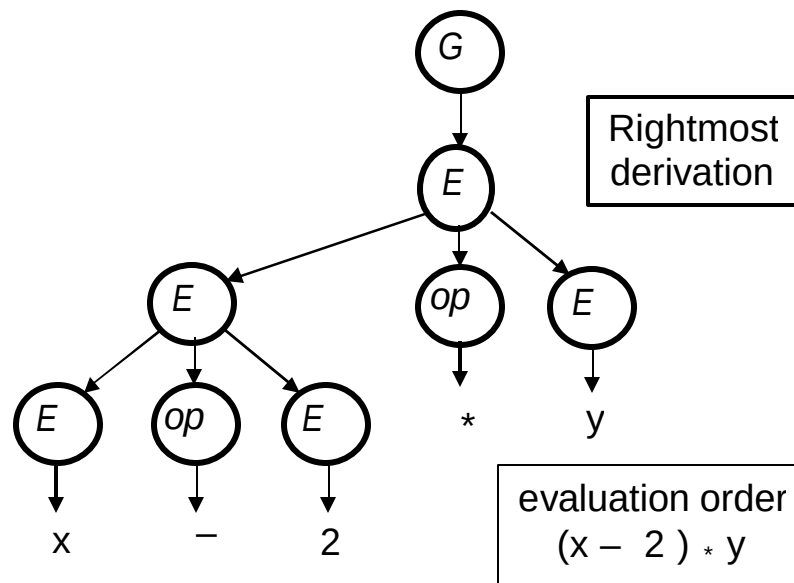
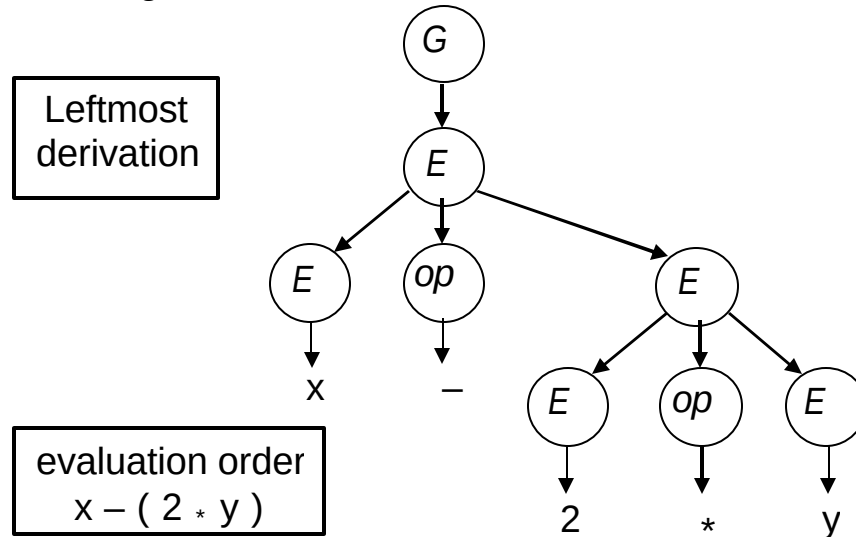


Lecture 12

Parse Trees

The derivations can be represented in a tree-like fashion. The interior nodes contain the non-terminals used during the derivation



Precedence

These two derivations point out a problem with the grammar. It has no notion of *precedence*, or implied order of evaluation. The normal arithmetic rules say that multiplication has higher precedence than subtraction. To add precedence, create a non-

terminal for each *level of precedence*. Isolate corresponding part of grammar to force parser to recognize high precedence sub-expressions first. Here is the revised grammar:

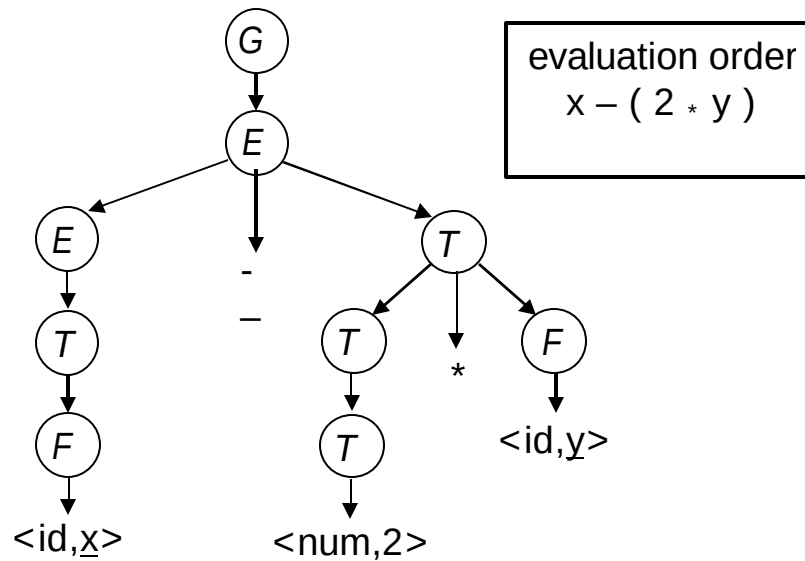
	1	Goal	?	expr
level two	2	expr	?	expr + term
	3			expr - term
	4			term
	5	term	?	term * factor
level one	6			term / factor
	7			factor
	8	factor	?	number
	9			Id

This grammar is larger and requires more rewriting to reach some of the terminal symbols. But it encodes expected precedence. Let's see how it parses

$$x - 2 * y$$

Rule	Sentential Form
-	Goal
1	Expr
3	expr - term
5	expr - term * factor
9	expr - term * <id,y>
7	expr - factor * <id,y>
8	expr - <num,2> * <id,y>
4	term - <num,2> * <id,y>
7	factor - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

This produces same parse tree under leftmost and rightmost derivations



Both leftmost and rightmost derivations give the *same expression* because the grammar directly encodes the desired precedence.

Ambiguous Grammars

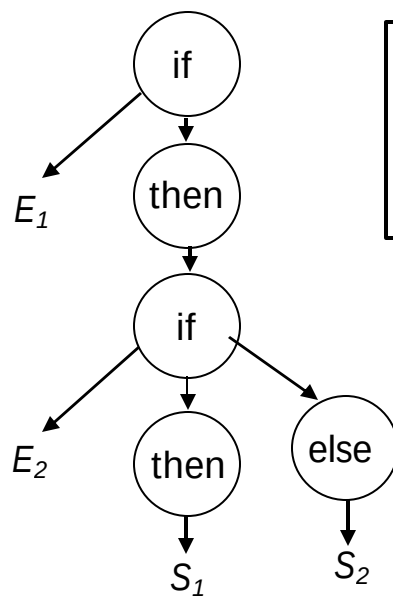
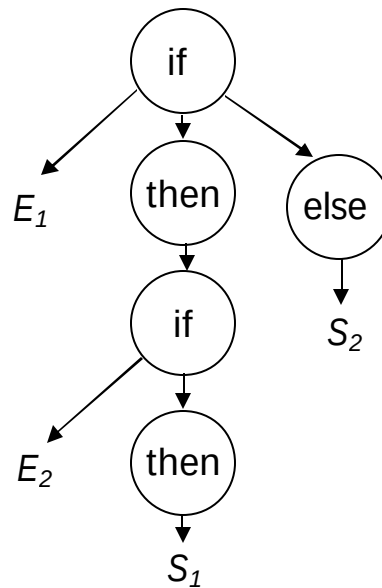
If a grammar has more than one leftmost derivation for a single sentential form, the grammar is *ambiguous*. The leftmost and rightmost derivations for a sentential form may differ, even in an *unambiguous* grammar. Let's consider the classic if-then-else example

$Stmt \ ? \ \underline{if} \ Expr \ \underline{then} \ Stmt$ $\quad \quad \ \underline{if} \ Expr \ \underline{then} \ Stmt \ \underline{else} \ Stmt$ $\quad \quad \ \dots \ other \ stmts \ \dots$
--

The following sentential form has two derivations:

$\underline{if} \ E_1 \ \underline{then} \ \underline{if} \ E_2 \ \underline{then} \ S \ \underline{else} \ S_2$

Production 1, then
 Production 2:
 if E_1 then
 if E_2 then S_1
 else S_2



Production 2, then
 Production 1:
 if E_1 then
 if E_2 then S_1
 else S_2

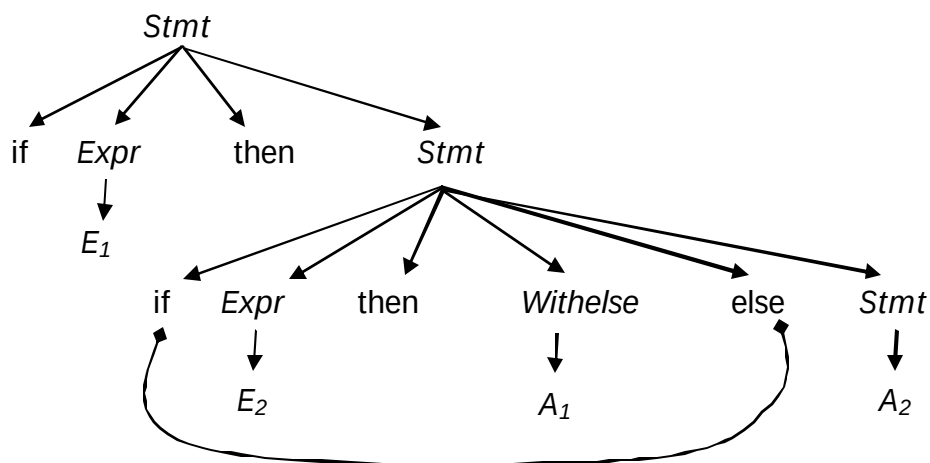
The convention in most programming languages is to match the else with the most recent if.

We can rewrite grammar to avoid generating the problem and match each else to innermost unmatched if:

- | | | |
|--------------------|---|---|
| 1. <i>Stmt</i> | ? | If <i>E</i> then <i>Stmt</i> |
| 2. | | If <i>E</i> then <i>WithElse</i> else <i>Stmt</i> |
| 3. | | <i>Assignment</i> |
| 4. <i>WithElse</i> | ? | If <i>E</i> then <i>WithElse</i> else <i>WithElse</i> |
| 5. | | <i>Assignment</i> |

Let derive the following using the rewritten grammar:

if *E*₁ then if *E*₂ then *A*₁ else *A*₂



This binds the else controlling *A*₂ to inner if

Context-Free Grammars

We have been using the term context-free without explaining why such rules are in fact “free of context”. The simple reason is that non-terminals appear by themselves to the left of the arrow in context-free rules:

$A \rightarrow a$

The rule $A \rightarrow a$ says that *A* may be replaced by *a* *anywhere*, regardless of where *A* occurs. On the other hand, we could define a context as pair of strings *b*, *g*, such that a rule would apply only if *b* occurs before and *g* occurs after the non-terminal *A*. We would write this as

$b A g \rightarrow b a g$

Such a rule in which $a \neq e$ is called a *context-sensitive* grammar rule. We would write this as

$$bAg \neq b a g$$

Such a rule in which $a = e$ is called a *context-sensitive* grammar rule

Parsing Techniques

There are two primary parsing techniques: top-down and bottom-up.

Top-down parsers

A top-down parser starts at the root of the parse tree and grows towards leaves. At each node, the parser picks a production and tries to match the input. However, the parser may pick the wrong production in which case it will need to backtrack. Some grammars are backtrack-free.

Bottom-up parsers

A bottom-up parser starts at the leaves and grows toward root of the parse tree. As input is consumed, the parser encodes possibilities in an internal state. The bottom-up parser starts in a state valid for legal first tokens. Bottom-up parsers handle a large class of grammars