = w_i A_i \beta_i$ and $\alpha_{i+1} = w_i \gamma_i \beta_i$.
- A derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$ is a rightmost derivation if, for every $1 \leq i < k$, there is some $(A_i \longrightarrow \gamma_i) \in R$, some sentence w_i, and some sentential form β_i such that $\alpha_i = \beta_i A_i w_i$ and $\alpha_{i+1} = \beta_i \gamma_i w_i$.
- Give leftmost and rightmost derivations of $\mathbf{id}+\mathbf{id}*(\mathbf{number})$ from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \le i < k$.
  - α_1 is the root of the tree.
  - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \le i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of **id**+**id**$*$(**number**) from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \leq i < k$.
  - α_1 is the root of the tree.
  - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \leq i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of **id**+**id**$*$(**number**) from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \leq i < k$.
    - α_1 is the root of the tree.
    - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \leq i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of $\textbf{id}+\textbf{id}*(\textbf{number})$ from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \leq i < k$.
  - α_1 is the root of the tree.
  - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \leq i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of **id**+**id**$*$(**number**) from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \leq i < k$.
  - α_1 is the root of the tree.
  - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \leq i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of $\mathbf{id}+\mathbf{id}*(\mathbf{number})$ from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \leq i < k$.
  - α_1 is the root of the tree.
  - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \leq i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of **id**+**id**$*$(**number**) from E in G_1.

Parse Trees

- A parse tree is a graphical representation of a derivation starting by a non-terminal symbol.
- Each internal node of the tree is a non-terminal.
- Each leaf is a terminal or a non-terminal.
- Consider a derivation $\alpha_1 (\in V) \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k$. The following is true for every $1 \leq i < k$.
  - α_1 is the root of the tree.
  - if $\alpha_i = \beta_i A_i \delta_i$ and $\alpha_{i+1} = \beta_i \gamma_i \delta_i$ and $(A_i \longrightarrow \gamma_i) \in R$, then nodes for symbols of the occurrence of γ_i following the prefix β_i occur at depth $j + 1 \leq i$ and are children of the corresponding A_i node at depth j.
- Give a parse tree for a derivation of **id**+**id**$*$(**number**) from E in G_1.

Observations

- For every derivation, there is a unique corresponding parse tree.
- For every parse tree, there are possibly many corresponding derivations.
- For every parse tree, there is exactly one corresponding leftmost (rightmost) derivation.

Observations

- For every derivation, there is a unique corresponding parse tree.
- For every parse tree, there are possibly many corresponding derivations.
- For every parse tree, there is exactly one corresponding leftmost (rightmost) derivation.

Observations

- For every derivation, there is a unique corresponding parse tree.
- For every parse tree, there are possibly many corresponding derivations.
- For every parse tree, there is exactly one corresponding leftmost (rightmost) derivation.

Example G_2

Example (Arithmetic Expressions 2)

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

Example G_3

Example (Arithmetic Expressions 3)

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

Example G_4

Example (If Statement 1)

$$
\begin{array}{lcl}
\mathit{stmnt} & \longrightarrow & \textbf{if } \mathit{bexpr} \textbf{ then } \mathit{stmnt} \\
\mathit{stmnt} & \longrightarrow & \textbf{if } \mathit{bexpr} \textbf{ then } \mathit{stmnt} \textbf{ else } \mathit{stmnt} \\
\mathit{stmnt} & \longrightarrow & \ldots \\
\vdots & & \vdots \\
\mathit{bexpr} & \longrightarrow & \ldots \\
\vdots & & \vdots
\end{array}
$$

Example G_5

Example (If Statement 2)

$$
\begin{array}{lcl}
\mathit{stmnt} & \longrightarrow & \mathit{open} \mid \mathit{matched} \\
\mathit{open} & \longrightarrow & \textbf{if}\ \mathit{bexpr}\ \textbf{then}\ \mathit{stmnt} \\
\mathit{open} & \longrightarrow & \textbf{if}\ \mathit{bexpr}\ \textbf{then}\ \mathit{matched}\ \textbf{else}\ \mathit{open} \\
\mathit{matched} & \longrightarrow & \textbf{if}\ \mathit{bexpr}\ \textbf{then}\ \mathit{matched}\ \textbf{else}\ \mathit{matched} \\
\mathit{matched} & \longrightarrow & \ldots \\
\vdots & & \vdots \\
\mathit{bexpr} & \longrightarrow & \ldots \\
\vdots & & \vdots
\end{array}
$$

Outline

Correctness of a Grammar

- A grammar G is correct simply if $L(G)$ is what it is supposed to be.
- More precisely, given a CFG G and a language L (which may be inductively defined), we need to prove that $L(G) = L$.
- This is done by proving two claims:

  Claim 1 $L(G) \subseteq L$ (Soundness).
  - Typically proved by strong induction on the length of derivations of sentences in G.

  Claim 2 $L \subseteq L(G)$ (Completeness).
  - Typically proved by strong induction on the length (or structure) of strings in L.

Example (I)

Example (Grammar with a single variable)

Consider the following CFG G.

$$S \quad \longrightarrow \quad \mathtt{a}\, S\, \mathtt{b} \mid \varepsilon$$

Prove that $L(G) = \{\mathtt{a}^m \mathtt{b}^m | m \in \mathbb{N}\}$.

Example (II)

Example (Claim 1)

We use induction on the length n of derivations.

Basis ($n = 1$). Only such derivation is $S \Rightarrow \varepsilon$, and $\varepsilon \in L$.

Hypothesis. For every $k < n + 1$, if $S \stackrel{k}{\Rightarrow} w$ ($w \in \Sigma^*$), then $w \in L$.

Step. Let $w \in L(G)$ where $S \xrightarrow{n+1} w$. Thus, $S \Rightarrow \mathtt{a}S\mathtt{b} \stackrel{n}{\Rightarrow} \mathtt{a}u\mathtt{b} = w$. Hence, $S \stackrel{n}{\Rightarrow} u$. By the induction hypothesis, $u = \mathtt{a}^l\mathtt{b}^l \in L$, for some $l \in \mathbb{N}$. Thus, $w = \mathtt{a}^{l+1}\mathtt{b}^{l+1} \in L$.

Example (III)

Example (Claim 2)

We use induction on the length n of $w \in L$.

Basis ($n = 0$). $w = \varepsilon$ and $\varepsilon \in L(G)$ (given $S \longrightarrow \varepsilon$).

Hypothesis. For every $k < n + 1$, if $|w| = k$ and $w \in L$, then $w \in L(G)$.

Step. Let $|w| = n + 1$ and $w \in L$. Thus, there is some $m \in \mathbb{N}^+$ with $w = \mathtt{a}^m\mathtt{b}^m = \mathtt{a}\mathtt{a}^{m-1}\mathtt{b}^{m-1}\mathtt{b}$. Now, since $|\mathtt{a}^{m-1}\mathtt{b}^{m-1}| < n + 1$, it follows by the induction hypothesis that $S \stackrel{*}{\Rightarrow} \mathtt{a}^{m-1}\mathtt{b}^{m-1}$. Hence, $S \Rightarrow \mathtt{a}S\mathtt{b} \stackrel{*}{\Rightarrow} \mathtt{a}\mathtt{a}^{m-1}\mathtt{b}^{m-1}\mathtt{b} = \mathtt{a}^m\mathtt{b}^m$.

Outline

Useful Grammar Transformations

When writing a grammar, it might be useful to carry out the following transformations.

- $G \longrightarrow$ unambiguous G'.
- $G \longrightarrow$ non-left-recursive G'.
- $G \longrightarrow$ left-factored G'.

Ambiguity

Definition

A CFG G is ambiguous if there is some $w \in L(G)$ with more than one parse tree (leftmost (rightmost) derivation).

- Unfortunately, there is no algorithm for transforming any ambiguous grammar to an equivalent unambiguous grammar.
- There are some common cases, though, in programming languages.

Ambiguity

Definition

A CFG G is ambiguous if there is some $w \in L(G)$ with more than one parse tree (leftmost (rightmost) derivation).

- Unfortunately, there is no algorithm for transforming any ambiguous grammar to an equivalent unambiguous grammar.
- There are some common cases, though, in programming languages.

Associativity and Precedence of Operators

- Operators with higher precedence occur in strings which are derivable from variables that appear *deeper* in the grammar (and, hence, in a parse tree).
- Left associative operators occur on the right-side of left-recursive rules.
- Right associative operators occur on the right-side of right-recursive rules.
- Compare G_1 and G_2.

Associativity and Precedence of Operators

- Operators with higher precedence occur in strings which are derivable from variables that appear *deeper* in the grammar (and, hence, in a parse tree).
- Left associative operators occur on the right-side of left-recursive rules.
- Right associative operators occur on the right-side of right-recursive rules.
- Compare G_1 and G_2.

Associativity and Precedence of Operators

- Operators with higher precedence occur in strings which are derivable from variables that appear *deeper* in the grammar (and, hence, in a parse tree).
- Left associative operators occur on the right-side of left-recursive rules.
- Right associative operators occur on the right-side of right-recursive rules.
- Compare G_1 and G_2.

Associativity and Precedence of Operators

- Operators with higher precedence occur in strings which are derivable from variables that appear *deeper* in the grammar (and, hence, in a parse tree).
- Left associative operators occur on the right-side of left-recursive rules.
- Right associative operators occur on the right-side of right-recursive rules.
- Compare G_1 and G_2.

The Dangling *else*

- The form

  $$\texttt{if}\ e1\ \texttt{then}\ \texttt{if}\ e2\ \texttt{then}\ S2\ \texttt{else}\ S3$$

  is ambiguous.
- Common policy: An occurrence of `else` is associated with the lexically closest `if`.
- Compare G_4 and G_5.

The Dangling *else*

- The form

$$\texttt{if}\ e1\ \texttt{then}\ \texttt{if}\ e2\ \texttt{then}\ S2\ \texttt{else}\ S3$$

  is ambiguous.
- Common policy: An occurrence of `else` is associated with the lexically closest `if`.
- Compare G_4 and G_5.

The Dangling *else*

- The form

$$\texttt{if}\ e1\ \texttt{then}\ \texttt{if}\ e2\ \texttt{then}\ S2\ \texttt{else}\ S3$$

  is ambiguous.
- Common policy: An occurrence of `else` is associated with the lexically closest `if`.
- Compare G_4 and G_5.

Left-Recursion

Definition

A CFG $G = (V, \Sigma, R, S)$ is left-recursive if, for some $A \in V$, $A \stackrel{+}{\Rightarrow} A\alpha$, for some sentential form α.

Definition

A CFG $G = (V, \Sigma, R, S)$ has immediate left-recursion if, $(A \longrightarrow A\alpha) \in R$, for some $A \in V$ and sentential form α.

Left-recursion may cause some parsers to get into infinite loops.

Left-Recursion

Definition

A CFG $G = (V, \Sigma, R, S)$ is left-recursive if, for some $A \in V$, $A \stackrel{+}{\Rightarrow} A\alpha$, for some sentential form α.

Definition

A CFG $G = (V, \Sigma, R, S)$ has immediate left-recursion if, $(A \longrightarrow A\alpha) \in R$, for some $A \in V$ and sentential form α.

Left-recursion may cause some parsers to get into infinite loops.

Left-Recursion

Definition

A CFG $G = (V, \Sigma, R, S)$ is left-recursive if, for some $A \in V$, $A \stackrel{+}{\Rightarrow} A\alpha$, for some sentential form α.

Definition

A CFG $G = (V, \Sigma, R, S)$ has immediate left-recursion if, $(A \longrightarrow A\alpha) \in R$, for some $A \in V$ and sentential form α.

Left-recursion may cause some parsers to get into infinite loops.

Eliminating Immediate Left-Recursion

From

$$A \longrightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \cdots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where

1. β_i does not start with A.
2. $\alpha_i \neq \varepsilon$

To

$$\begin{aligned} A &\longrightarrow \beta_1\,A' \mid \beta_2\,A' \mid \cdots \mid \beta_n\,A' \\ A' &\longrightarrow \alpha_1\,A' \mid \alpha_2\,A' \mid \cdots \mid \alpha_m\,A' \mid \varepsilon \end{aligned}$$

Compare $G2$ and G_3.

Eliminating Immediate Left-Recursion

From

$$A \longrightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \cdots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where

1. β_i does not start with A.
2. $\alpha_i \neq \varepsilon$

To

$$\begin{array}{lcl} A & \longrightarrow & \beta_1\,A' \mid \beta_2\,A' \mid \cdots \mid \beta_n\,A' \\ A' & \longrightarrow & \alpha_1\,A' \mid \alpha_2\,A' \mid \cdots \mid \alpha_m\,A' \mid \varepsilon \end{array}$$

Compare $G2$ and G_3.

Eliminating Immediate Left-Recursion

From

$$A \longrightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \cdots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where

1. β_i does not start with A.
2. $\alpha_i \neq \varepsilon$

To

$$\begin{aligned} A &\longrightarrow \beta_1\,A' \mid \beta_2\,A' \mid \cdots \mid \beta_n\,A' \\ A' &\longrightarrow \alpha_1\,A' \mid \alpha_2\,A' \mid \cdots \mid \alpha_m\,A' \mid \varepsilon \end{aligned}$$

Compare $G2$ and G_3.

Eliminating Left-Recursion

Algorithm 4.19: Eliminating left recursion.

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

OUTPUT: An equivalent grammar with no left recursion.

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

Example (I)

Example (Problem)

Eliminate left-recursion from the grammar

$$\begin{array}{rcl} S & \longrightarrow & A\,\mathtt{a} \mid \mathtt{b} \\ A & \longrightarrow & A\,\mathtt{c} \mid S\,\mathtt{d} \mid \varepsilon \end{array}$$

Example (II)

Example (Eliminating ε rule)

$$
\begin{array}{lcl}
S & \longrightarrow & A\,\mathtt{a} \mid \mathtt{b} \mid \mathtt{a} \\
A & \longrightarrow & A\,\mathtt{c} \mid S\,\mathtt{d} \mid \mathtt{c}
\end{array}
$$

Example (III)

Example (After iteration 1)

$$
\begin{aligned}
S &\longrightarrow A\,\mathtt{a} \mid \mathtt{b} \mid \mathtt{a} \\
A &\longrightarrow A\,\mathtt{c} \mid S\,\mathtt{d} \mid \mathtt{c}
\end{aligned}
$$

Example (IV)

Example (After inner loop in iteration 2)

$$
\begin{array}{lcl}
S & \longrightarrow & A\,\mathtt{a} \mid \mathtt{b} \mid \mathtt{a} \\
A & \longrightarrow & A\,\mathtt{c} \mid A\,\mathtt{a}\,\mathtt{d} \mid \mathtt{b}\,\mathtt{d} \mid \mathtt{a}\,\mathtt{d} \mid \mathtt{c}
\end{array}
$$

Example (V)

Example (Finally)

$$
\begin{array}{lcl}
S & \longrightarrow & A\,\mathtt{a} \mid \mathtt{b} \\
A & \longrightarrow & \mathtt{b}\,\mathtt{d}\,A' \mid \mathtt{a}\,\mathtt{d}\,A' \mid \mathtt{c}\,A' \\
A' & \longrightarrow & \mathtt{c}\,A' \mid \mathtt{a}\,\mathtt{d}\,A' \mid \varepsilon
\end{array}
$$

Left Factoring

Definition

A CFG $G = (V, \Sigma, R, S)$ is left-factored if, for every $\{A \longrightarrow \alpha, A \longrightarrow \beta\} \subseteq R$, $(\alpha \neq \beta)$, the longest common prefix of α and β is ε

Left-factored grammars allow us to construct more efficient parsers.

Left Factoring

Definition

A CFG $G = (V, \Sigma, R, S)$ is left-factored if, for every $\{A \longrightarrow \alpha, A \longrightarrow \beta\} \subseteq R$, $(\alpha \neq \beta)$, the longest common prefix of α and β is ε

Left-factored grammars allow us to construct more efficient parsers.

Left-Factoring a Grammar

From

$$A \longrightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \cdots \mid \alpha\, \beta_m \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$$

where

1. γ_i does not start with α.
2. $\alpha \neq \varepsilon$

To

$$\begin{aligned} A &\longrightarrow \alpha\, A' \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n \\ A' &\longrightarrow \beta_1 \mid \beta_2 \mid \cdots\ \beta_m \end{aligned}$$

Left-Factoring a Grammar

From

$$A \longrightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \cdots \mid \alpha\,\beta_m \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$$

where

1. γ_i does not start with α.
2. $\alpha \neq \varepsilon$

To

$$\begin{aligned} A &\longrightarrow \alpha\,A' \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n \\ A' &\longrightarrow \beta_1 \mid \beta_2 \mid \cdots\ \beta_m \end{aligned}$$

Example (I)

Example (G_1)

Left-factor the following grammar.

$$E \quad \longrightarrow \quad E + E \mid E * E \mid (\ E\ ) \mid \textbf{id} \mid \textbf{number}$$

Example (II)

Example (Result)

$$
\begin{array}{lcl}
E & \longrightarrow & E\,E' \mid (\,E\,) \mid \mathbf{id} \mid \mathbf{number} \\
E' & \longrightarrow & +\,E \mid *\,E
\end{array}
$$