

CS421 Lecture 10: Introduction to Grammars¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

June 19, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Objectives and Review

Context-Free Grammars

Properties of Grammars

Objectives

Your goal for this lecture is to learn how to do the following things:

- ▶ Identify and explain the parts of a grammar.
- ▶ Define *terminal*, *nonterminal*, *production*, *sentence*, *derivation*, *parse tree*, *left-recursive*, *ambiguous*.
- ▶ Use a grammar to draw the parse tree of a sentence.
- ▶ Identify a grammar that is *left-recursive*.
- ▶ Know about *ambiguous grammars*:
 - ▶ Be able to identify one when you see it.
 - ▶ Be able to show ambiguity using derivations or parse trees.
 - ▶ Know two strategies that could eliminate ambiguity.

Resources

- ▶ Compilers: Principles, Techniques, and Tools, by Aho, Sethi, and Ullman (aka The Dragon Book)
- ▶ Engineering a Compiler: Cooper and Torczon
- ▶ Modern Compiler Implementation in Java/ML/C: Appel
- ▶ Most material also in Sebesta and Scott books
- ▶ Theoretical material also in Introduction to the Theory of Computation by Sipser, and Introduction to Automata Theory, Languages and Computation by Hopcroft, Motwani, and Ullman

Reminder: The Problem

- ▶ Computer programs are entered as a stream of ASCII characters.

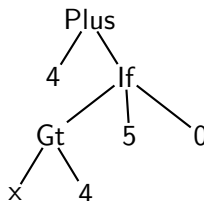
4 + if x > 4 then 5 else 0

- ▶ We want to convert them into an *Abstract Syntax Tree*

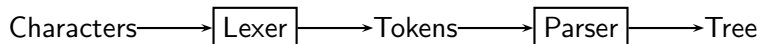
ML code

```

1 PlusExp(
2   IntExp 4,
3   IfExp(
4     GtExp(VarExp "x",
5           IntExp 4),
6     IntExp 5,
7     IntExp 0))
  
```



Reminder: The Solution



The conversion from strings to trees is accomplished in two steps.

- ▶ First, convert the stream of characters into a stream of *tokens*.
 - ▶ This is called *lexing* or *scanning*.
 - ▶ Turns characters into words and categorizes them.
 - ▶ We did this in lectures 6 – 8!
- ▶ Second, convert the stream of tokens into an abstract syntax tree.
 - ▶ This is called *parsing*.
 - ▶ Turns words into *sentences*.

Context-free Grammars

Def: A **Context-free Grammar** (CFG) is a 4-tuple:

$$G = (N, \Sigma, P, S)$$

where:

1. N is a finite, nonempty set of symbols (non-terminals)
2. Σ is a finite set of symbols (terminals)
 $N \cap \Sigma = \emptyset$
 $V \equiv N \cup \Sigma$ (vocabulary)
3. P is a finite subset of $N \times V^*$ (production rules)
4. $S \in N$ (Goal symbol or start symbol)

Sometimes written as $G = (V, \Sigma, P, S)$, $N = V - \Sigma$

Example Grammar: Arithmetic Expressions

$G = (N, \Sigma, P, S)$ where:

$$N = \{E, T, F\}$$

$$\Sigma = \{(\,), +, *, \underline{\text{id}}\}$$

$$P = \{E \rightarrow T$$

$$E \rightarrow E + T$$

$$T \rightarrow F$$

$$T \rightarrow T * F$$

$$F \rightarrow \underline{\text{id}}$$

$$F \rightarrow (E)\}$$

$$S = E$$

Note: $P \subseteq N \times V^*$, where

$$V = N \cup \Sigma = \{E, T, F, (\,), +, *, \underline{\text{id}}\}$$

Note: $(A, \alpha) \in P$ is usually written:

$$A \rightarrow \alpha$$

or $A ::= \alpha$

or $A : \alpha$

Notational Conventions

- ▶ Lower-case letters early in the alphabet (a,b,c), operator symbols, punctuation symbols, digits, bold names like **id** are **terminals**
- ▶ Upper-case letters early in the alphabet (A,B,C), S, and lower-case italic names like *expr* or *stmt* are **non-terminals**
- ▶ Upper-case letters later in the alphabet (X,Y,Z) are **grammar symbols**, either terminals or non-terminals
- ▶ Lower-case letters later in the alphabet (u,v,w,x,y,z) represent **strings of terminals**
- ▶ Lower-case Greek letters (α, β, γ) represent **strings of grammar symbols**

These are from The Dragon Book, pp. 166-167.

Derivations of a Grammar

Directly Derives or \Longrightarrow :

If α and β are strings in V^* (vocabulary), then α directly derives β (written $\alpha \Longrightarrow \beta$) *iff* there is a production $A \rightarrow \delta$ s.t.

- A is a symbol in α
- Substituting string δ for A in α produces the string β .

Canonical Derivation Step :

The above derivation step is called rightmost if A is the rightmost non-terminal in α . (Similarly, leftmost.)

A rightmost derivation step is also called canonical.

Derivations and Sentential Forms

Derivation :

A sequence of steps $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n$, where $\alpha_0 = S$ is called a derivation. It is written $S \Rightarrow^* \alpha_n$.

If every derivation step is rightmost, then this is a canonical derivation.

Sentential Form :

Each α_j in a derivation is called a sentential form of G. Sentential forms derived by rightmost (leftmost) derivations are called right(left)-sentential forms.

Sentences and Languages

Sentences and the Language $L(G)$:

A sentential form α_i consisting only of tokens (i.e., terminals) is called a sentence of G .

The language generated by G is the set of all sentences of G . It is denoted $L(G)$.

Parse Trees of a Grammar

A **Parse Tree** for a grammar G is any tree in which:

- ▶ The root is labeled with S .
- ▶ Each leaf is labeled with a token a ($a \in \Sigma$) or ϵ (the empty string)
- ▶ Each interior node is labeled by a non-terminal.
- ▶ If an interior node is labeled A and has children labeled X_1, \dots, X_n , then $A \rightarrow X_1 \dots X_n$ is a production of G .
- ▶ If $A \rightarrow \epsilon$ is a production in G , then a node labeled A may have a single child labeled ϵ

The string formed by the leaf labels (left to right) is the **yield** of the parse tree.

Parse Trees (continued)

- ▶ An **intermediate parse tree** is the same as a parse tree except the leaves can be non-terminals.

Notes:

- ▶ Every $\alpha \in L(G)$ is the yield of some parse tree.
- ▶ Consider a derivation, $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n$, where $\alpha_n \in L(G)$.

For each α_i , we can construct an intermediate parse tree.

The last one will be the parse tree for the sentence α_n .

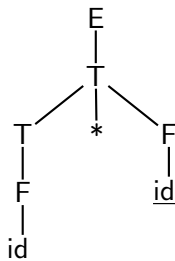
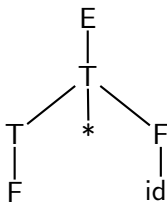
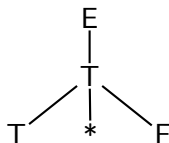
- ▶ A parse tree ignores the order in which symbols are replaced to derive a string.

Derivations and Parse Trees

Example: The rightmost derivation and the parse tree for:

id * id

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \underline{id} \Rightarrow F * \underline{id} \Rightarrow \underline{id} * \underline{id}$



Uniqueness of Derivations

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: **Yes? No?**
- ▶ Every *parse tree* has a unique *rightmost derivation*: **Yes? No?**
- ▶ Every *parse tree* has a unique *leftmost derivation*: **Yes? No?**

Derivations and Parse Trees

- ▶ Every $u \in L(G)$ has a unique *derivation*: **Yes? No?**
- ▶ Every $u \in L(G)$ has a unique *rightmost derivation*: **Yes? No?**
- ▶ Every $u \in L(G)$ has a unique *leftmost derivation*: **Yes? No?**

Uniqueness of Derivations (see Dragon Book, p.171)

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: **Yes?** No?

Uniqueness of Derivations (see Dragon Book, p.171)

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: Yes? **No?**
- ▶ Every *parse tree* has a unique *rightmost derivation*: **Yes?**
No?

Uniqueness of Derivations (see Dragon Book, p.171)

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: Yes? No?
- ▶ Every *parse tree* has a unique *rightmost derivation*: Yes?
No?
- ▶ Every *parse tree* has a unique *leftmost derivation*: Yes? No?

Uniqueness of Derivations (see Dragon Book, p.171)

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: Yes? No?
- ▶ Every *parse tree* has a unique *rightmost derivation*: Yes?
No?
- ▶ Every *parse tree* has a unique *leftmost derivation*: Yes? No?

Derivations and Parse Trees

- ▶ Every $u \in L(G)$ has a unique *derivation*: Yes? No?

Uniqueness of Derivations (see Dragon Book, p.171)

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: Yes? No?
- ▶ Every *parse tree* has a unique *rightmost derivation*: Yes?
No?
- ▶ Every *parse tree* has a unique *leftmost derivation*: Yes? No?

Derivations and Parse Trees

- ▶ Every $u \in L(G)$ has a unique *derivation*: Yes? No?
- ▶ Every $u \in L(G)$ has a unique *rightmost derivation*: Yes?
 No?

Uniqueness of Derivations (see Dragon Book, p.171)

Derivations and Parse Trees

- ▶ Every *parse tree* has a unique *derivation*: Yes? No?
- ▶ Every *parse tree* has a unique *rightmost derivation*: Yes?
No?
- ▶ Every *parse tree* has a unique *leftmost derivation*: Yes? No?

Derivations and Parse Trees

- ▶ Every $u \in L(G)$ has a unique *derivation*: Yes? No?
- ▶ Every $u \in L(G)$ has a unique *rightmost derivation*: Yes?
 No?
- ▶ Every $u \in L(G)$ has a unique *leftmost derivation*: Yes? No?

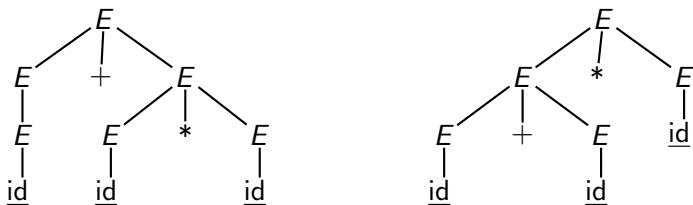
Ambiguity

Unambiguous Grammar: A grammar, G , is said to be unambiguous if $\forall u \in L(G), \exists$ exactly one leftmost or rightmost (canonical) derivation $S \Rightarrow^* u$.

Ambiguous Grammar: Otherwise, G is said to be ambiguous.

E.g., Grammar: $E \rightarrow E + E \mid E * E \mid (E) \mid \underline{id}$

Two parse trees for $u = \underline{id} + \underline{id} * \underline{id}$:



These are different syntactic interpretations of the input code.

Order of Evaluation of Parse Tree

Note: These are conventions, not theorems

- ▶ Code for a non-terminal is evaluated as a single “block”
 - ▶ I.e., cannot partially execute it, then execute something else, then evaluate the rest
 - ▶ A different parse tree would be needed to achieve that
 - ▶ E.g. 1: Non-terminal T enforces precedence of * over +
 - ▶ E.g. 2: $E \rightarrow E + T$ enforces left-associativity,
 $E \rightarrow T + E$ enforces right-associativity.
- ▶ Parse tree does not specify order of execution of code blocks
 - ▶ Must be enforced by code generated for parent block. Obey:
 - ▶ Operator (e.g, +) cannot be evaluated before operands
 - ▶ Associativity rules

Common Sources of Ambiguity

- ▶ There are two common forms of ambiguity:

- ▶ The “dangling else” form:

$E \rightarrow \text{if } E \text{ then } E \text{ else } E$

$E \rightarrow \text{if } E \text{ then } E$

$E \rightarrow \text{whatever}$

Example: `if a then if x then y else z ... to which if does the else belong?`

- ▶ The “double-ended recursion” form:

$E \rightarrow E + E$

$E \rightarrow E * E$

Example “`3 + 4 * 5`” ... is it “`(3 + 4) * 5`” or “`3 + (4 * 5)`”?

The Dangling-Else Ambiguity

Draw two separate parse trees for the “dangling else” example:

if a then if x then y else z

$E \rightarrow \text{if } E \text{ then } E \text{ else } E$

$E \rightarrow \text{if } E \text{ then } E$

$E \rightarrow \underline{\text{id}}$

Note: id is the common token for variable names a, x, y, z.

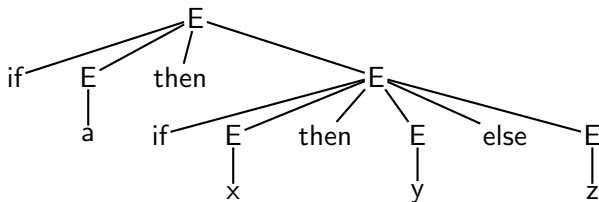
Parse Tree 1 for Dangling-Else

- ▶ Draw two separate parse trees for the “dangling else” example:

$E \rightarrow \text{if } E \text{ then } E \text{ else } E$
 $E \rightarrow \text{if } E \text{ then } E$
 $E \rightarrow \underline{\text{id}}$

if a then if x then y else z

Tree 1:



Parse Tree 2 for Dangling-Else

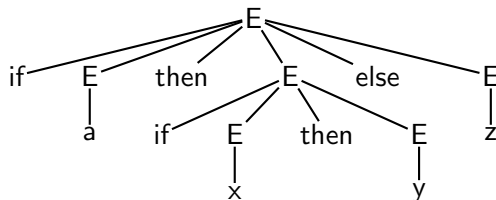
- ▶ Draw two separate parse trees for the “dangling else” example: `if a then if x then y else z`

$E \rightarrow \text{if } E \text{ then } E \text{ else } E$

$E \rightarrow \text{if } E \text{ then } E$

$E \rightarrow \underline{\text{id}}$

Tree 2:



Fixing Ambiguity

- ▶ Ambiguity can often be eliminated by thinking more carefully about what you are trying to express with your grammar.
- ▶ Double-ended recursion usually reveals a lack of precedence and associativity information.
- ▶ “Dangling else” usually matches with the nearest `if`. This can be encoded in the grammar. See §4.3 of the Dragon Book for details.
- ▶ Language fixes can eliminate this problem – for instance, keywords or symbols to identify the start and end of control blocks (i.e. `if-then-else-fi`)

Fixing Ambiguity

- ▶ The “double-ended recursion” form usually reveals a lack of precedence and associativity information. A technique called *stratification* often fixes this.
 - ▶ Left-recursive means “associates to the left”, similarly right-recursive.
 - ▶ Higher precedence rules occur lower in the grammar.

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{integer}$

Properties of Grammars

It is important to be able to say what properties a grammar has. Informally,

Epsilon Productions A production of the form “ $E \rightarrow \epsilon$ ”, where ϵ represents the empty string.

Right Linear Grammar Grammars where all the productions have the form
“ $E \rightarrow x E$ ” or “ $E \rightarrow x$ ”.

Left-Recursive Grammar a grammar that can generate
“ $E \rightarrow E + X$ ” (for example).
Similarly, “**right-recursive grammars.**”

Ambiguous Grammar More than one parse tree is possible for a specific sentence.

Epsilon Productions

- ▶ Sometimes we want to specify that a symbol can become nothing.
- ▶ Example: “ $E \rightarrow \epsilon$ ”
- ▶ Another example:
 - $S \rightarrow NP \text{ verb } PP$
 - $NP \rightarrow \text{det } A \text{ noun}$
 - $PP \rightarrow \text{prep } NP$
 - $A \rightarrow \text{adjective } A$
 - $A \rightarrow \epsilon$

This says that adjectives are an optional part of noun phrases.

Right Linear Grammars

- ▶ A *right linear* grammar is one in which all the productions have the form “ $E \rightarrow x E$ ” or “ $E \rightarrow x$ ”.
- ▶ This corresponds to the *regular languages*.
- ▶ Example: regular expression $(10)^*23$ describes same language as this grammar:

$$A_0 \rightarrow 1A_1 \mid 2A_2$$

$$A_1 \rightarrow 0A_0$$

$$A_2 \rightarrow 3A_3$$

$$A_3 \rightarrow \epsilon$$

Left-Recursive Grammars

- ▶ A grammar is *recursive* if a symbol being produced (the one on the left-hand side) reappears in the right hand side after one or more steps.

Example: “ $E \rightarrow \text{if } E \text{ then } E \text{ else } E$ ”

- ▶ A grammar is *left-recursive* if the production symbol appears as the first symbol on the right-hand-side (in one or more steps).

Example: “ $E \rightarrow E + F$ ”

- ▶ Example with indirect left recursions (two steps):

Example: $A \rightarrow Bx$
 $B \rightarrow Ay$

Representing Parse Trees in OCaml

Our end goal of parsing will be to build an OCaml data structure representing the parsed form of a program. Although we will focus more on this later, our strategy will be to

- ▶ create an OCaml datatype for each syntactic category in the language
- ▶ this datatype will most likely be *mutually recursive*, to represent the inherent recursive structure of most language definitions
- ▶ generate an OCaml term, using these mutually recursive types, representing the parsed form of the program – containment in a type constructor shows that the contained items are children of the containing item in the AST