Plan for Today

Ambiguous Grammars

Disambiguating ambiguous grammars

Left-factoring for predictive parsers

REMINDERS

- HW5 will be posted tonight and is due Monday.
- PA3 is due on Monday October 17th
- HW4 feedback will be provided by Saturday night

Ambiguous Grammars

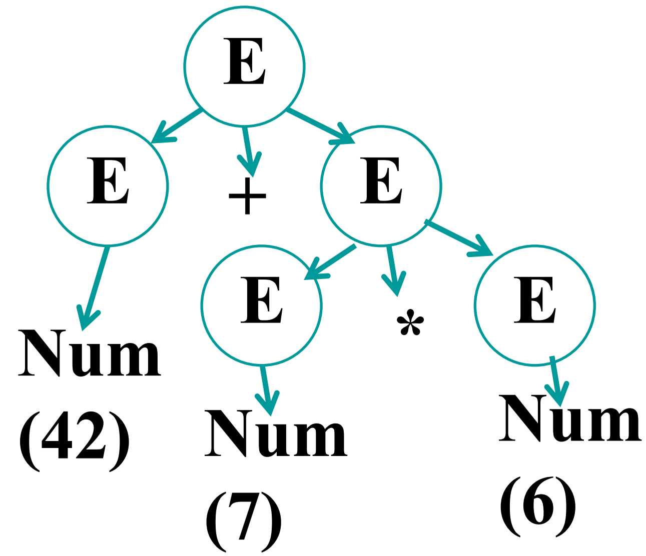
Ambiguous grammar:

>2+ parse trees for 1 sentence

Expression grammar

E → E * E
E → E + E
E → E - E
E → (E)
E → ID
E → NUM

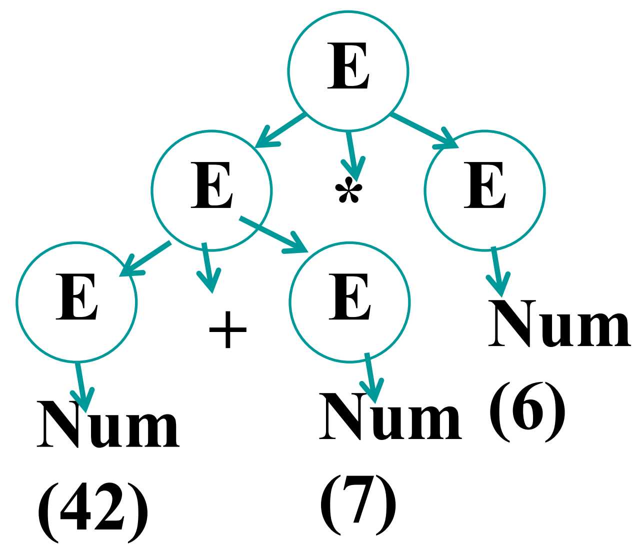
parse tree 1



String

42 + 7 * 6

parse tree 2



what about 42-7-6?

Goal: disambiguate the grammar

Cause

- the grammar did not specify the precedence nor the associativity of the operators +,-,*

Some Options

- Keep the ambiguous grammar, but add extra directives to the parser (many LR parser generators can do this).
- Rewrite the grammar, making the precedence and associativity explicit in the grammar.
- Use a general purpose parser and deal with $O(N^3)$ for parsing.

Unambiguous grammar for simple expressions

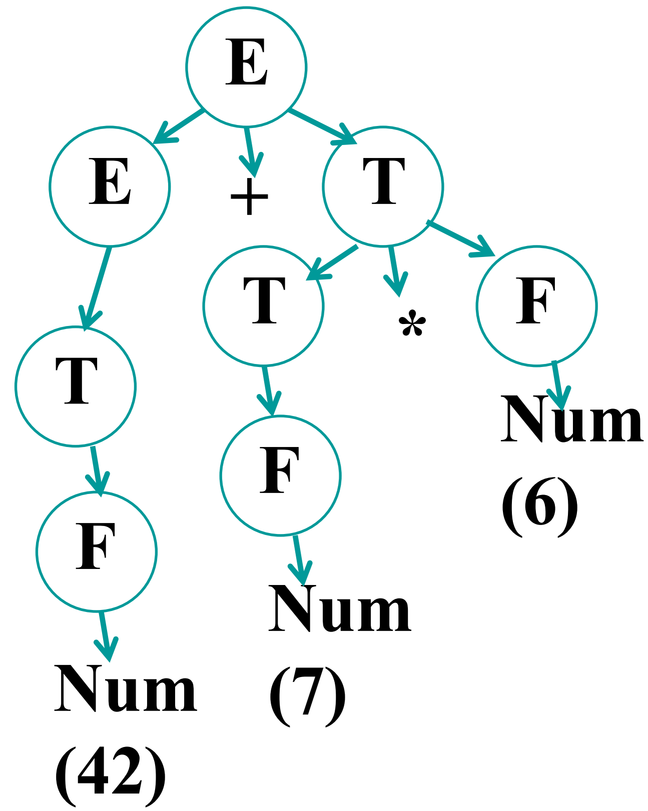
Grammar

E → E + T | E-T | T

T → T * F | F

F → (E) | ID | NUM

parse tree



String

42+7*6

How is the precedence encoded?

How is the associativity encoded?

An Example including AST Construction

Grammar

E → E + E | E ^ E | ID | NUM

String

2 ^ 2 ^ N

When Predictive Parsing works, when it does not

What about our expression grammar:

E → E + T | E-T | T
T → T * F | F
F → (E) | ID | NUM

Predictive parser

- The E method cannot decide looking one token ahead whether to predict E+T, E-T, or T.
- Same problem for T.

Predictive parsing works for grammars where

The first terminal symbol of each sub expression provides enough information to decide which production to use.

Terminology Interlude

LL(k) parser

- Left-to-right parsing of input
- Leftmost derivation of the sentence is found while parsing
- k tokens of lookahead needed to parse
- Top down parsing
- Example parser generators: ANTLR, JavaCC,

LR(k) parser

- Left-to-right parsing of input
- reversed Rightmost derivation
- k tokens of lookahead needed to parse
- Example parser generators: Yacc, bison, JavaCup, and Happy

See wikipedia entry for “Comparison of parser generators”.

Left recursion and Predictive parsing

What happens to the recursive descent parser if we have a left recursive production rule, e.g. **$E \rightarrow E{+}T|T$**

E calls E calls E forever

To eliminate left recursion we rewrite the grammar:

from:	**to:**
$E \rightarrow E + T \mid E{-}T \mid T$	**$E \rightarrow T\ E'$**
$T \rightarrow T * F \mid F$	**$E' \rightarrow +\ T\ E' \mid -\ T\ E' \mid \varepsilon$**
$F \rightarrow (\ E\ ) \mid ID \mid NUM$	**$T \rightarrow F\ T'$**
	$T' \rightarrow *\ F\ T' \mid \varepsilon$
	$F \rightarrow (\ E\ ) \mid ID \mid NUM$

replacing left recursion $X \rightarrow X\gamma \mid \alpha$ (where α does not start with X) with right recursion, $X \rightarrow \alpha X'$, $X' \rightarrow \gamma X' \mid \varepsilon$, that can be produced right recursively. Now we can compute nullable, FIRST and FOLLOW, and produce an LL(1) predictive parse table.

Left Factoring

Left recursion does not work for predictive parsing. Neither does a grammar that has a non-terminal with two productions that start with a common phrase, so we left factor the grammar:

$$S \rightarrow \alpha\beta_1 \qquad S \rightarrow \alpha S'$$

$$S \rightarrow \alpha\beta_2 \quad \xrightarrow{\textit{Left refactor}} \quad S' \rightarrow \beta_1 \mid \beta_2$$

E.g.: if statement:

S → IF t THEN S ELSE S | IF t THEN S | o

becomes

S → IF t THEN S X | o

X→ ELSE S | ε

When building the predictive parse table, there will be a multiple entries.

WHY?

Dangling else problem: ambiguity

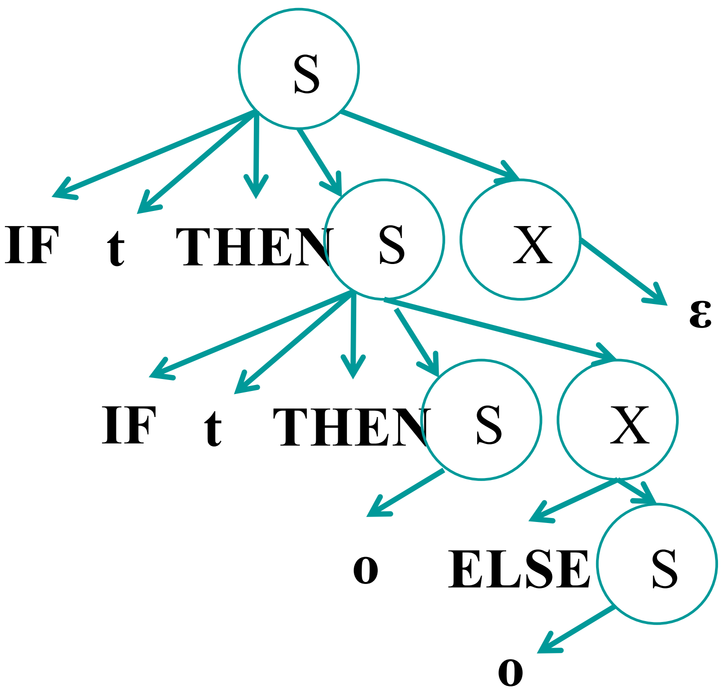
Given

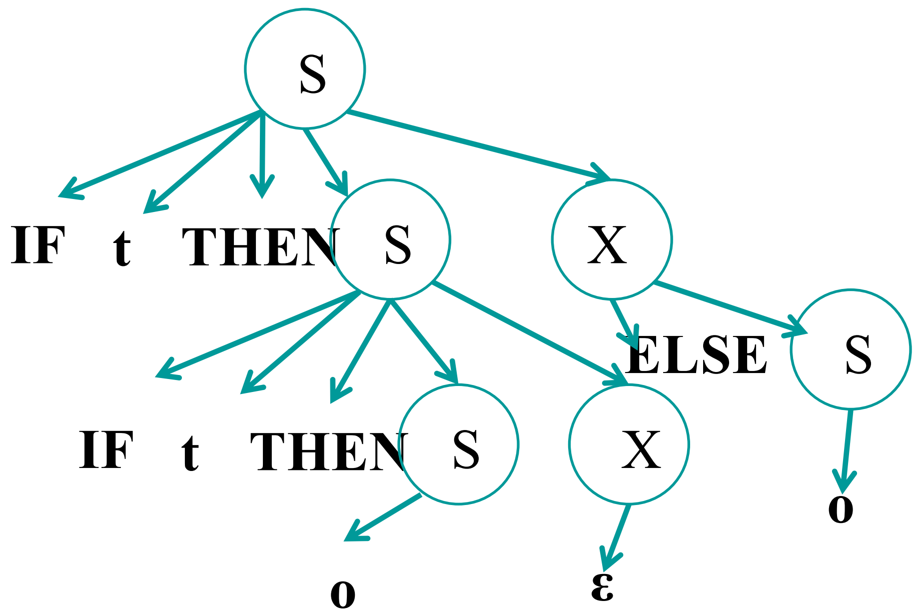
S → IF t THEN S X | o

X→ ELSE S | ε

construct two parse trees for

IF t THEN IF t THEN o ELSE o

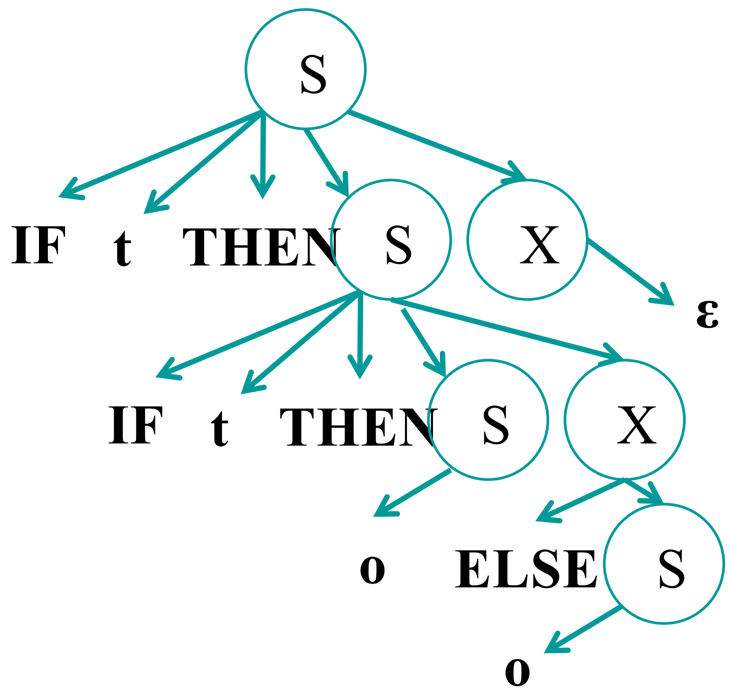




Which is the correct parse tree? (C, Java rules)

Dangling else disambiguation

The correct parse tree is:



We can get this parse tree by removing the X→ε rule in the multiple entry slot in the parse tree.