

G54FOP: Lecture 2

Context-free Grammars, Derivation Trees, and Ambiguity

Henrik Nilsson

University of Nottingham, UK

G54FOP: Lecture 2 – p.1/34

Derivations and Derivation Trees

Given a derivation tree for a grammar G :

- The string of leaf labels read from left to right is the **yield** of the tree.
- The yield is a sentential form of G .

The derives relation and derivation trees are related as follows:

A string α is the yield of some derivation tree for a grammar G iff $S \xrightarrow{*}_G \alpha$.

G54FOP: Lecture 2 – p.4/34

Leftmost and Rightmost Derivations

- A derivation is **leftmost** if productions are always applied to the leftmost nonterminal at each step in a derivation.
- A derivation is **rightmost** if productions are always applied to the rightmost nonterminal at each step in a derivation.

Leftmost derivation:

$$G: \begin{array}{l} S \rightarrow AB \mid BA \\ A \rightarrow a \\ B \rightarrow Ab \end{array} \quad \begin{array}{l} S \xRightarrow{lm} BA \xRightarrow{lm} AbA \\ \xRightarrow{lm} abA \xRightarrow{lm} aba \end{array}$$

G54FOP: Lecture 2 – p.7/34

Derivation Tree

A tree is a **derivation** or **parse tree** for CFG $G = (N, T, P, S)$ if:

- every vertex has a **label** from $N \cup T \cup \{\epsilon\}$
- the label of the root is S
- labels of interior vertices belong to N
- if vertex n has label A and vertices n_1, n_2, \dots, n_k are the children of n , from left to right, with labels X_1, X_2, \dots, X_k , then $A \rightarrow X_1X_2 \dots X_k$ is a production in P
- if a vertex n has label ϵ , then n is a leaf and the only child of its parent.

G54FOP: Lecture 2 – p.2/34

Exercise

Grammar for Simple Arithmetic Expressions:

$$G = (\{E, E_p, V, I, D\}, \{+, -, *, /, (,), x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, E)$$

where P consists of the productions

$$\begin{array}{l} E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E_p \\ E_p \rightarrow V \mid I \mid (E) \\ V \rightarrow x \mid y \mid z \\ I \rightarrow DI \mid D \\ D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

G54FOP: Lecture 2 – p.5/34

Ambiguous Grammars (1)

A CFG G is **ambiguous** if some word in $L(G)$ has **more than one derivation tree**.

A derivation tree determines a unique leftmost and a unique rightmost derivation.

Thus, equivalently: A CFG G is **ambiguous** if some word in $L(G)$ has

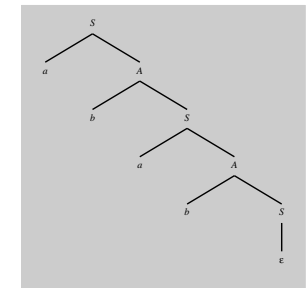
- more than one leftmost derivation**, or
- more than one rightmost derivation**.

G54FOP: Lecture 2 – p.8/34

Derivation Tree: Example

Derivation tree for the string $abab \in L(G)$:

$$G: \begin{array}{l} S \rightarrow \epsilon \mid aA \\ A \rightarrow bS \end{array}$$



G54FOP: Lecture 2 – p.3/34

Exercise

Draw the derivation trees for the following arithmetic expressions:

- $1 + x$
- $42 * x - z$
- $(13 + 7 * x * (y - 123))$
- $(1$

G54FOP: Lecture 2 – p.6/34

Ambiguous Grammars (2)

- A CFL for which every CFG is ambiguous is **inherently ambiguous**.

- The following language L is inherently ambiguous:

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

- Reason: All but a finite number of strings of the form $a^n b^n c^n d^n$ must be generated in two different ways. (The proof is not easy!)

G54FOP: Lecture 2 – p.9/34

Ambiguous Grammars (3)

- Most CFLs are not inherently ambiguous; i.e., an ambiguous CFG G for a language L can often be **transformed** into an **equivalent** but unambiguous grammar G' .
- The ambiguity of a CFG is in general **undecidable**.

Eliminating Ambiguity: Dangling-Else

Note that the distinction is important, as the two trees suggest **different semantics**.

For example, suppose $expr_1$ evaluates to true, and $expr_2$ evaluates to false. Which, if any, of $stmt_1$ and $stmt_2$ gets executed?

Exercise

Is the grammar for Simple Arithmetic Expressions ambiguous or not?

- If it is, demonstrate this.
- If not, explain why.

Eliminating Ambiguity: Dangling-Else

Consider the following “dangling-else” grammar:

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

and the following program fragment:

$$\text{if } expr_1 \text{ then if } expr_2 \text{ then } stmt_1 \text{ else } stmt_2$$

Two possible parse trees!
Hence the grammar is ambiguous!

Eliminating Ambiguity: Dangling-Else

Preferred interpretation:

“Match each `else` with the closest previous unmatched `then`”

That is, **Tree 1** is preferred.

Q: How can that be achieved?

A: Transform the grammar into an **equivalent** but **unambiguous** grammar.

(Exercise: convince yourselves that the following grammar indeed is equivalent!)

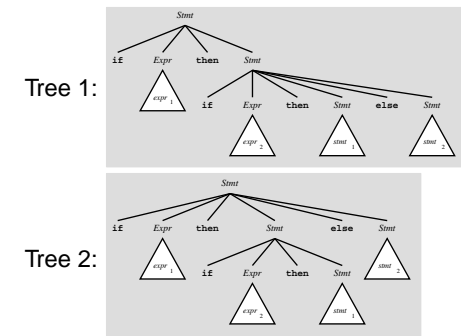
Concrete vs. Abstract Syntax (1)

Arithmetic expressions are usually disambiguated by adopting conventions for operator precedence and associativity. The same ideas can be used to disambiguate grammars.

However, what we have seen so far are examples of **Concrete Syntax**: the fundamental objects are words, unstructured strings of symbols.

A derivation tree can imply a structure for such a word, but if there is more than one possibility, i.e., the grammar is ambiguous, we have a problem.

Eliminating Ambiguity: Dangling-Else



Eliminating Ambiguity: Dangling-Else

Idea: a statement appearing between a `then` and an `else` must be a “matched” statement.

$$\begin{aligned}
 Stmt &\rightarrow MatchedStmt \\
 &| UnmatchedStmt \\
 MatchedStmt &\rightarrow \text{if } Expr \text{ then } MatchedStmt \\
 &\quad \text{else } MatchedStmt \\
 &| \text{other} \\
 UnmatchedStmt &\rightarrow \text{if } Expr \text{ then } Stmt \\
 &| \text{if } Expr \text{ then } MatchedStmt \\
 &\quad \text{else } UnmatchedStmt
 \end{aligned}$$

Concrete vs. Abstract Syntax (2)

For this reason, and because derivation trees are inconveniently detailed, we will mainly be concerned with **abstract syntax** in the following: here the objects already are **trees**. Ambiguity is thus a non-issue, and the grammars can be as simple as possible.

Note on the following slides

The following slides are mainly included for self study.

054FOP: Lecture 2 – p.19/34

More on CFG Notation (3)

For example:

$AssignStmt \rightarrow \underline{Identifier} := Expr$

Here,

- $AssignStmt$ and $Expr$ are nonterminals
- $:=$ is a terminal
- $\underline{Identifier}$ is also a terminal, but its possible spellings are defined elsewhere (usually by a **lexical grammar**).

054FOP: Lecture 2 – p.22/34

More on CFG Notation (6)

For example, a statement like

- “a production $A \rightarrow \beta$ in grammar G ” matches any production in G
- “a production $A \rightarrow b$ in grammar G ” matches any production in G whose right-hand side is a single terminal
- “a production $A \rightarrow X$ in grammar G ” matches any production in G whose right-hand side is a single terminal or non-terminal.

054FOP: Lecture 2 – p.25/34

More on CFG Notation (1)

We will often give CFGs by just stating the productions in one of two styles:

- **Mathematical style:**
 - Mainly used for small, abstract examples; to illustrate general principles; etc.
 - Simple naming conventions used to distinguish terminals and non-terminals:
 - nonterminals: uppercase letters, like A, B, S
 - terminals: lowercase letters or digits, like $a, b, 3$
 - Start symbol usually called S .

054FOP: Lecture 2 – p.20/34

More on CFG Notation (4)

The mathematical style will also be used at the **meta level**, i.e. when talking **about** grammars. The following conventions will then be used:

- uppercase letters near the beginning of the alphabet range over the nonterminals of the grammar; e.g. A, B, C
- S stands for the start symbol
- lowercase letters near the beginning of the alphabet range over the terminals of the grammar; e.g. a, b, c

054FOP: Lecture 2 – p.23/34

Backus-Naur Form (BNF)

- The CFGs we have seen so far have (essentially) been expressed in **Backus-Naur Form** (BNF):
 - productions are of the form $A \rightarrow \alpha$
 - grouping of productions for the same non-terminal using $|$; e.g., $A \rightarrow \alpha|\beta$.
- Introduced by John Backus in UNESCO report on Algol 58, slightly modified by Peter Naur for Algol 60.
- BNF proper uses $::=$ instead of \rightarrow and nonterminals are enclosed in angle brackets, $<$ and $>$.

054FOP: Lecture 2 – p.26/34

More on CFG Notation (2)

- **Programming Language Specification style:**
 - Used for larger, more realistic examples.
 - Typographical conventions used to distinguish terminals and non-terminals:
 - nonterminals are written like *this*
 - terminals are written like **this**
 - terminals with **variable spelling** and special symbols are written like *this*
 - The start symbol is often implied by the context.

054FOP: Lecture 2 – p.21/34

More on CFG Notation (4)

- X, Y, Z , range over terminals and nonterminals
- α, β, γ range over **strings** of terminals and non-terminals.

054FOP: Lecture 2 – p.24/34

Extended BNF

Extended BNF (EBNF) is a more convenient way of describing CFGs than is BNF.

- Additional EBNF constructs:
 - parentheses for grouping
 - $|$ for alternatives **within** parentheses
 - $*$ for iteration (W&B's notation).
- EBNF is **no more powerful** than BNF: the languages described are still the CFLs, and any EBNF grammar can be transformed into BNF.

054FOP: Lecture 2 – p.27/34

EBNF: Grouping

Grouping can be eliminated by introducing a new nonterminal for each group:

$$A \rightarrow \dots (\alpha_1) \dots (\alpha_k) \dots$$

is equivalent to

$$\begin{aligned} A &\rightarrow \dots A_1 \dots A_k \dots \\ A_1 &\rightarrow \alpha_1 \\ &\vdots \\ A_k &\rightarrow \alpha_k \end{aligned}$$

054FOP: Lecture 2 – p.28/34

EBNF: Iteration (2)

The grammar G with the single production

$$S \rightarrow a(bb)^*c$$

generates the language

$$\begin{aligned} L(G) &= \{a(bb)^i c \mid i \geq 0\} \\ &= \{ac, abbc, abbbbc, abbbbbbc, \dots\} \end{aligned}$$

An equivalent left-recursive grammar is

$$\begin{aligned} S &\rightarrow aAc \\ A &\rightarrow \epsilon \mid Abb \end{aligned}$$

054FOP: Lecture 2 – p.31/34

EBNF: ISO Notation (2)

ISO	W&B
{ A }	A*
[A]	(A ε)

Example:

$$\begin{aligned} \text{ISO: } ParamList &\rightarrow ([Param \{ , Param \}]) \\ \text{W\&B: } ParamList &\rightarrow (Param (, Param)^* | \epsilon) \end{aligned}$$

This production describes parameter lists like:

$$\begin{aligned} () & \\ (x) & \quad \text{(assuming } x, y, \text{ and } z \\ (x, y) & \quad \text{are valid parameters)} \\ (x, y, z) & \end{aligned}$$

054FOP: Lecture 2 – p.34/34

EBNF: Grouping of Alternatives

Alternatives in a group does not add anything new:

$$A \rightarrow B (C \mid D \mid E) F$$

is by elimination of grouping equivalent to

$$\begin{aligned} A &\rightarrow BA_1F \\ A_1 &\rightarrow C \mid D \mid E \end{aligned}$$

which in turn is just a shorter way of writing

$$\begin{aligned} A &\rightarrow BA_1F & A_1 &\rightarrow D \\ A_1 &\rightarrow C & A_1 &\rightarrow E \end{aligned}$$

054FOP: Lecture 2 – p.29/34

EBNF: Example

The following EBNF grammar

$$Block \rightarrow \text{begin } (Decl \mid Stmt)^* \text{end}$$

(where $Decl$ and $Stmt$ are defined elsewhere) is equivalent to the following BNF grammar:

$$\begin{aligned} Block &\rightarrow \text{begin } BlockRec \text{end} \\ BlockRec &\rightarrow \epsilon \mid BlockRec BlockAlts \\ BlockAlts &\rightarrow Decl \mid Stmt \end{aligned}$$

Thus we see that EBNF can be quite a bit more concise and readable than plain BNF.

054FOP: Lecture 2 – p.32/34

EBNF: Iteration (1)

The iterative construct can be replaced by explicit recursion:

$$A \rightarrow \dots B^* \dots$$

is equivalent to (left recursion)

$$\begin{aligned} A &\rightarrow \dots A_1 \dots \\ A_1 &\rightarrow \epsilon \mid A_1 B \end{aligned}$$

or (right recursion)

$$\begin{aligned} A &\rightarrow \dots A_1 \dots \\ A_1 &\rightarrow \epsilon \mid BA_1 \end{aligned}$$

054FOP: Lecture 2 – p.30/34

EBNF: ISO Notation (1)

Watt & Brown use their own EBNF variant.

The more common variant is the ISO (International Organization for Standardization) version (ISO/IEC 14977:1996):

- curly braces (“{” and “}”) used to denote iteration (zero, one or more)
- square brackets (“[” and “]”) used to denote options (zero or one)

054FOP: Lecture 2 – p.33/34