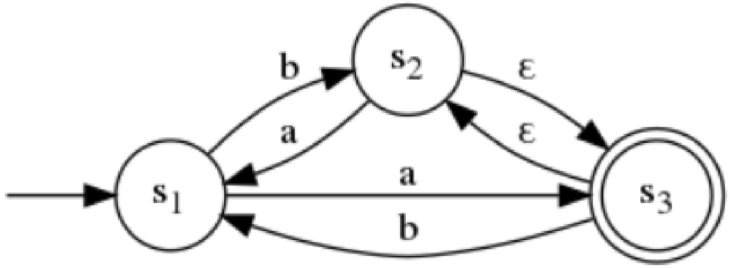
Checkin 4

These questions use the following state machine:



1. Draw out an equivalent finite automaton to the given one after applying the ε-elimination technique described in class.
2. Is the state machine that results from 1. an NFA or DFA?
3. If the machine is a DFA, create an NFA that recognizes the same language. If the machine is an NFA, create a DFA that recognizes the same language.

Administrivia

Housekeeping

- Quiz 1 NEXT Friday
- Project 1 due **<u>NEXT</u>** Wednesday (or that Thursday for 1 late mark or that Friday for 2 late marks)
  - Sorta/Kinda delayed – kept original start date, gave extra time for kickstart
  - Additional delay so it isn't due on Labor Day

University of Kansas | Drew Davidson

EECS 665

COMPILER CONSTRUCTION

4 – Syntactic Ambiguity

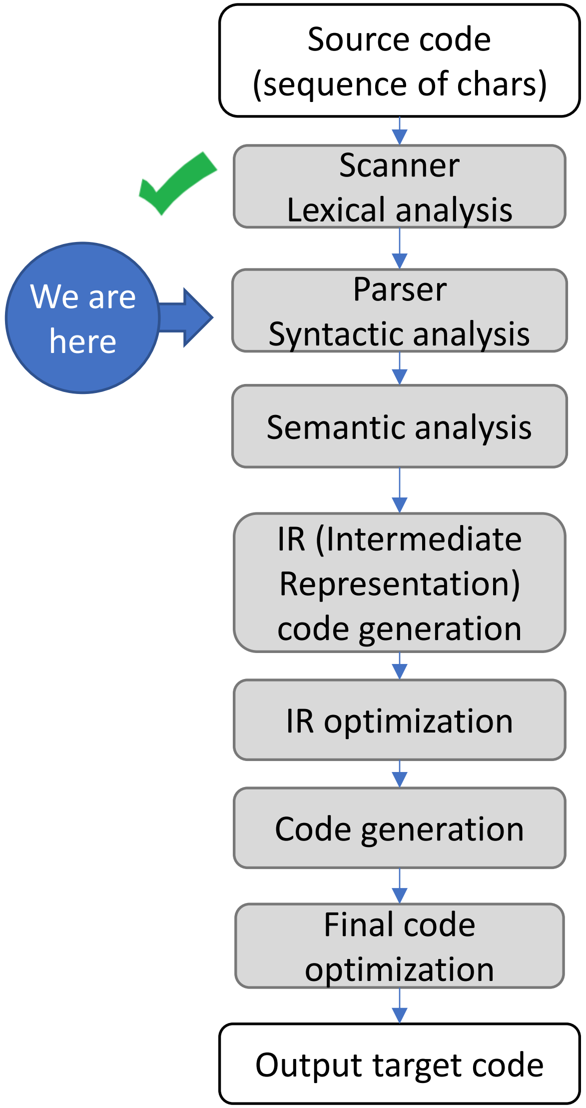
I *Think* We're Mostly Done with Review

Lecture 4 – Syntactic Ambiguity


COMPILER
Code Generation
Execution
Runtime Environment
Optimization
Intermediate Representation
SDD
Parsing
Semantics
Lexical Analysis
Syntactic Definiton

Compiler Construction

Progress Pics



Quick Comparison

Lexer VS Parser

Lexer VS Parser

Definition: RegExes

Input: char stream

Output: token stream

Recognizer: DFA

Definition: CF Grammar

Input: token stream

Output: parse tree

Recognizer: PDA

Let's breeze through this
(I bet you've seen it already)

Lexer VS Parser

Definition: RegExes

Input: char stream

Output: token stream

Recognizer: DFA

Translator: Tokenizer

Definition: CF Grammar

Input: token stream

Output: parse tree

Recognizer: PDA

Translator: Parser

Last Time

Review Lecture 3 – Defining Syntax

From regular language recognition to stream tokenization

- Lexer achieved!

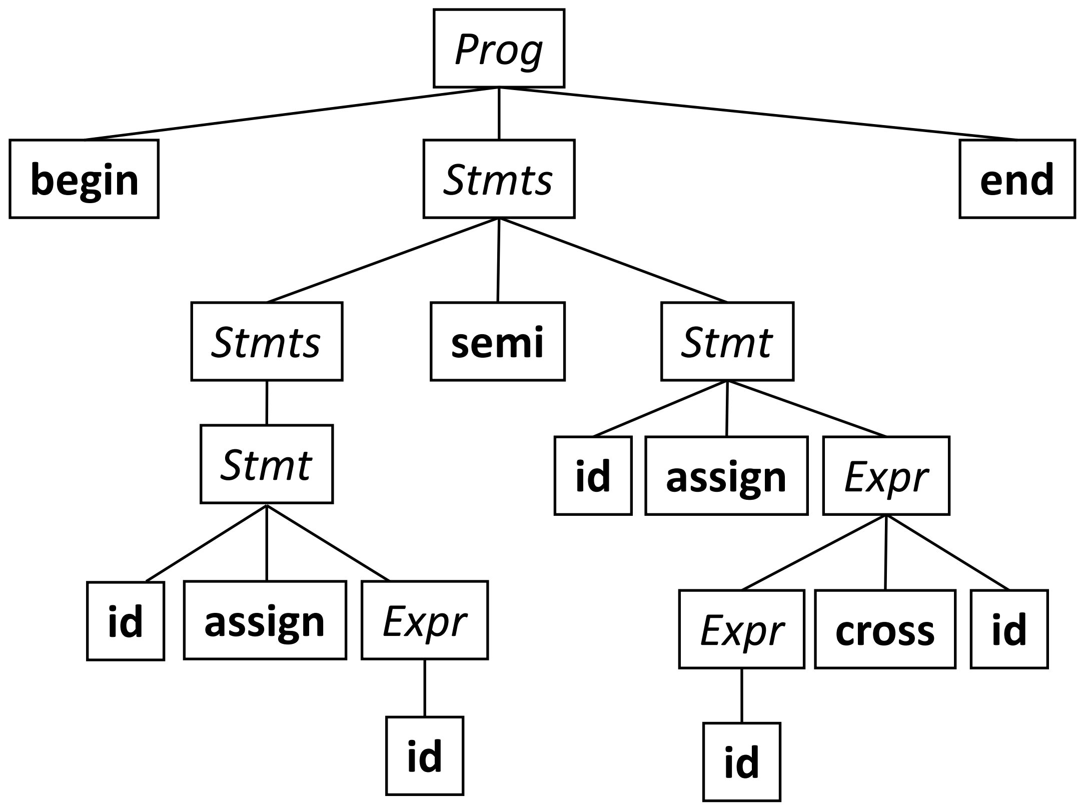
Reviewed Context-Free Grammars

- Language Definition via grammar
- BNF notation
- Parse trees

CFGs - usable for syntactic definition

BNF is a good way for programmers to communicate syntax

Parse Tree



Derivation Sequence

Prog

Prod. 1

⇒ **begin** Stmts **end**

Prod. 2

⇒ **begin** Stmts **semi** Stmt **end**

Prod. 3

⇒ **begin** Stmt semi Stmt **end**

Prod. 4

⇒ **begin id assign** *Expr* **semi** *Stmt* **end**

Prod. 4

⇒ **begin id assign** *Expr* **semi id assign** *Expr* **end**

Prod. 5

⇒ **begin id assign id semi id assign** *Expr* **end**

Prod. 6

⇒ **begin id assign id semi id assign** *Expr* **cross id end**

Prod. 5

⇒ **begin id assign id semi id assign id cross id end**

Productions

1. *Prog* ::= **begin** *Stmts* **end**
2. *Stmts* ::= *Stmts* **semi** *Stmt*
3. | *Stmt*
4. *Stmt* ::= **id assign** *Expr*
5. *Expr* ::= **id**
6. | *Expr* **cross id**

Lecture Outline

Lecture 4 –Syntactic Ambiguity

Compiler Concerns in Syntactic Analysis

- Recognizing CF-Languages
- Constraining Derivations
- Ambiguous Syntax
  - What it is
  - How we resolve it

Constraining Derivations

Lecture 4 –Syntactic Ambiguity

Goal: 1-1 map derivation sequences to sentences

- Reason #1: predictable, reliable algorithms
- Reason #2: we care about the structure of sentences

Consistency is key

Derivation Order

Syntactic Ambiguity: Constraining Derivations

Grammar

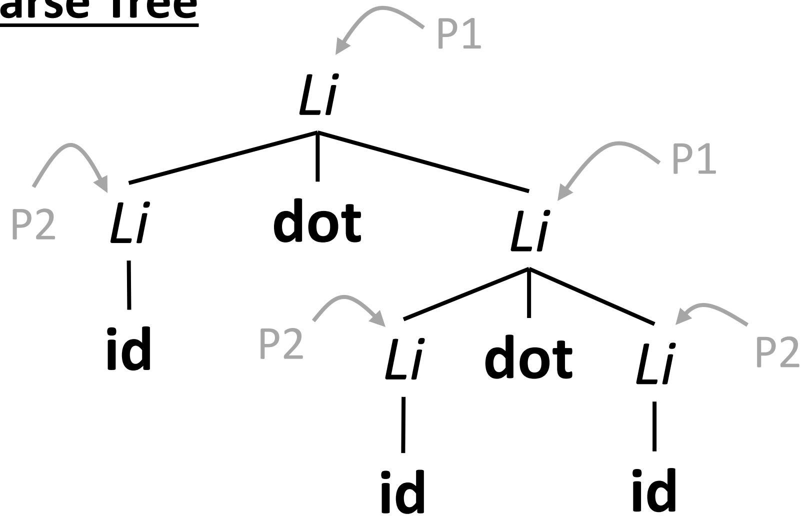
P1: *Li* ::= *Li* **dot** *Li*

P2: | **id**

Token stream

id dot id dot id

Parse Tree



How can we derive the sentence?

- Multiple orderings of the same productions to generate sentence
  - That's bad for consistency!
- We usually commit to a single ordering
  - *Leftmost derivation*: expand the leftmost nonterminal
  - *Rightmost derivation*: expand the rightmost nonterminal

Derivation Order

Syntactic Ambiguity: Constraining Derivations

Grammar

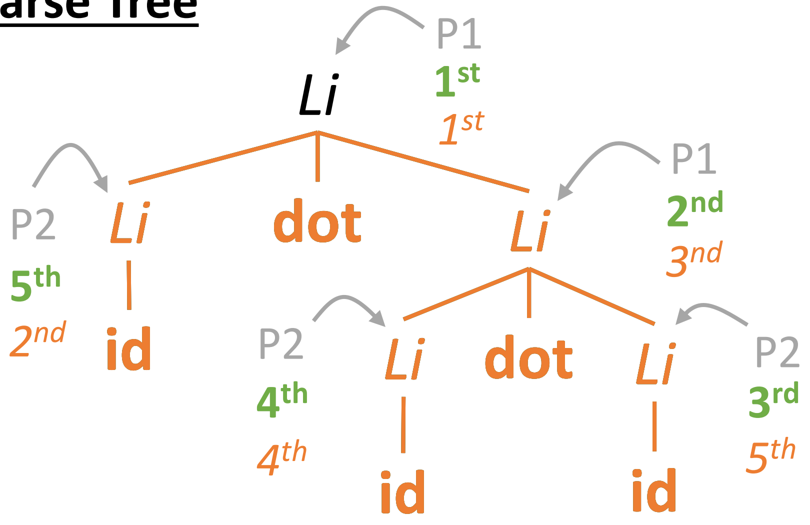
P1: *Li* ::= *Li* **dot** *Li*

P2: | **id**

Token stream

id dot id dot id

Parse Tree



How can we derive the sentence?

- Multiple orderings of the same productions to generate sentence
  - That's bad for consistency!
- We usually commit to a single ordering
  - *Leftmost derivation*: expand the leftmost nonterminal
  - *Rightmost derivation*: expand the rightmost nonterminal

Leftmost derivation

Li ⇒ *Li* **dot** *Li*
⇒ **id dot** *Li*
⇒ **id dot** *Li* **dot** *Li*
⇒ **id dot id dot** *Li*
⇒ **id dot id dot id**

Rightmost derivation

Li ⇒ *Li* **dot** *Li*
⇒ *Li* **dot** *Li* **dot** *Li*
⇒ *Li* **dot** *Li* **dot id**
⇒ *Li* **dot id dot id**
⇒ **id dot id dot id**

Bigger Problems On the Horizon

Syntactic Ambiguity: Constraining Derivations

May be able to produce different parse trees even under the same derivation order!

The Real Problem

Syntactic Ambiguity: Constraining Derivations

Grammar

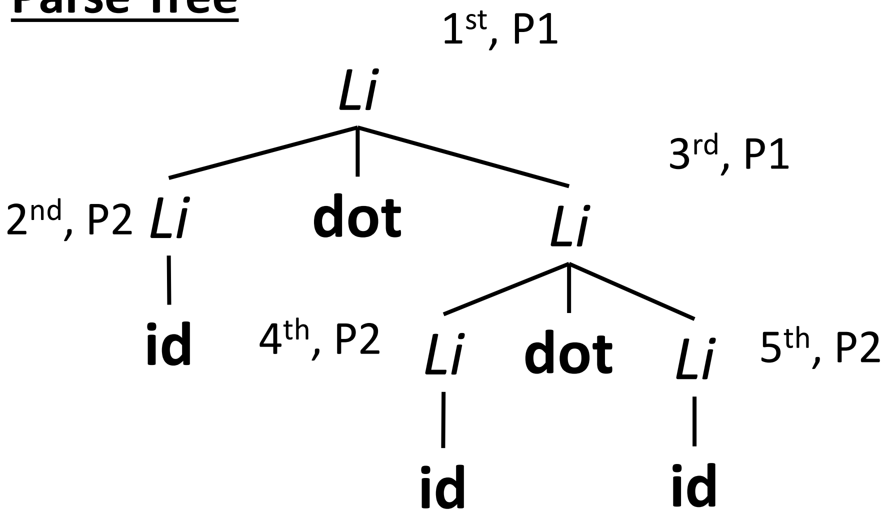
P1. *Li* ::= *Li* **dot** *Li*

P2. | **id**

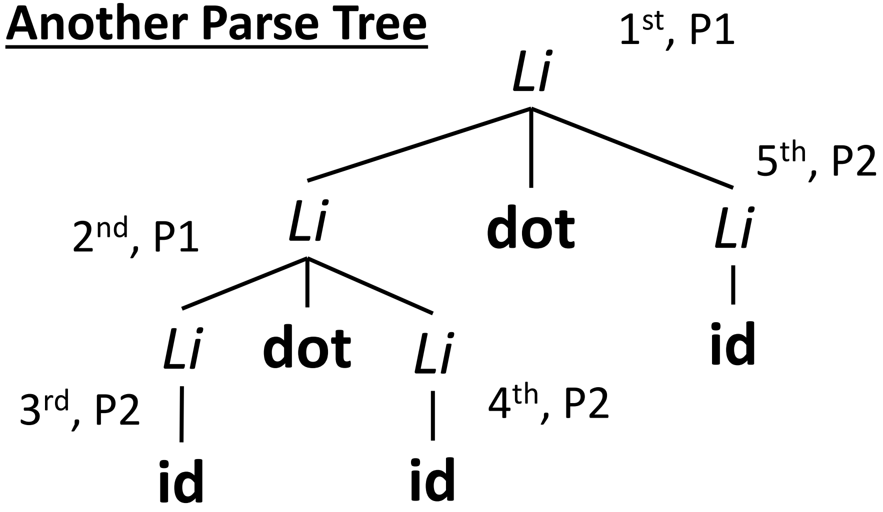
Input

id dot id dot id

Parse Tree



Another Parse Tree



Same (leftmost) derivation order, different productions

The Real Problem

Syntactic Ambiguity: Constraining Derivations

- Even with a fixed derivation order, possible to derive the same sentence in multiple ways

Syntactic Ambiguity (Definition):

For Grammar $\mathcal{G}$ and string w

- $\mathcal{G}$ is ambiguous if

  >1 leftmost derivation of w

  $\equiv$

  >1 rightmost derivation of w

  $\equiv$

  > 1 parse tree for w

Why is syntactic ambiguity a problem?

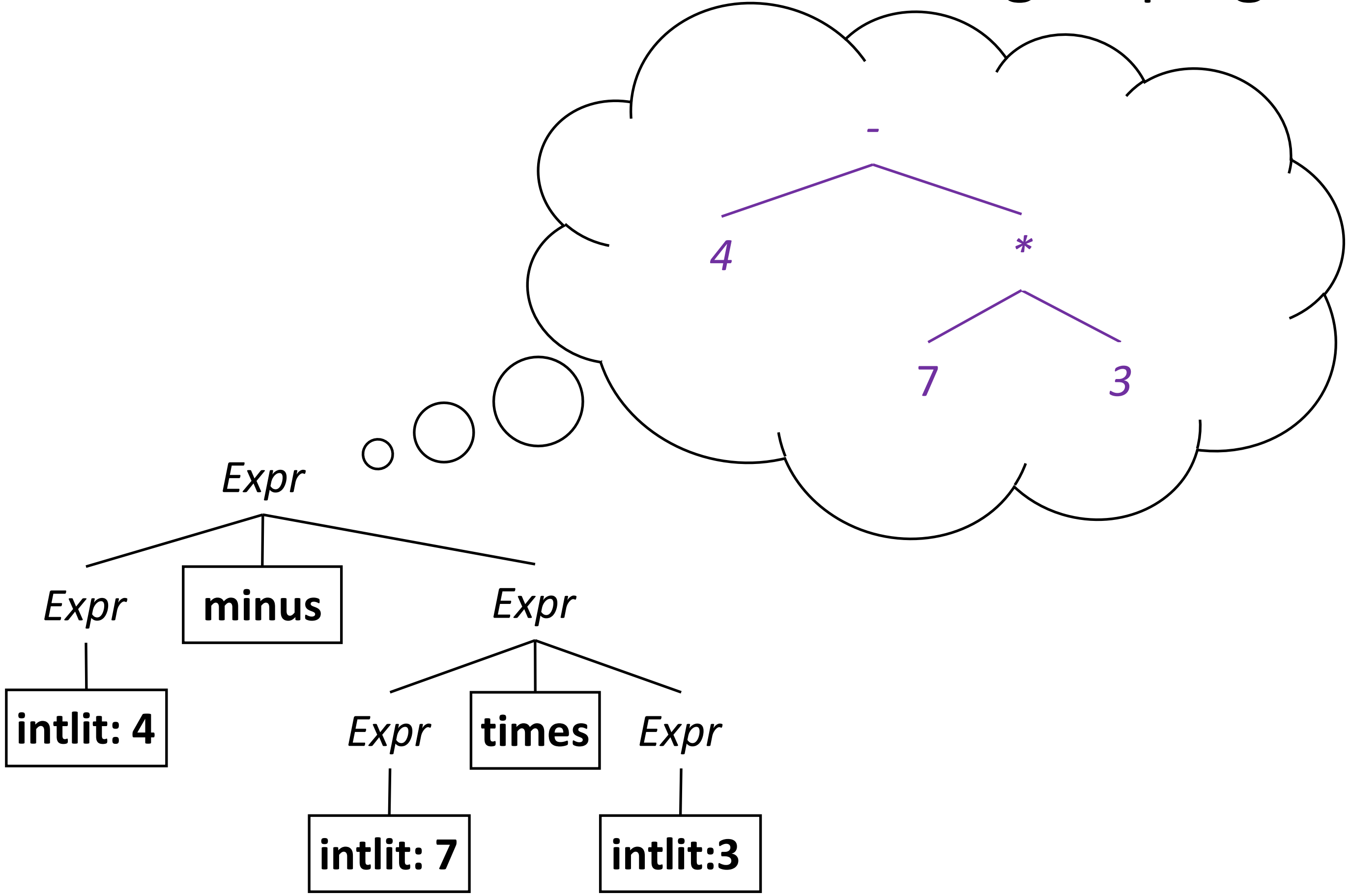
- Allows inconsistency
- Unreliable tree structure

Particularly troubling

Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

- Maintain the integrity of the underlying operations
- Intuitively, parse tree should mimic semantic grouping



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Grammar

P1. *Expr* ::= **intlit**

P2. | *Expr* **minus** *Expr*

P3. | *Expr* **times** *Expr*

Char Stream:

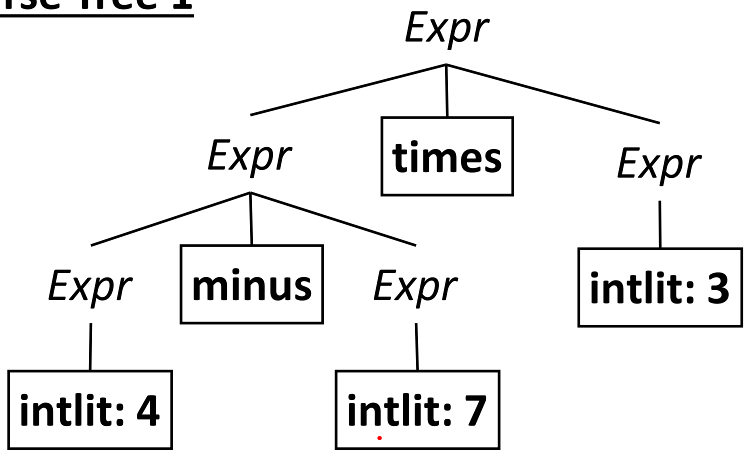
4 – 7 * 3

Token Stream:

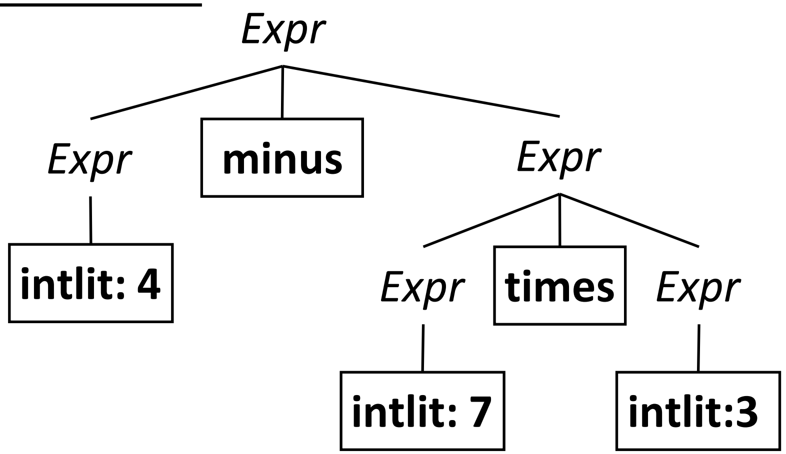
intlit minus intlit times intlit

Which tree is correct?

Parse Tree 1



Parse Tree 2



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Grammar

P1. *Expr* ::= **i**

P2. | *Ex*

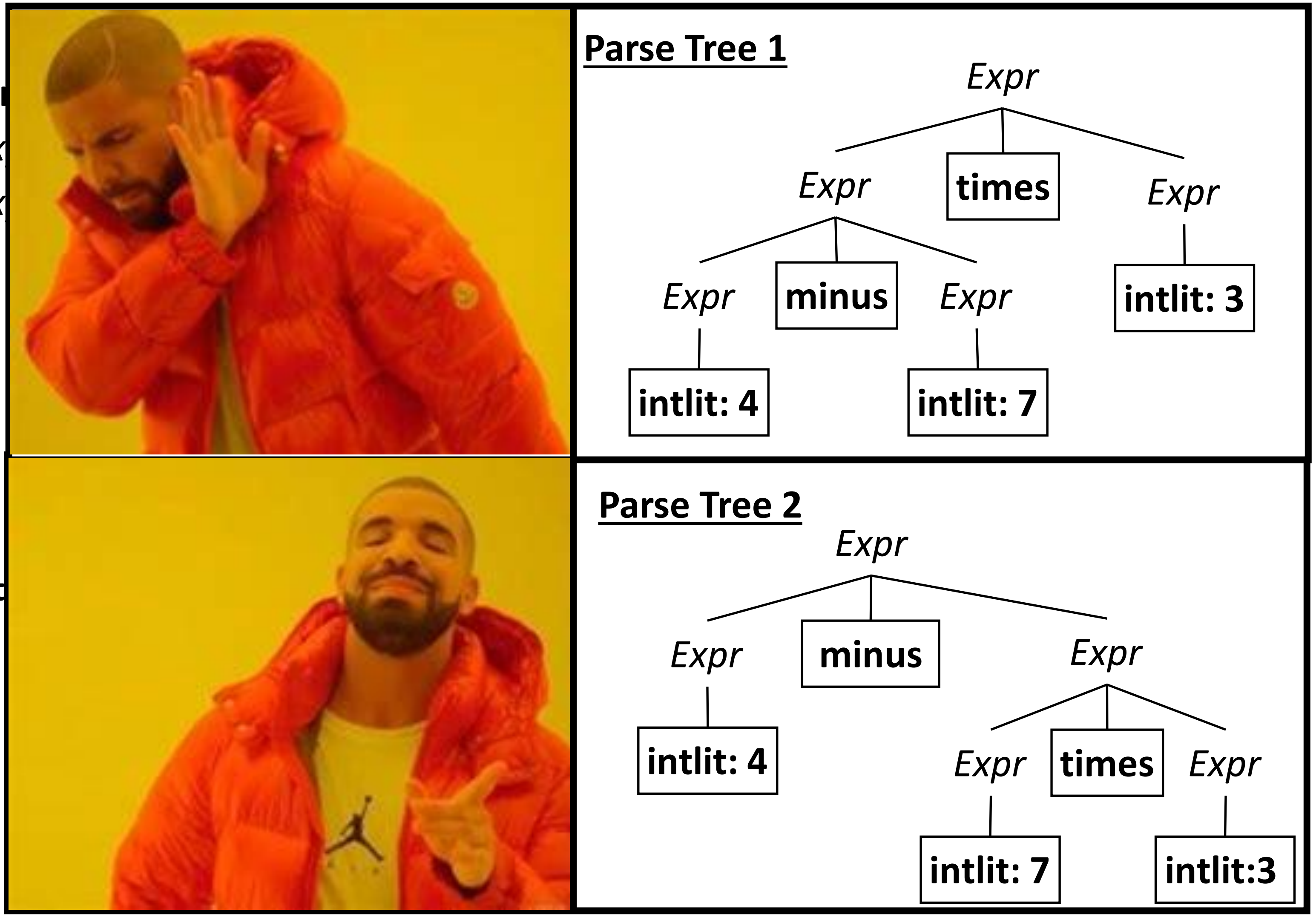
P3. | *Ex*

Char Stream:

(4 – 7) * 3

Token Stream:

intlit minus int



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Grammar

P1. *Expr* ::= **intlit**

P2. | *Expr* **minus** *Expr*

P3. | *Expr* **times** *Expr*

Char Stream:

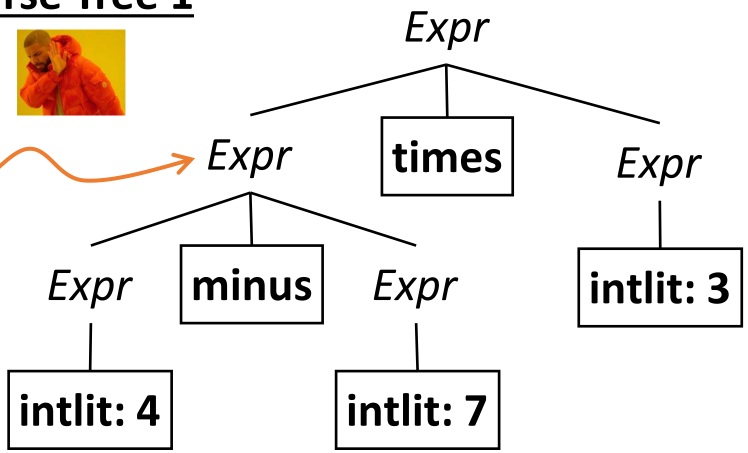
4 – 7 * 3

Token Stream:

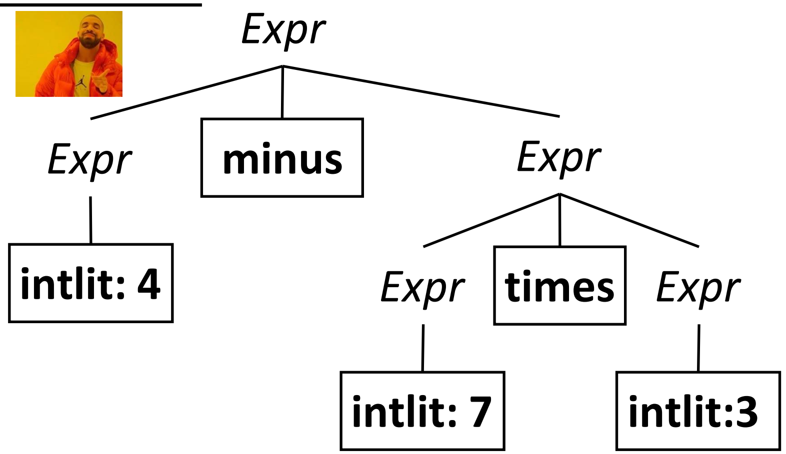
intlit minus intlit times intlit

Intuitive problem: Depth
Minus subtrees
(can be) produced
beneath times subtrees

Parse Tree 1



Parse Tree 2



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Grammar

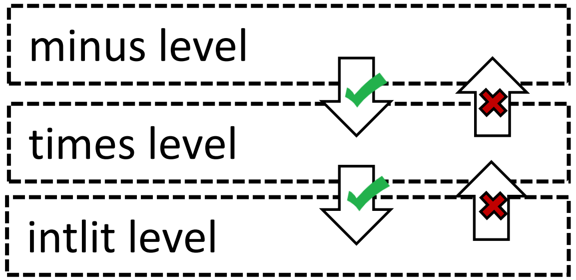
P1. *Expr* ::= **intlit**
P2. | *Expr* **minus** *Expr*
P3. | *Expr* **times** *Expr*

Intuitive problem: Depth
Minus subtrees
(can be) produced
beneath times subtrees

Fixed Grammar

P1. *Expr* ::= *Expr* **minus** *Expr*
P2. | *Term*
P3. *Term* ::= *Term* **times** *Term*
P4. | *Factor*
P5. *Factor* ::= **intlit**

Solution: Embed precedence "levels" into the grammar



- You can go down a level "for free"
- You can't go up a level

Use a new nonterminal for each precedence level

Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Fixed Grammar

P1. *Expr* ::= *Expr* **minus** *Expr*
P2. | *Term*
P3. *Term* ::= *Term* **times** *Term*
P4. | *Factor*
P5. *Factor* ::= **intlit**

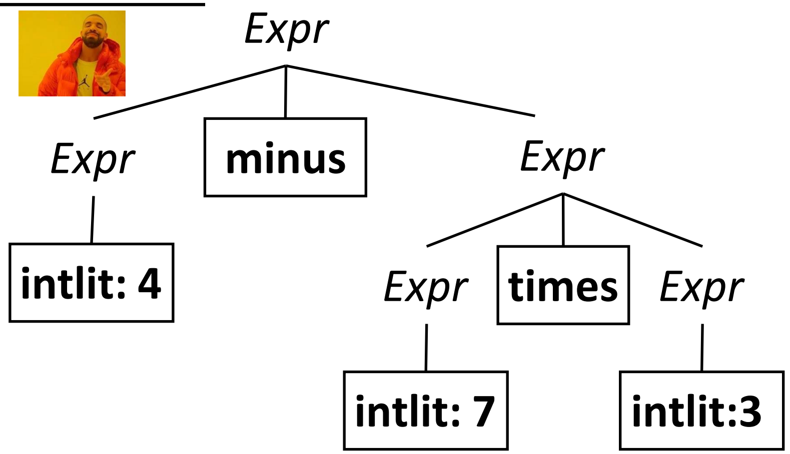
Char Stream:

4 – 7 * 3

Token Stream:

intlit minus intlit times intlit

Parse Tree 2



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

<u>Fixed Grammar</u>

P1. *Expr* ::= *Expr* **minus** *Expr*
P2. | *Term*
P3. *Term* ::= *Term* **times** *Term*
P4. | *Factor*
P5. *Factor* ::= **intlit**

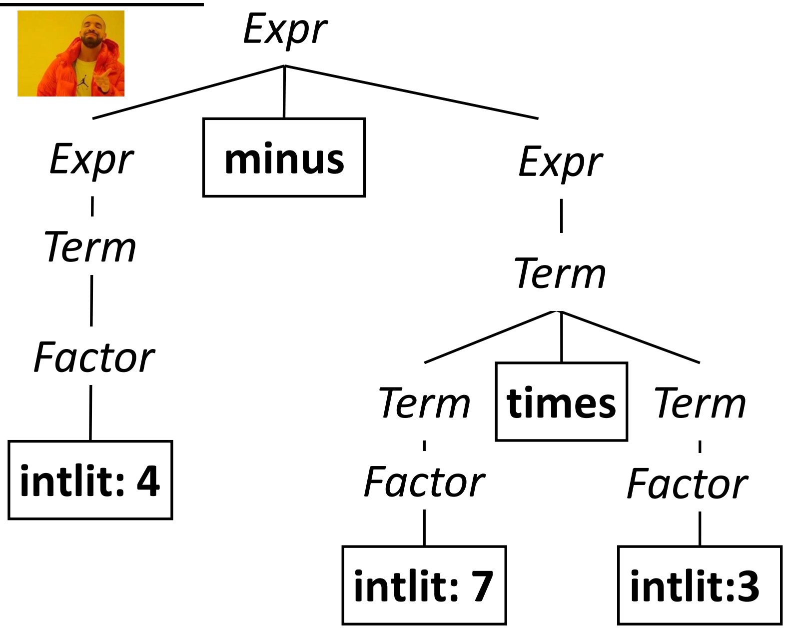
<u>Char Stream:</u>

4 – 7 * 3

<u>Token Stream:</u>

intlit minus intlit times intlit

<u>Parse Tree 2</u>



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Fixed Grammar

P1. *Expr* ::= *Expr* **minus** *Expr*
P2. | *Term*
P3. *Term* ::= *Term* **times** *Term*
P4. | *Factor*
P5. *Factor* ::= **intlit**

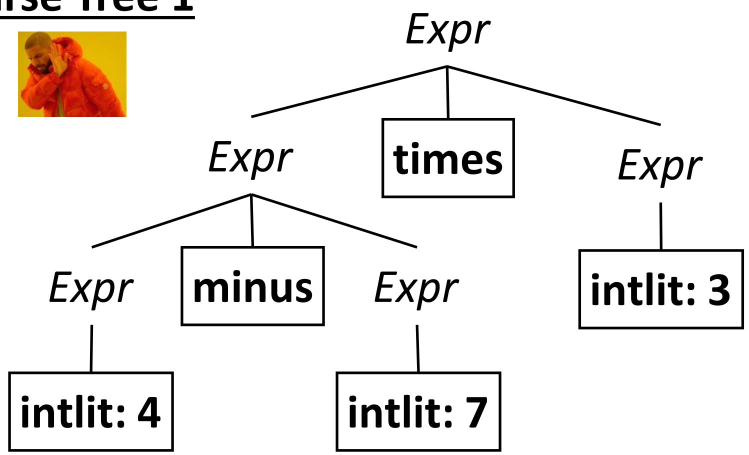
Char Stream:

4 – 7 * 3

Token Stream:

intlit minus intlit times intlit

Parse Tree 1



Parse Tree Structure

Syntactic Ambiguity: Constraining Derivations

Fixed Grammar

P1. *Expr* ::= *Expr* **minus** *Expr*

P2. | *Term*

P3. *Term* ::= *Term* **times** *Term*

P4. | *Factor*

P5. *Factor* ::= **intlit**

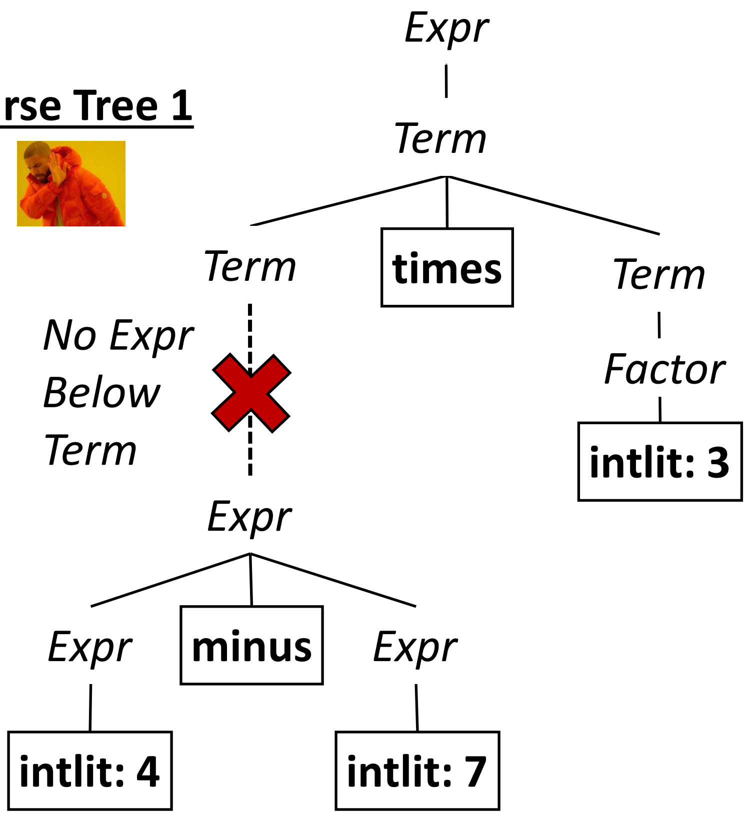
Char Stream:

4 – 7 * 3

Token Stream:

intlit minus intlit times intlit

Parse Tree 1



Did We Fix All Ambiguity?

Resolving Syntactic Ambiguity

Fixed Grammar

P1. *Expr* ::= *Expr* **minus** *Expr*
P2. | *Term*
P3. *Term* ::= *Term* **times** *Term*
P4. | *Factor*
P5. *Factor* ::= **intlit**

Nope!

Grammar is still ambiguous!

Char Stream:

3 – 4 – 7

Did We Fix All Ambiguity?

Resolving Syntactic Ambiguity

Fixed Grammar

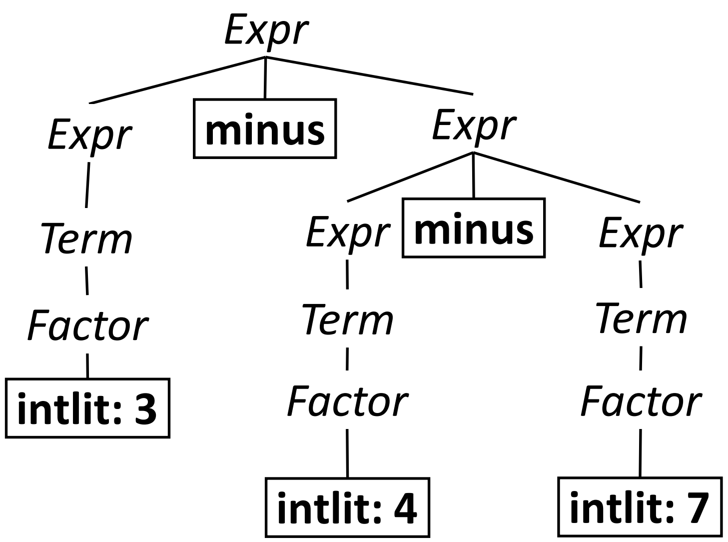
P1. *Expr* ::= *Expr* **minus** *Expr*
P2. | *Term*
P3. *Term* ::= *Term* **times** *Term*
P4. | *Factor*
P5. *Factor* ::= **intlit**

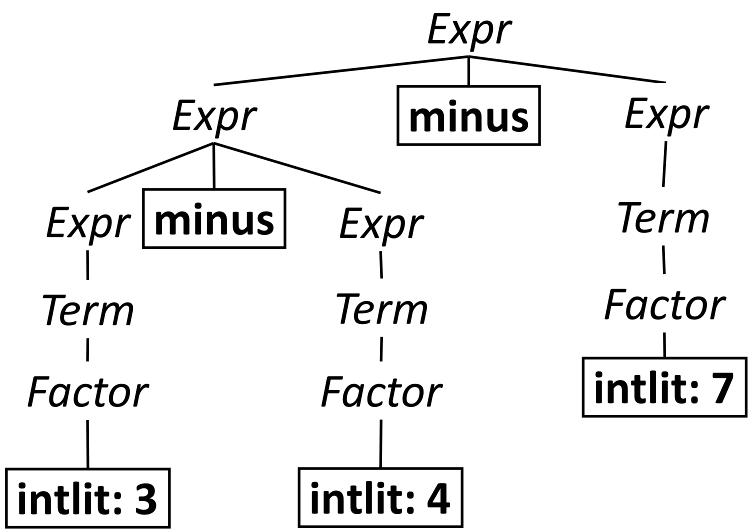
Nope!

Grammar is still ambiguous!

Char Stream:

3 – 4 – 7





A Second Ambiguity Issue

Syntactic Ambiguity: Constraining Parse Trees

Solved!
enforce depth with nonterminal "levels"

Precedence

- We want correct behavior on
  `4 – 7 * 9`
- High-precedence operators deeper in the tree

Idea...
enforce with production "skew"

Associativity

- We want correct behavior on
  `4 – 7 – 9`
- Minus should be *left associative*:
  `a – b – c – d = ((a - b) - c) - d`

A Second Ambiguity Issue

Syntactic Ambiguity: Constraining Parse Trees

*Idea…
enforce with
production "skew"*

Associativity

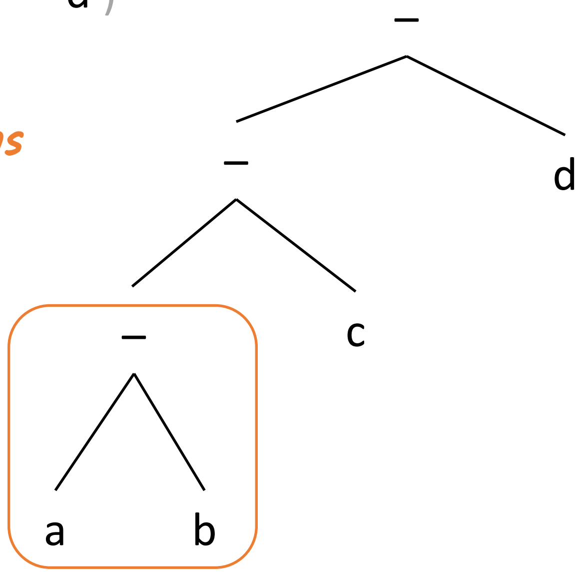
- We want correct behavior on

  `4 – 7 – 9`

- Minus should be *left associative*:

  `a – b – c – d` = ((a - b) - c) - d)

*Deeply nested parens
=
Deep in the tree*



Enforcing Associativity

Resolving Syntactic Ambiguity

- Recognize left-associative operators with *left-recursive* productions
- Recognize right-associative operators with *right-recursive* productions

- **A grammar is *recursive* in (nonterminal) *X* if**

$X \overset{+}{\Rightarrow} \alpha X \gamma$ for arbitrary strings of symbols α and γ

Enforcing Associativity

Resolving Syntactic Ambiguity

- Recognize left-associative operators with *left-recursive* productions
- Recognize right-associative operators with *right-recursive* productions

- **A grammar is *recursive* in (nonterminal) *X* if**

  $X \overset{+}{\Rightarrow} \alpha X \gamma$ for arbitrary strings of symbols α and γ

- **A grammar is *left-recursive* in *X* if**

  $X \overset{+}{\Rightarrow} X \gamma$ for non-empty string of symbols γ

Enforcing Associativity

Resolving Syntactic Ambiguity

- Recognize left-associative operators with *left-recursive* productions
- Recognize right-associative operators with *right-recursive* productions

- **A grammar is *recursive* in (nonterminal) *X* if**

  $X \overset{+}{\Rightarrow} \alpha X \gamma$ for arbitrary strings of symbols α and γ

- **A grammar is *left-recursive* in *X* if**

  $X \overset{+}{\Rightarrow} X\gamma$ for non-empty string of symbols γ

  immediately

- **A grammar is *right-recursive* in *X* if**

  $X \overset{\cancel{+}}{\Rightarrow} \alpha X$ for non-empty string of symbols α

Enforcing Associativity
Resolving Syntactic Ambiguity

- Recognize left-associative operators with *left-recursive* productions
- Recognize right-associative operators with *right-recursive* productions

Expr → *Expr* **minus** ~~*Expr*~~ *Term*

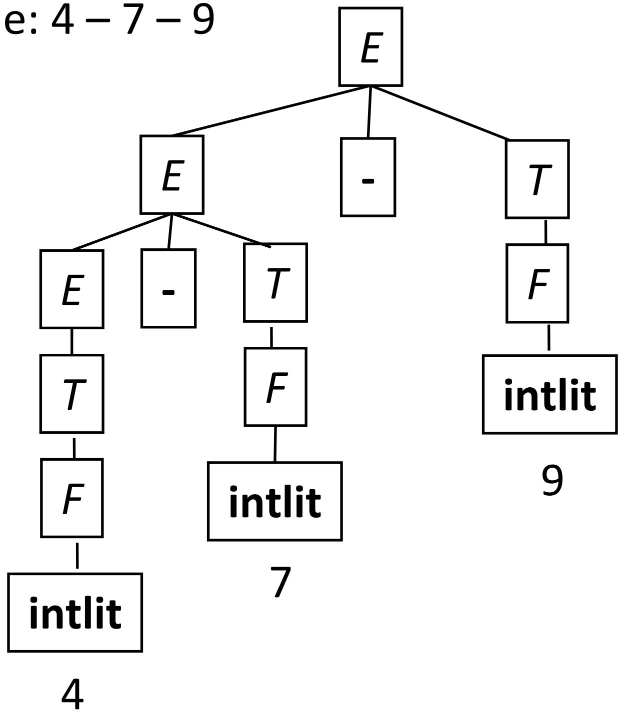
| *Term*

Term → *Term* **times** ~~*Term*~~ *Factor*

| *Factor*

Factor → **intlit** | **lparen** *Expr* **rparen**

Example: 4 – 7 – 9



Enforcing Associativity

Resolving Syntactic Ambiguity

Expr → *Expr* **minus** *Term*

| *Term*

Term → *Term* **times** *Factor*

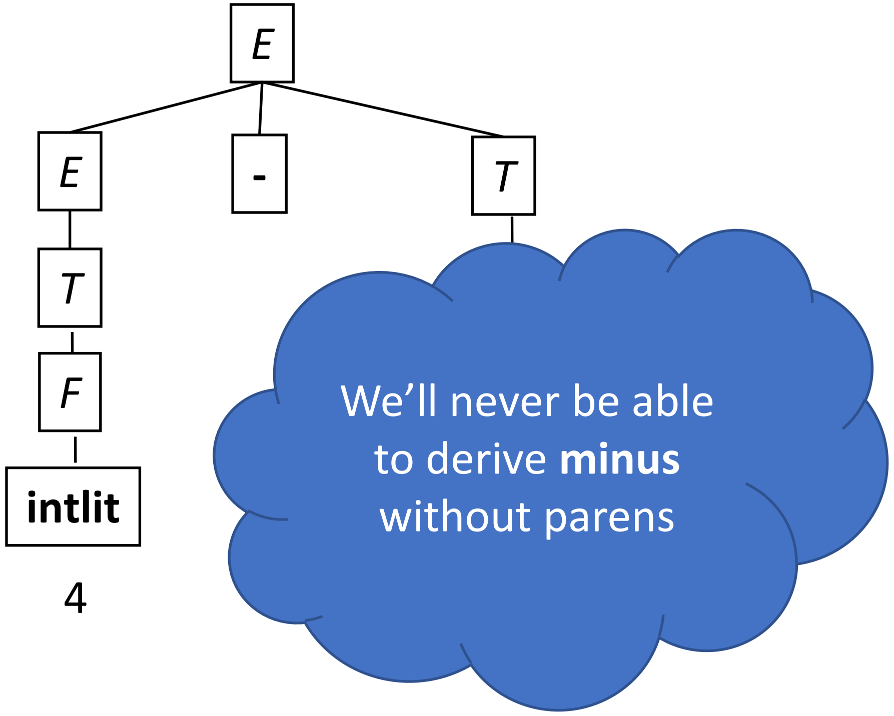
| *Factor*

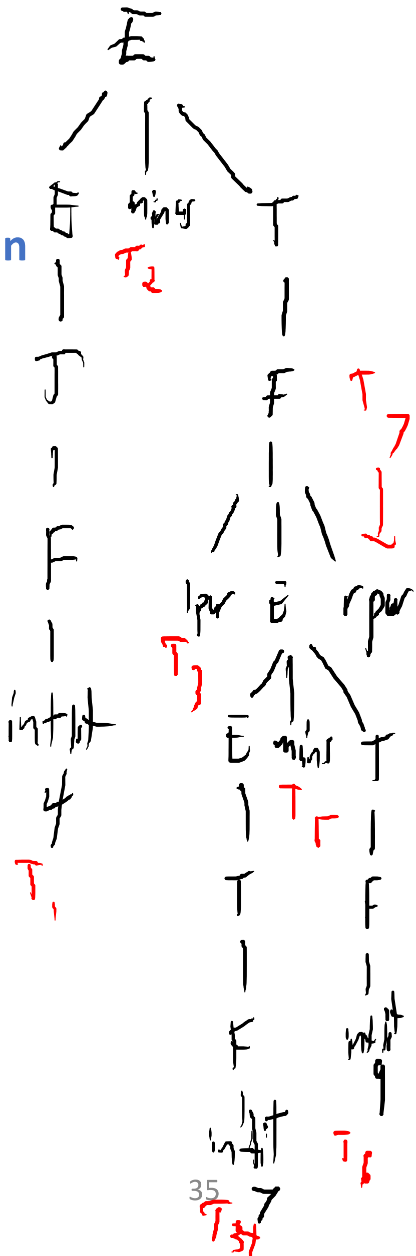
Factor → **intlit**

| lpar Expr rpar

Example: 4 – 7 – 9

Let's try to re-build the ~~wrong~~ parse tree again





End of Lecture
Syntactic Ambiguity

2 ^ 3 ^ 4

$2^{3^{4}}$

Summary

- We set out to make derivations and parse trees map 1-1 with any string in a grammar
- We note several forms of ambiguity and consider fixes:
  - Derivation order: use a fixed strategy
  - Syntactic ambiguity (precedence) – "stratify" the grammar
  - Syntactic ambiguity (associativity) – "lean" the grammar

Next Time

Parse tree translation: ascribing meaning to parse trees