Lecture C:

Parsing & Context-Free Grammars

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Reminders
  - Project partner signup (due Tuesday night)
    - Who’s still looking for a partner
  - HW1 (due Thursday night)
    - * vs. * similarly, please avoid messy \e\s\c\a\p\e\s — this is why I’ve been using $\underline{*}$ underlining to distinguish concrete characters. Please add a short comment to help your grader
    - (Re-)read the notes at the top of the homework when you think you’re “done” 🙂

Administrivia (Wednesday)

- Partner Signups — Done?
  - I will make an Ed post once the starter repositories are set up later today
- Reminder: HW1 (due Thursday night)
- Section Thu — Very Important for Project Setup!
  - If using IntelliJ make sure to read the README, **BEFORE** you open up the project in the IDE. This will save you a lot of headache
- Reading: This lecture (3.1-3.2), Next lecture (3.4)

Outline

Parsing Overview

Context-Free Grammars

Ambiguous Grammars

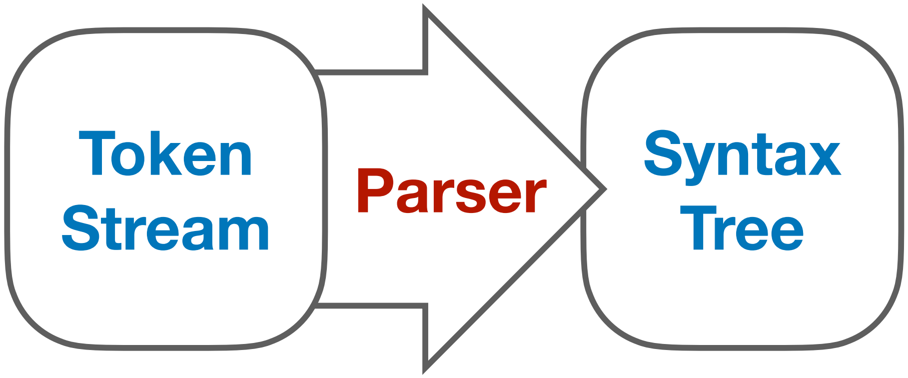
Outline

Parsing Overview

Context-Free Grammars

Ambiguous Grammars

Parsing*



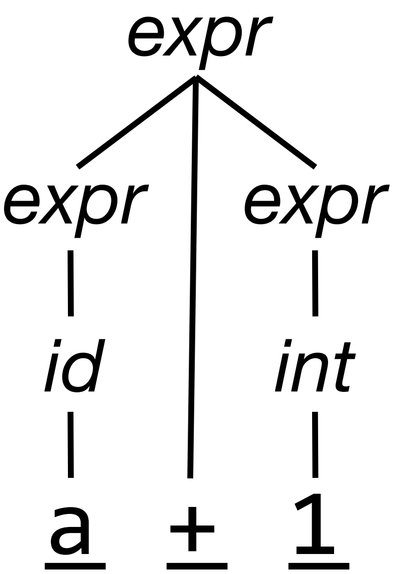
- Input: token stream
- Output: abstract syntax tree
- **Abstract Syntax Tree** (AST)
  - captures the grammatical structure of a program
  - primary data structure for the rest of the front-end
- Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing & building ASTs

*btw, if you ever see someone say *syntactic analysis*, they mean (scanning &) parsing

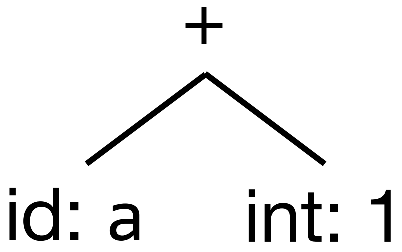
Concrete vs. Abstract Syntax

- The full (concrete) parse tree contains all of the derivation details. The Abstract Syntax Tree (AST) omits information that is necessary to parse the input, but not needed for later processing
- e.g.

Concrete Syntax



Abstract Syntax



Context Free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CFG)
- CFGs are more expressive than regular expressions but less so than general grammars — in a sweet spot
    - powerful enough to describe nesting and recursion
    - **but** unlike general grammars, CFG-membership is ***decidable***. (very efficiently with minor restrictions)
- Not perfect
    - Cannot capture every constraint we want to impose to define ***valid*** programs, such as typing
    - Can be ***ambiguous***

Derivations & Parse Trees

- **Derivation** — (generation) a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals (a.k.a. tokens)
- **Parsing** — (recognition + output) the inverse to the process of derivation
  - Starting with a sequence of terminals, we want to recover (discover, really) the non-terminals and structure, i.e. the parse tree (a.k.a. concrete syntax tree).

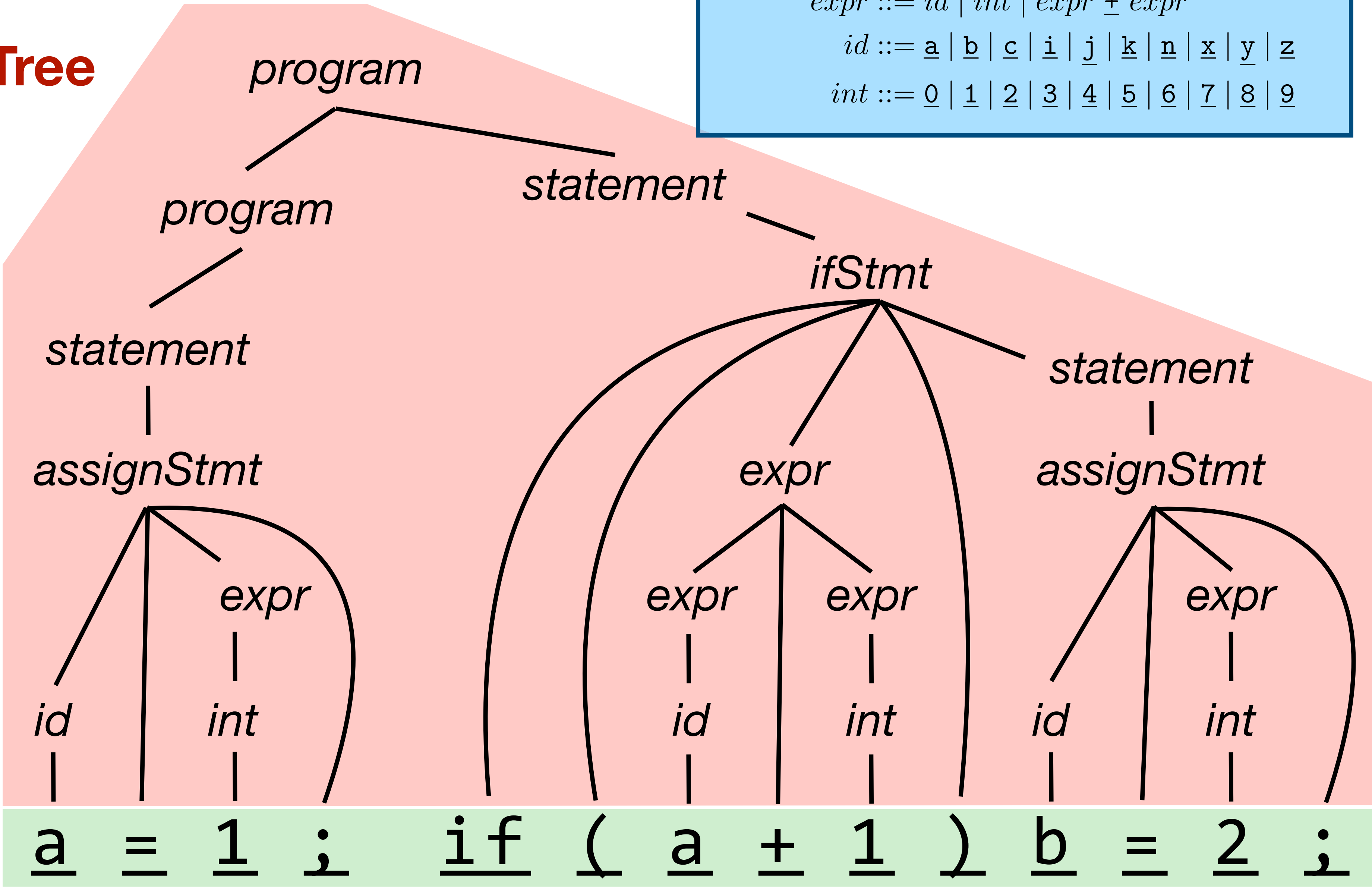
Old Example

Grammar

$$\begin{aligned}
\mathit{program} &::= \mathit{statement} \mid \mathit{program}\ \mathit{statement} \\
\mathit{statement} &::= \mathit{assignStmt} \mid \mathit{ifStmt} \\
\mathit{assignStmt} &::= \mathit{id}\ \underline{=}\ \mathit{expr}\ \underline{;} \\
\mathit{ifStmt} &::= \underline{\texttt{if}}\ \underline{(}\ \mathit{expr}\ \underline{)}\ \mathit{statement} \\
\mathit{expr} &::= \mathit{id} \mid \mathit{int} \mid \mathit{expr}\ \underline{+}\ \mathit{expr} \\
\mathit{id} &::= \underline{\texttt{a}} \mid \underline{\texttt{b}} \mid \underline{\texttt{c}} \mid \underline{\texttt{i}} \mid \underline{\texttt{j}} \mid \underline{\texttt{k}} \mid \underline{\texttt{n}} \mid \underline{\texttt{x}} \mid \underline{\texttt{y}} \mid \underline{\texttt{z}} \\
\mathit{int} &::= \underline{0} \mid \underline{1} \mid \underline{2} \mid \underline{3} \mid \underline{4} \mid \underline{5} \mid \underline{6} \mid \underline{7} \mid \underline{8} \mid \underline{9}
\end{aligned}$$

Parse Tree

String of Tokens



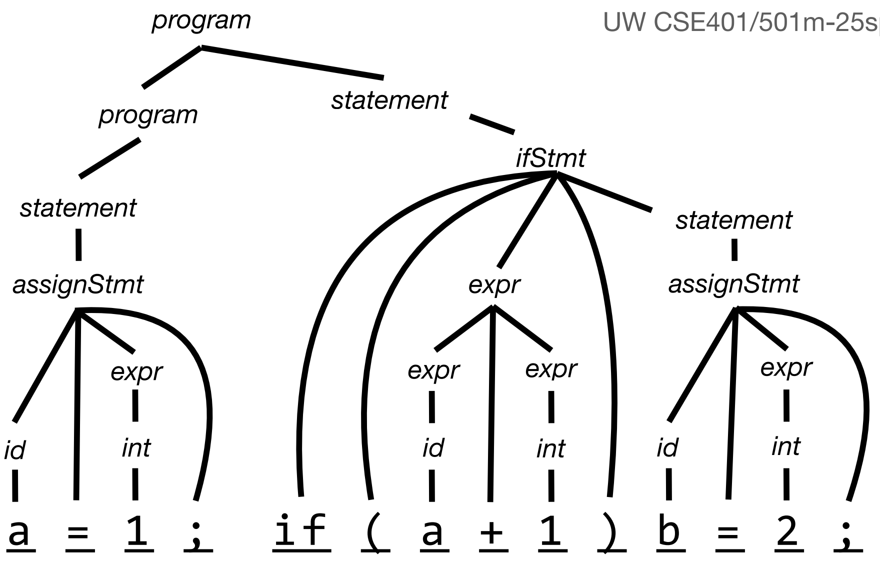
The Parsing Problem

- (Input) Let G be a grammar
- (Input) Let w be a sentence (i.e. string of tokens)
- (Output) Then
  - ✦ Decide whether or not $w \in L(G)$
  - ✦ If so, traverse the parse tree of w ***in some standard order*** and ***do something useful*** at each node
    - The tree might not be produced explicitly, but the control flow of the parser will correspond to a traversal.

Standard Order

- For practical reasons we want the parser to be ***deterministic*** (no backtracking), and we want to examine the source program from ***left to right***.
  - i.e. parse the program in linear time in the order it appears in the source file

Common Orders



- Top-down
  - ✦ Start with the root
  - ✦ Traverse the parse tree *depth-first,* and *left-to-right* (aka. a **left-most derivation**)
  - ✦ LL(k), recursive-descent
- Bottom-up
  - ✦ Start at leaves and build up to the root
    - Effectively a **right-most derivation** in reverse(!)
  - ✦ LR(k) and subsets (LALR(k), SLR(k), etc.)

Something Useful

- At each point (node) in the traversal, perform some semantic action, e.g.
  - Construct nodes of full parse tree (rare)
  - Construct abstract syntax tree (AST) (common)
  - Construct linear, lower-level representation (usually done in a second ***pass*** after constructing AST)
  - Generate target code on the fly
    - used to be done in 1-pass compilers
    - (Gilbert's opinion) don't write 1-pass compilers

Outline

Parsing Overview

Context-Free Grammars

Ambiguous Grammars

Context-Free Grammars

- A **grammar** G consists of
  - ✦ N — a finite set of **non-terminal symbols**
  - ✦ Σ — a finite set of **terminal symbols** (aka. alphabet)
  - ✦ P — a finite set of **productions**
    - a production is of the form $\alpha ::= \beta$,
      where $\alpha \in N$ and $\beta \in (N \cup \Sigma)^*$
  - ✦ S — the **start symbol**, a distinguished element of N
    - if not otherwise specified, this is usually assumed to be the left-hand non-terminal α of the first production

Example: CFG

$$\begin{aligned} expr &::= expr + expr \\ &\mid \ expr - expr \\ &\mid \ expr * expr \\ &\mid \ expr \ / \ expr \\ &\mid \ int \\ int &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \\ &\mid \ 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Productions

$$P = \left\{ \begin{array}{l} expr ::= expr + expr, \\ expr ::= expr - expr, \\ expr ::= expr * expr, \\ expr ::= expr \ / \ expr, \\ expr ::= int, int ::= 0, \\ int ::= 1, int ::= 2, int ::= 3, \\ int ::= 4, int ::= 5, int ::= 6, \\ int ::= 7, int ::= 8, int ::= 9, \end{array} \right\}$$

Non-terminal symbols

$$N = \{expr, int\}$$

Terminal symbols

$$\Sigma = \left\{ \begin{array}{l} +, -, *, /, \\ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \end{array} \right\}$$

Start symbol

$$S = expr$$

Meta-Syntax vs. Concrete

$$\begin{array}{rl} expr ::= & expr + expr \\ | & expr - expr \\ | & expr * expr \\ | & expr \,/\, expr \\ | & int \\ int ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \\ | & 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Productions

$$P = \left\{ \begin{array}{l} expr ::= expr + expr, \\ expr ::= expr - expr, \\ expr ::= expr * expr, \\ expr ::= expr \,/\, expr, \\ expr ::= int, int ::= 0, \\ int ::= 1, int ::= 2, int ::= 3, \\ int ::= 4, int ::= 5, int ::= 6, \\ int ::= 7, int ::= 8, int ::= 9, \end{array} \right\}$$

Non-terminal symbols

$$N = \{expr, int\}$$

Start symbol

$$S = expr$$

Terminal symbols

$$\Sigma = \left\{ \begin{array}{l} +, -, *, /, \\ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \end{array} \right\}$$

The Derivation Relation

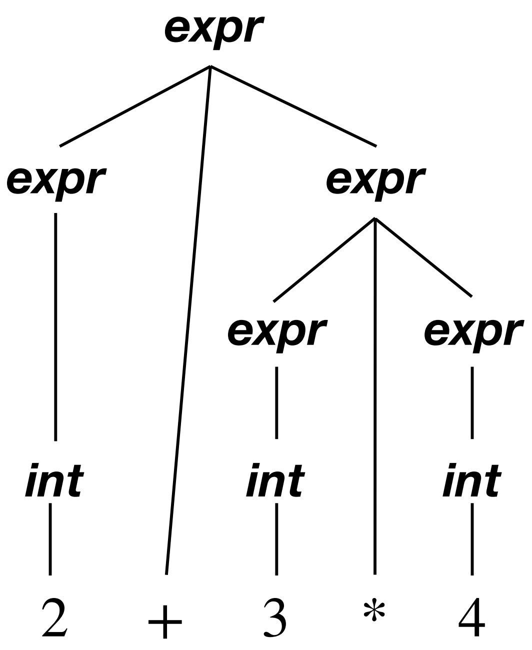
- One step-derivation
  - Let $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, and $\delta \in N$
  - If $\delta ::= \beta$ is a production, then $\alpha\delta\gamma \Rightarrow \alpha\beta\gamma$, which we read as “from $\alpha\delta\gamma$ we **derive** $\alpha\beta\gamma$“
- We say that a sequence of “sentences” $\alpha_0, \alpha_1, \ldots, \alpha_n \in (N \cup \Sigma)^*$ is a **derivation** when $a_0 \Rightarrow a_1 \Rightarrow \cdots \Rightarrow a_n$
  - We write $\alpha \Rightarrow^* \beta$ when there is a derivation of β from α
- ***Useful***: if $w \in \Sigma^*$ is a sentence of only *terminal symbols*, then w is “terminal” (i.e. there is no α such that $w \Rightarrow \alpha$)

Note: $\Rightarrow^*$ is the transitive, reflexive closure of $\Rightarrow$ but it’s not necessary to know this

Example: Derivation of $2 + 3 * 4$

sentence	**production used**
$expr$	
$\boxed{expr + expr}$	$expr ::= expr + expr$
$\boxed{int} + expr$	$expr ::= int$
$\boxed{2} + expr$	$int ::= 2$
$2 + \boxed{expr * expr}$	$expr ::= expr * expr$
$2 + \boxed{3} * expr$	$expr ::= int,\ int ::= 3$
$2 + 3 * \boxed{4}$	$expr ::= int,\ int ::= 4$

parse tree



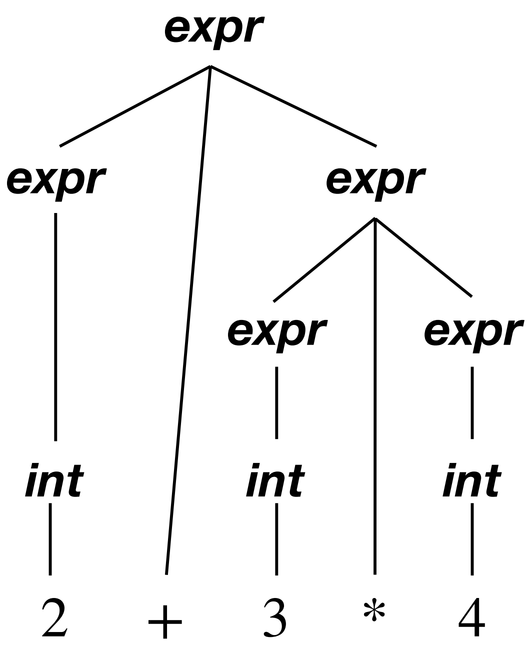
Left- and Right-Most Derivations

- The preceding derivation relation doesn't impose ***order***
- Let $\alpha, \beta, \gamma \in (N \cup \Sigma)*$ be sentences of any symbols, but let $w \in \Sigma*$ be a sentence of only terminal symbols
- If $\delta ::= \beta$, then $w\delta\gamma \Rightarrow_{lm} w\beta\gamma$ (***derives leftmost***)
- If $\delta ::= \beta$, then $\alpha\delta w \Rightarrow_{rm} \alpha\beta w$ (***derives rightmost***)
- We will only be interested in left-most and right-most derivations, not arbitrary orderings

Example: Rightmost Derivation

sentence	**production used**
$expr$	
$\boxed{expr + expr}$	$expr ::= expr + expr$
$expr + \boxed{expr * expr}$	$expr ::= expr * expr$
$expr + expr * \boxed{4}$	$expr ::= int,\ int ::= 4$
$expr + \boxed{3} * 4$	$expr ::= int,\ int ::= 3$
$\boxed{2} + 3 * 4$	$expr ::= int,\ int ::= 2$

parse tree



Example: L vs. R (one slide)

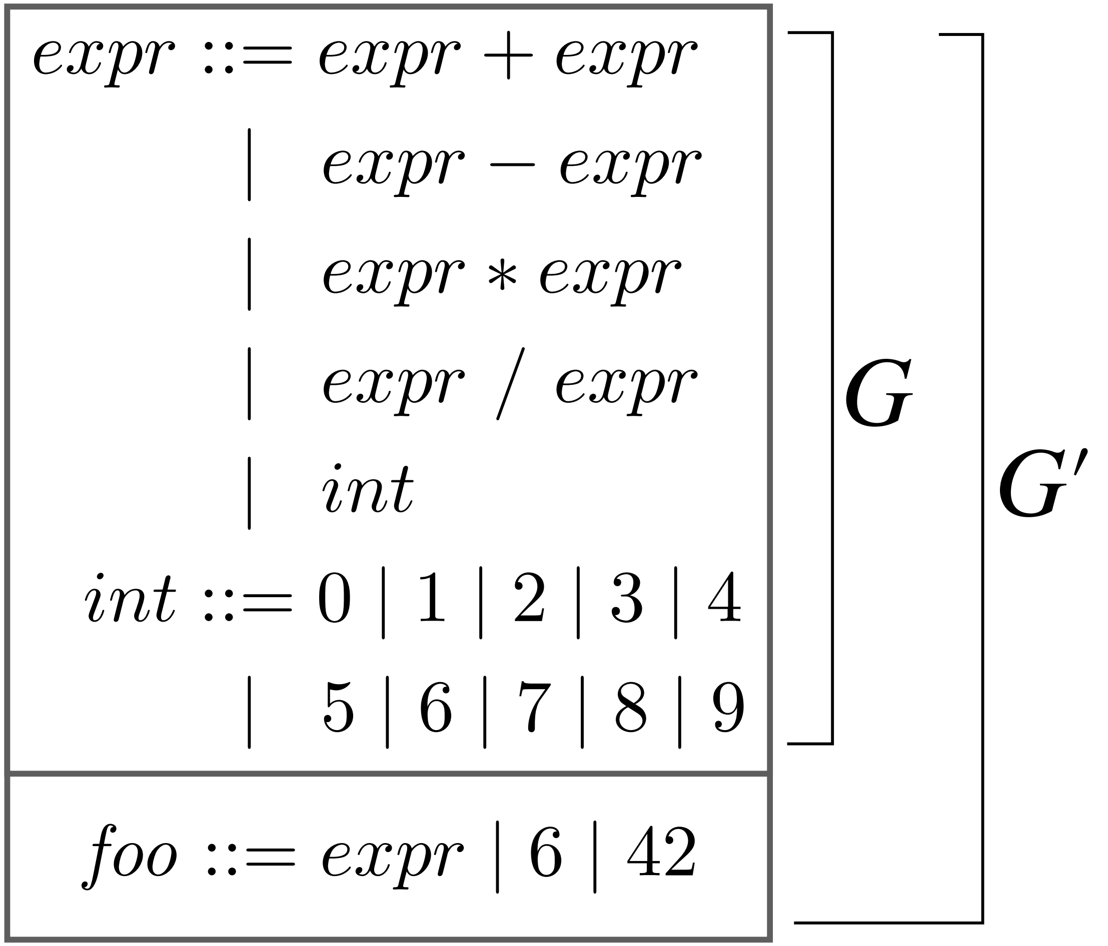
Leftmost	**Rightmost**
expr	expr
$\boxed{\mathit{expr} + \mathit{expr}}$	$\boxed{\mathit{expr} + \mathit{expr}}$
$\boxed{\mathit{int}} + \mathit{expr}$	$\mathit{expr} + \boxed{\mathit{expr} * \mathit{expr}}$
$\boxed{2} + \mathit{expr}$	$\mathit{expr} + \mathit{expr} * \boxed{4}$
$\boxed{2 +}\boxed{\mathit{expr} * \mathit{expr}}$	$\mathit{expr} + \boxed{3}\boxed{* 4}$
$\boxed{2 +}\boxed{3} * \mathit{expr}$	$\boxed{2}\boxed{+ 3 * 4}$
$\boxed{2 + 3 *}\boxed{4}$	

Observe: Everything to the left/right of the derivation step is a terminal

From Grammars to Languages

- Let $G = (N, \Sigma, P, S)$ be a grammar
- For every nonterminal $A \in N$, define the **language** associated with that non-terminal to be $L(A) = \{w \in \Sigma^* \mid A \Rightarrow^* w\}$
  - i.e. the language of all (terminal) sentences that derive from A
- Since S is the initial symbol of G, define $L(G) = L(S)$
  - Again, the non-terminal on the left of the first production rule is taken to be the start symbol if no other start symbol is specified.

Example: Useless Productions



$$
\begin{array}{rcl}
expr & ::= & expr + expr \\
 & | & expr - expr \\
 & | & expr * expr \\
 & | & expr \ / \ expr \\
 & | & int \\
int & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \\
 & | & 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

$$foo ::= expr \mid 6 \mid 42$$

Question 1 — Is $L(G) = L(G')$?

Question 2 — Does the non-terminal foo occur in any derivation using grammar G' ?

Question 3 — Why would we ever write a grammar like G' ?

Reduced Grammars

- A grammar G is **reduced** iff. every production $\alpha ::= \beta$ in G is used in some derivation

  $$S \Rightarrow^* \alpha\alpha\gamma \Rightarrow \alpha\beta\gamma \Rightarrow^* w$$

  - in other words, every production is useful
- Convention: we will only use reduced grammars
  - There are algorithms for pruning useless productions from grammars — see a formal language or compiler book for details

Outline

Parsing Overview

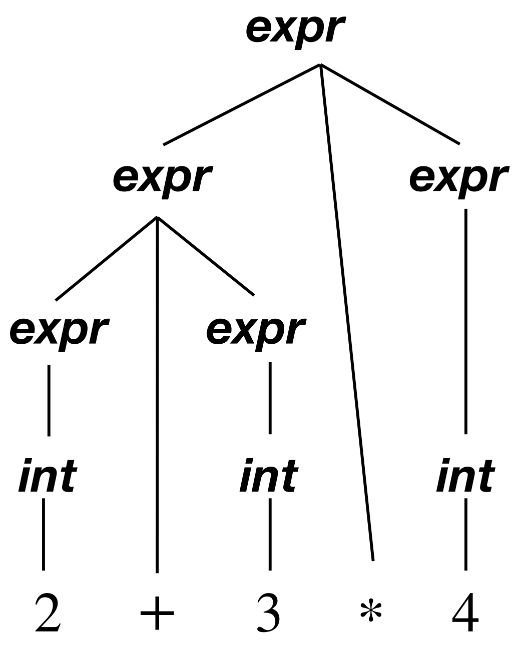
Context-Free Grammars

Ambiguous Grammars

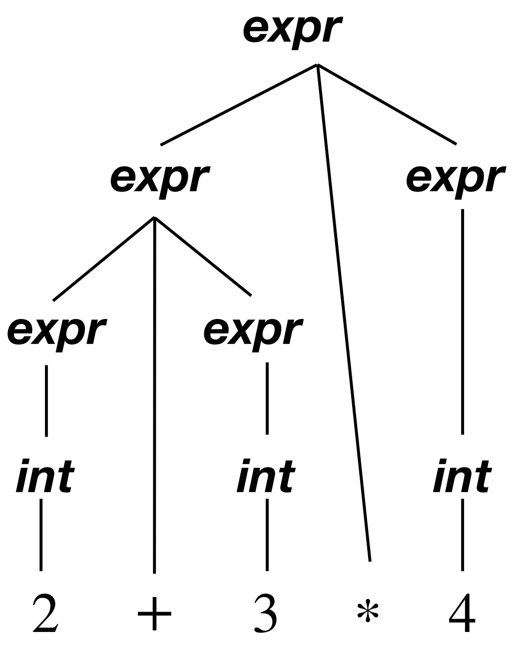
Another Derivation of $2 + 3 * 4$

sentence	**production used**
$expr$	
$\boxed{expr * expr}$	$expr ::= expr * expr$
$\boxed{expr + expr} * expr$	$expr ::= expr + expr$
$\boxed{2} + expr * expr$	$expr ::= int,\ int ::= 2$
$2 + \boxed{3} * expr$	$expr ::= int,\ int ::= 3$
$2 + 3 * \boxed{4}$	$expr ::= int,\ int ::= 4$

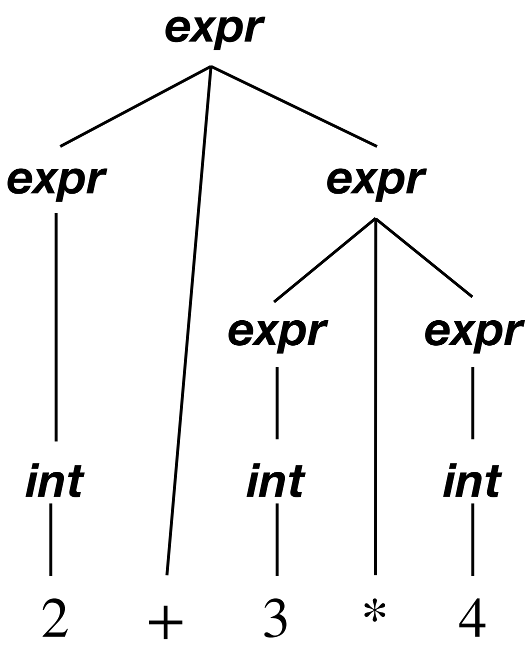
parse tree



Two Derivations of 2 + 3 * 4



“(2 + 3) * 4”

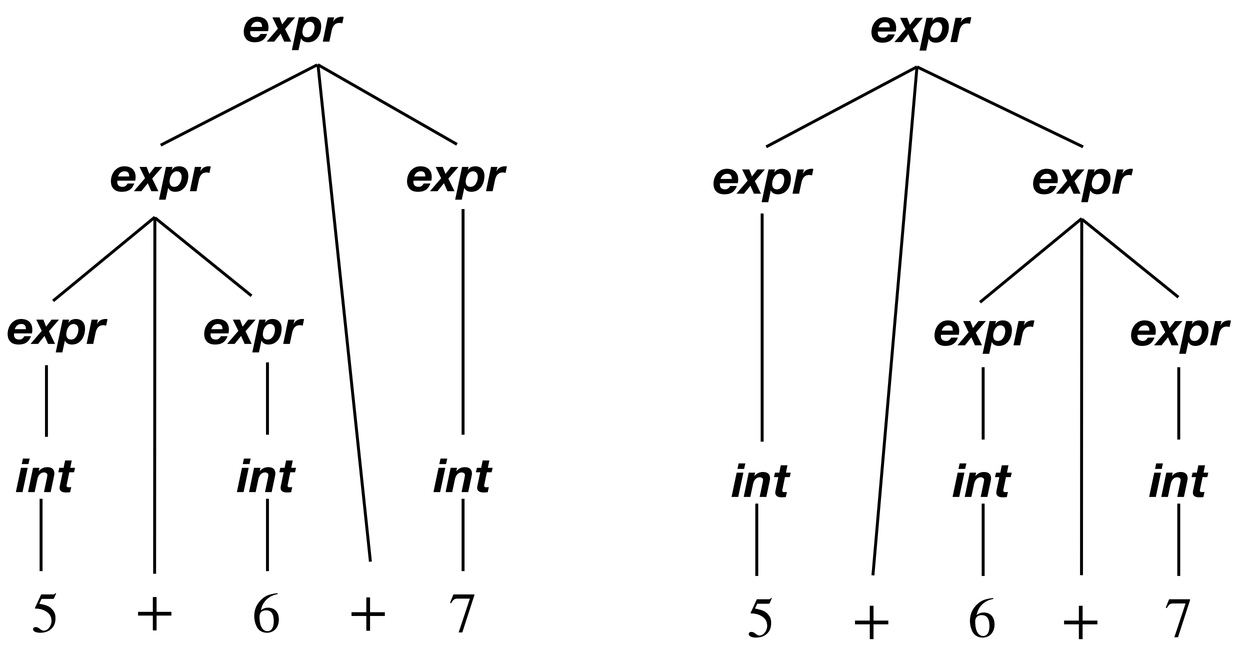


“2 + (3 * 4)”

(Un)Ambiguous Grammars

- A Grammar G is **unambiguous** iff every $w \in L(G)$ has a unique left-most (or right-most) derivation
    - ✦ (theorem) a sentence $w \in L(G)$ has a unique left-most derivation iff. it has a unique right-most derivation
- A grammar without this property is **ambiguous**
    - ✦ But other grammars that generate the same language might be unambiguous — ambiguity is a property of grammars, not languages
- We need unambiguous grammars in order to ensure that parsing is a deterministic process

Two Derivations of $5 + 6 + 7$



What’s Going on Here?

- The grammar has no notion of precedence
  - e.g. interpreting as 2 + (3 * 4) vs. (2 + 3) * 4
- The grammar has no notion of associativity
  - e.g. interpreting as 5 + (6 + 7) vs. (5 + 6) + 7
- Traditional solution
  - Create a non-terminal for each level of precedence
  - Force the parser to recognize higher-precedence subexpressions first
  - Use left or right recursion in the grammar for left or right associativity of operators

Classic Expression Grammar

(first used in ALGOL 60)

$$expr ::= expr + term \mid expr - term \mid term$$

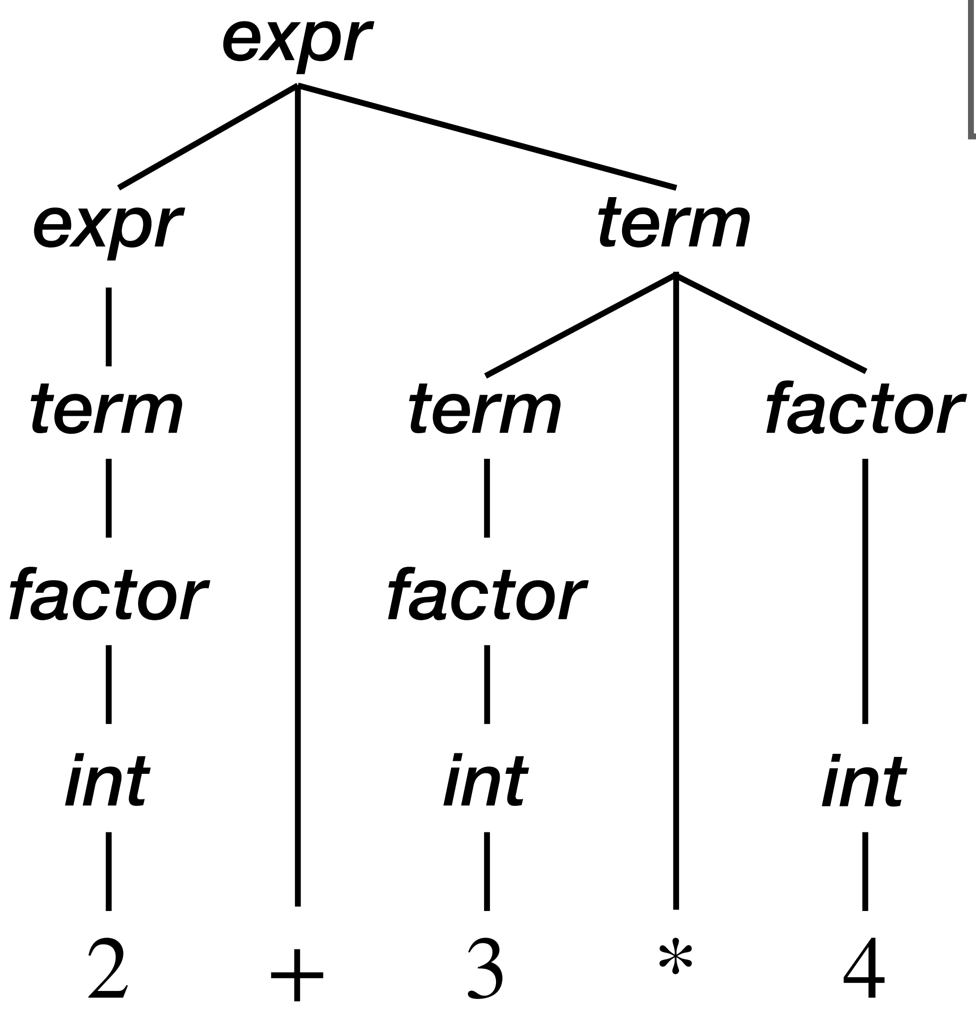
$$term ::= term * factor \mid term \ / \ factor \mid factor$$

$$factor ::= int \mid (expr)$$

$$int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Check: Derive $2 + 3 * 4$

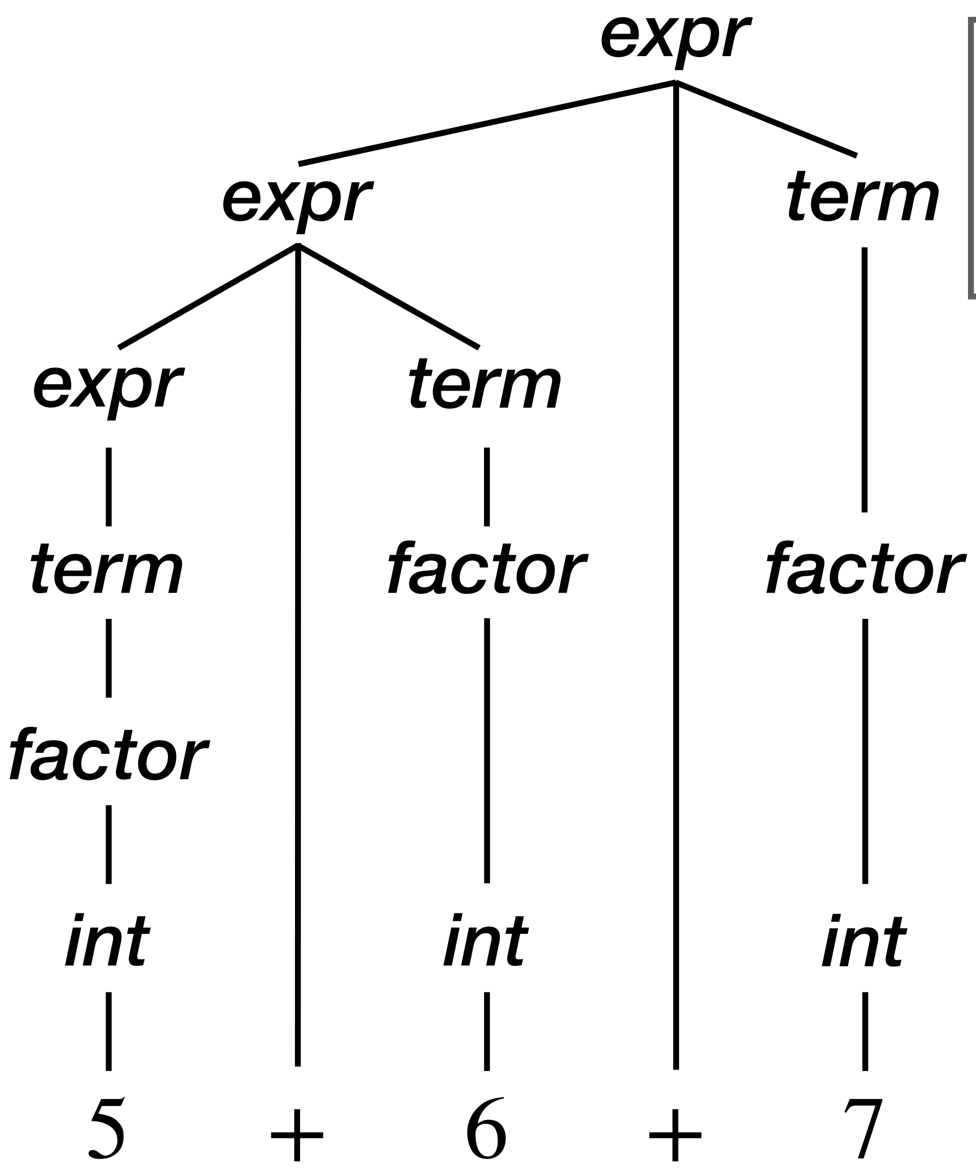
$$\begin{aligned} expr &::= expr + term \mid expr - term \mid term \\ term &::= term * factor \mid term\ /\ factor \mid factor \\ factor &::= int \mid (expr) \\ int &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$



Key observation:

The separation of non-terminals (expr/term/factor as opposed to expr alone) enforces precedence

Check: Derive $5 + 6 + 7$



$$
\begin{aligned}
expr &::= expr + term \mid expr - term \mid term \\
term &::= term * factor \mid term \ / \ factor \mid factor \\
factor &::= int \mid (expr) \\
int &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Key observation:

The choice of whether the rules for expr/term/factor are left vs. right-recursive (left-recursive here) controls the associativity

Check: Derive $5 + (6 + 7)$

$$\begin{aligned} expr &::= expr + term \mid expr - term \mid term \\ term &::= term * factor \mid term \; / \; factor \mid factor \\ factor &::= int \mid (expr) \\ int &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

(left as an exercise 🙂)

Another Classic Example

- Grammar for conditional statements

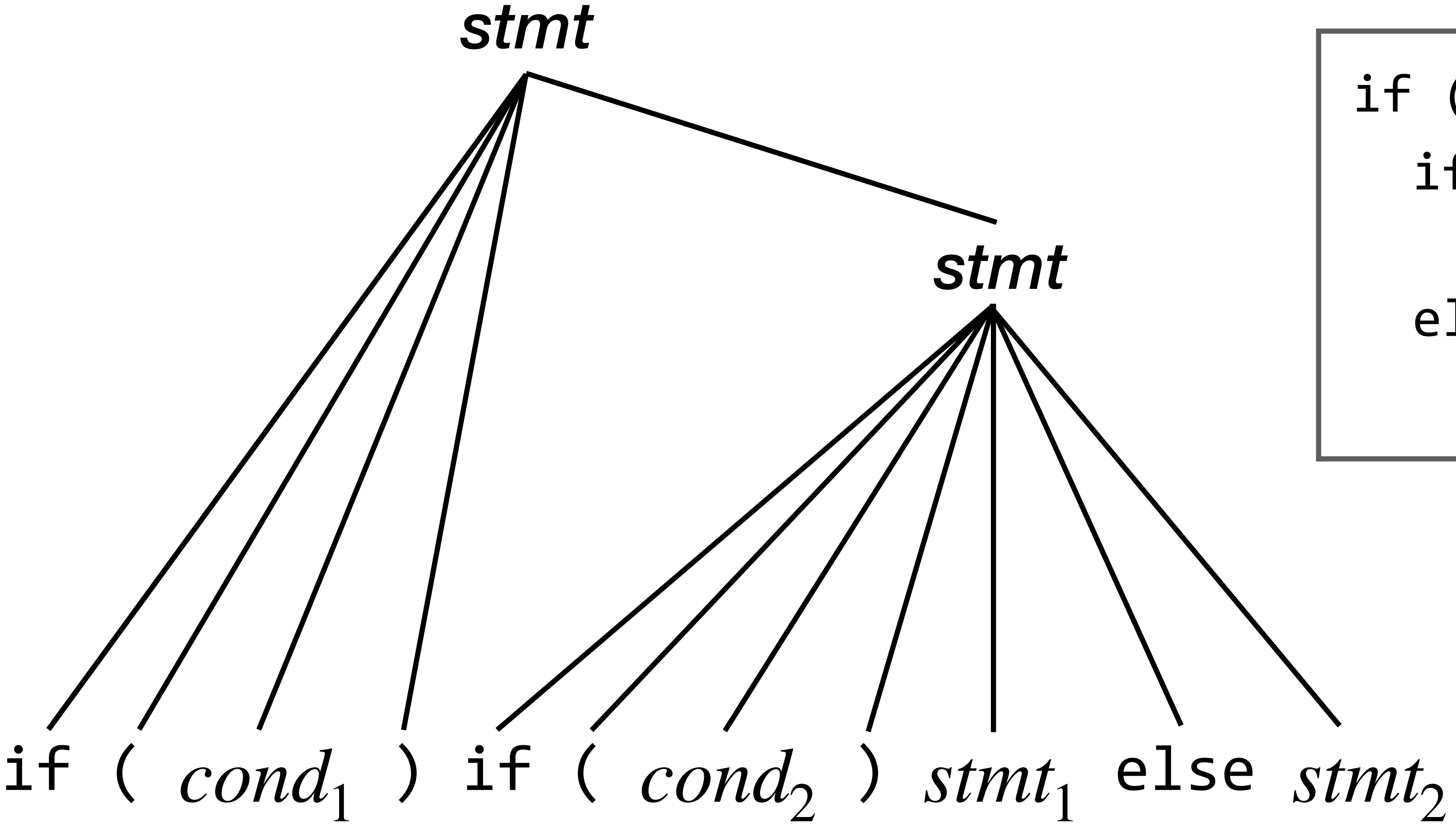
$$\begin{aligned} stmt ::= &\ \texttt{if (}\ cond\ \texttt{)}\ stmt \\ \mid &\ \texttt{if (}\ cond\ \texttt{)}\ stmt\ \texttt{else}\ stmt \end{aligned}$$

  - (this is the *dangling else* problem found in many, many grammars for languages, beginning with ALGOL 60)
- Exercise: show that this is ambiguous
  - How do we do this?

One Derivation

$$\begin{aligned} stmt ::= \; & \texttt{if} \; \texttt{(} \; cond \; \texttt{)} \; stmt \\ \mid \; & \texttt{if} \; \texttt{(} \; cond \; \texttt{)} \; stmt \; \texttt{else} \; stmt \end{aligned}$$

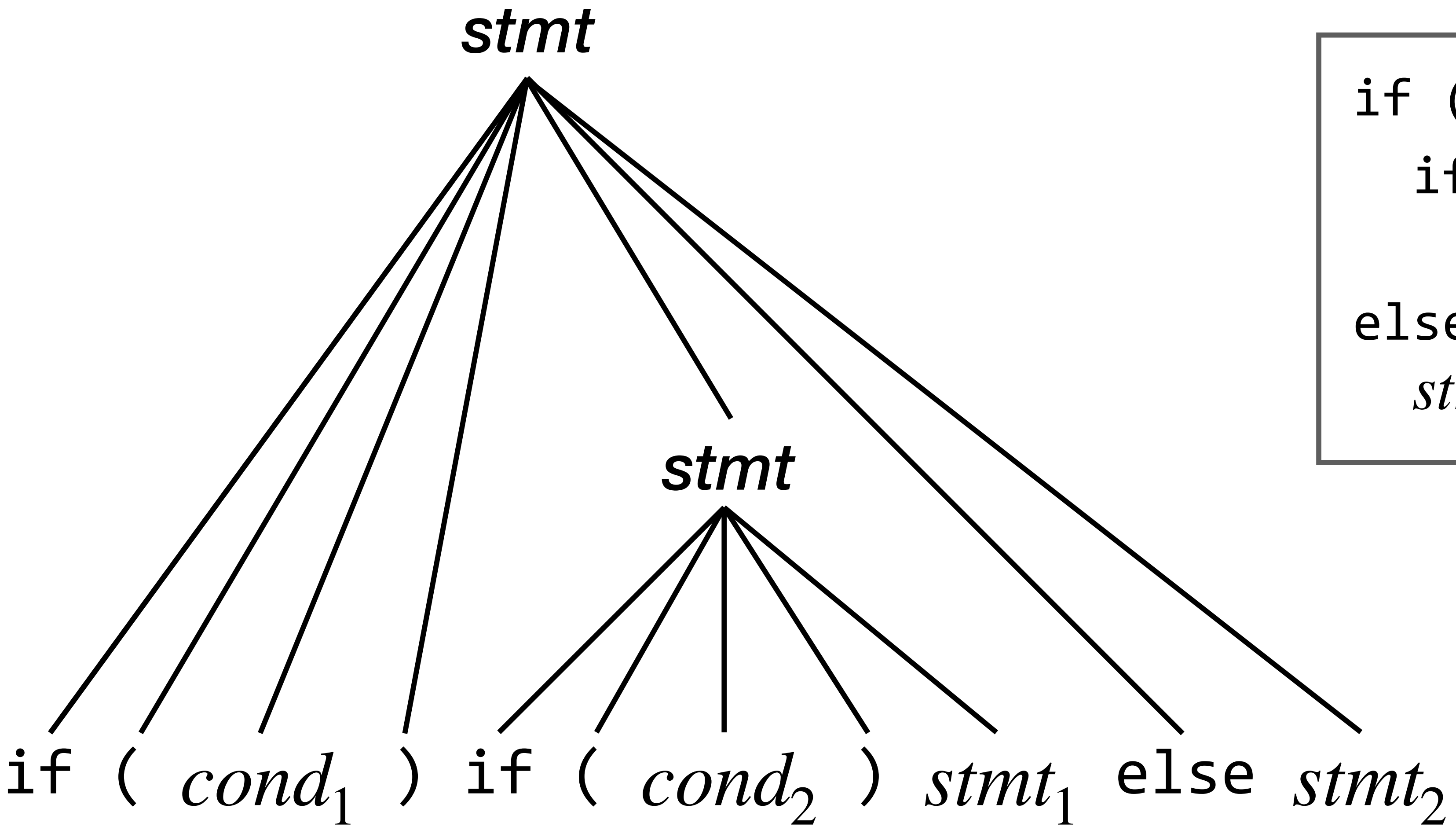
`if` ($cond_1$)
  `if` ($cond_2$)
    $stmt_1$
  `else`
    $stmt_2$



Another Derivation

$stmt ::=$ `if` `(` $cond$ `)` $stmt$

$\quad\quad\mid$ `if` `(` $cond$ `)` $stmt$ `else` $stmt$



`if` `(`$cond_1$`)`

  `if` `(`$cond_2$`)`

    $stmt_1$

`else`

  $stmt_2$

Removing the “if” Ambiguity

- Fix the grammar so that the “if-then” and “if-then-else” productions proceed from distinct non-terminals
  - Similar to precedence fix for operators, but more subtle
  - This is done in the Java reference grammar; downside: results in more non-terminals
- OR change the language (if you’re designing it)
  - e.g. require a delimiter — `if (cond) stmt end`
- OR use some ad-hoc rule in the parser (not great)
  - “else matches closest unpaired if”

Use Grammatical Precedence (1)

Original, Ambiguous Grammar

$$\begin{aligned} stmt ::= & \ \ldots \\ & \mid \ \texttt{if} \ (\ cond\ )\ stmt \\ & \mid \ \texttt{if} \ (\ cond\ )\ stmt\ \texttt{else}\ stmt \end{aligned}$$

another `else` is not allowed in these positions

Modified, Unambiguous Grammar

$$\begin{aligned} stmt ::= & \ other\_stmt \\ & \mid \ \texttt{if} \ (\ cond\ )\ stmt \\ & \mid \ \texttt{if} \ (\ cond\ )\ with\_else\ \texttt{else}\ stmt \\ stmt ::= & \ other\_stmt \\ & \mid \ \texttt{if} \ (\ cond\ )\ with\_else\ \texttt{else}\ with\_else \\ other\_stmt ::= & \ \ldots \end{aligned}$$

Check

$$\begin{aligned} stmt &::= other\_stmt \\ &\mid\ \texttt{if}\ (\ cond\ )\ stmt \\ &\mid\ \texttt{if}\ (\ cond\ )\ with\_else\ \texttt{else}\ stmt \\ stmt &::= other\_stmt \\ &\mid\ \texttt{if}\ (\ cond\ )\ with\_else\ \texttt{else}\ with\_else \\ other\_stmt &::= \ldots \end{aligned}$$

stmt

(exercise 🙂)

$$\texttt{if}\ (\ cond_1\ )\ \texttt{if}\ (\ cond_2\ )\ stmt_1\ \texttt{else}\ stmt_2$$

Change Language Design (2)

- If you can (re-)design the language, you can just avoid this problem entirely

$$
\begin{aligned}
stmt ::= {} & \ldots \\
\mid {} & \texttt{if (} cond \texttt{) } stmt \texttt{ end} \\
\mid {} & \texttt{if (} cond \texttt{) } stmt \texttt{ else } stmt \texttt{ end}
\end{aligned}
$$

- Pros: unambiguous formally, and for mere humans
- Alternate: In Rust and Swift, $\{stmt\}$ braces are required
- Cons: requires programmers to type additional syntax
  - Debatable whether it’s better in this case

Parser Tools: *Operators*

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler — *if* used with discipline
- Usually can specify precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for parser — let the tool handle the details (but only when it makes sense)
    - (i.e. expr ::= expr + expr | expr * expr | … with assoc. & precedence declarations is often the best solution)
- Take advantage of this to simplify the grammar when using parser-generator tools
  - We *will* do this in our compiler project

Parser Tools: *Ambiguity*

- Possible rules for resolving other problems
  - Earlier productions in the grammar preferred to later ones (danger here if parser input changes)
  - Longest match used if there is a choice (good solution for dangling ifs and a few similar things)
- Parser tools normally allow for this
  - BUT is it really the behavior you want?
  - Now your language's behavior depends on arbitrary choices in the specification of your parser generator tool… (what happens if tool behavior changes?)

Next Time…

- LR Parsing
  - Continue reading Chapter 3 (3.1-3.2 if not already)
  - It’s OK to SKIP top-down parsing (3.3) for now and go immediately to LR/bottom-up parsing (3.4)
- Sections on Thursday
  - Most important section to attend!