Plan for Today

Ambiguous Grammars

Disambiguating ambiguous grammars

Left-factoring for predictive parsers

Ambiguous Grammars

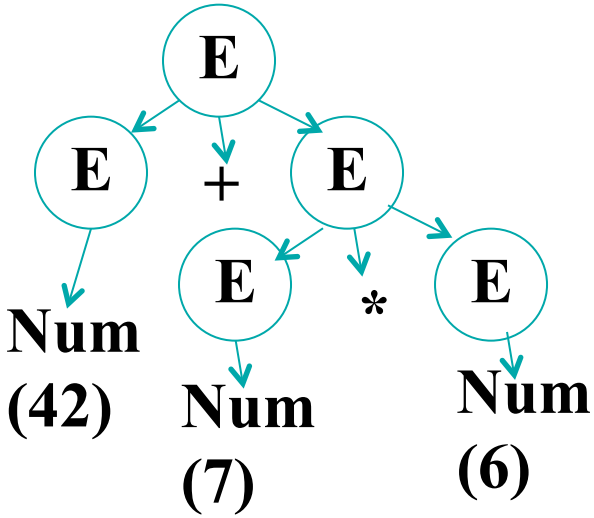
Ambiguous grammar:

>2+ parse trees for 1 sentence

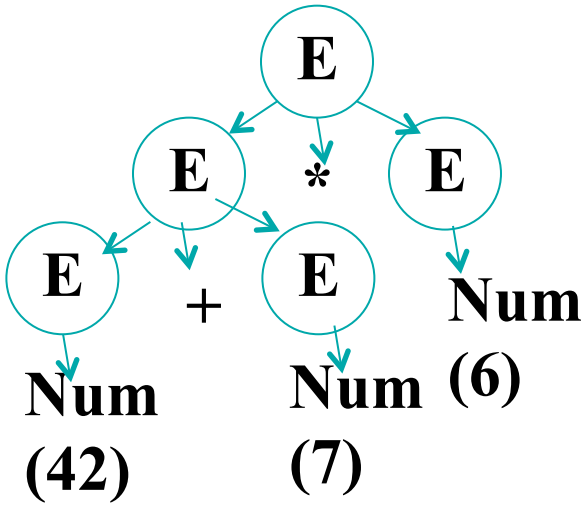
Expression grammar

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

parse tree 1



parse tree 2



String

42 + 7 * 6

what about 42-7-6?

Goal: disambiguate the grammar

Cause

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

Two Options

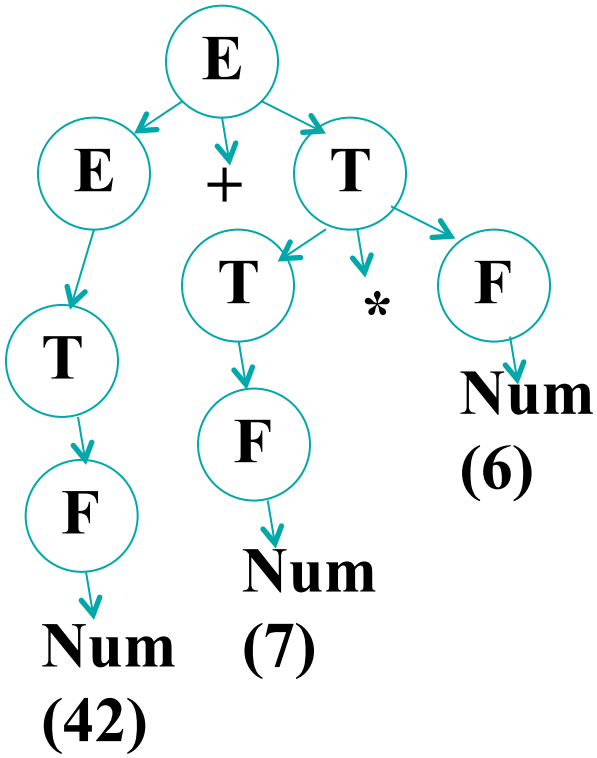
- keep the ambiguous grammar, but add extra directives to the parser (many LALR parser generators can do this)
- Rewrite the grammar, making the precedence and associativity explicit in the grammar.

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?

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.

Left recursion and Predictive parsing

What happens to the recursive descent parser if we have a left recursive production rule, e.g. E → E+T|T

E calls E calls E forever

To eliminate left recursion we rewrite the grammar:

from:	**to:**
E → E + T | E-T | T	**E →T E'**
T → T * F | F	**E' → + T E' | - T E' | ε**
F → (E) | ID | NUM	**T → F T'**
	T' → * F T' | ε
	F → (E) | ID | NUM

replacing left recursion X→Xγ | α (where α does not start with X) with right recursion, X→ α X', X' →γX' | **ε,** that can be produced right recursively. Now we can augment the grammar (S→E$), 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:

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

$$\xrightarrow{\textit{Left refactor}}$$

$$S \rightarrow \alpha\beta_2 \qquad\qquad S' \rightarrow \beta_1 \mid \beta_2$$

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

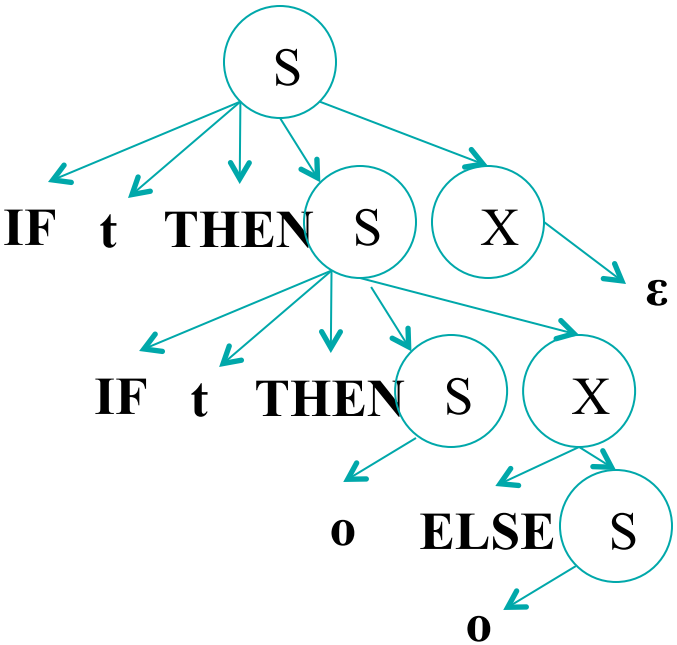
Given

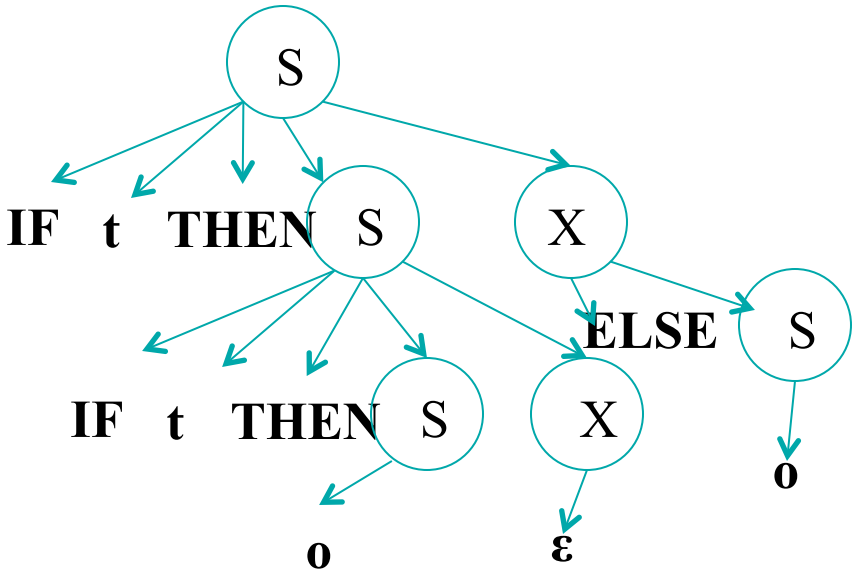
S → IF t THEN S X | o

X→ ELSE S | ε

construct two parse trees for

IF t THEN IF t THEN o ELSE o

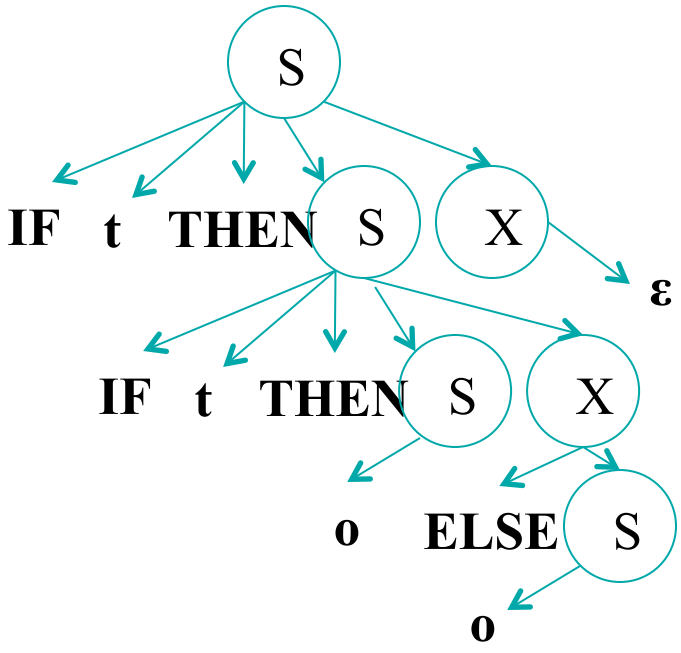




Which is the correct parse tree? (C, Java rules)

Dangling else disambiguation

The correct parse tree is:



We can get this parse tree by removing the $X \rightarrow \varepsilon$ rule in the multiple entry slot in the parse tree.