

# CSE 401/M501 – Compilers

Parsing & Context-Free Grammars

Hal Perkins

Autumn 2024

# Administrivia

- Reminders:
  - **Project partner signup**. Please fill out the form with your and your partner's names and cse netids
    - One form per group, only; +1 point for the group if it's right
    - Please finish by tomorrow, 11:59 pm
    - Who's still looking for a partner? 401? M501?
      - Mingle at end of class? Postings on ed? or ...?
  - **hw1** due Thur. night (regexps, etc.) via gradescope
    - \* vs \*: Avoid messy `\e\s\c\a\p\e\s` –use something simple like \* (underlined terminal) vs \* (operator). Add a short explanation (sentence or 2) to help grader with notation.
    - (Re-)read the notes at the top of the hw when you think you're "done" 😊

# Administrivia (added Wed.)

- HW1 due tomorrow night
- Scanner assignment, first part of the project, and general project overview and documents, posted now; scanner due a week from Thursday
  - Details, demos, tools, etc. in sections tomorrow – BE THERE!!
  - Follow the readme instructions in the repo to set up IntelliJ if you are using that (clone the repo first without IntelliJ and read the instructions before starting the IDE)
  - Will set up gitlab repos with starter code for as many groups as possible later today – watch for post on ed when done
    - Will also contact people who do not have partners in the next couple of days to figure out a plan – but enter last-minute info now!

# Agenda for Today

- Parsing overview
- Context free grammars
- Ambiguous grammars
- Reading: Cooper & Torczon 3.1-3.2
  - Dragon book (Aho et al) is also particularly strong on grammars and languages

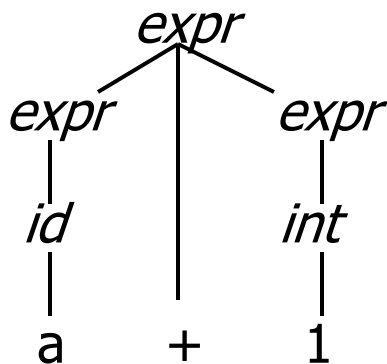
# Syntactic Analysis / Parsing

- Goal: Convert token stream to an **abstract syntax tree**
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for next phases of compilation
- Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing and building ASTs

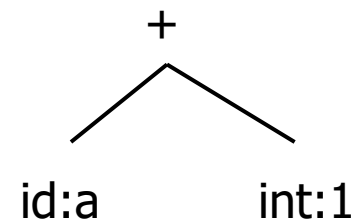
# Concrete vs Abstract Syntax

- The full parse tree includes 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
- Example:

Concrete Syntax



Abstract Syntax



# Context-free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CFG)
- Compromise between
  - REs: can't nest or specify recursive structure
  - General grammars: more power than needed, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; but also some restrictions for speed
- Not perfect
  - Cannot capture semantics, like “must declare every variable” or “must be **int**”. Requires later semantic pass
  - Can be ambiguous

# Derivations and Parse Trees

- Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals
- Parsing: inverse of derivation
  - Given a sequence of terminals (aka tokens) want to recover (discover) the nonterminals and structure, i.e., the parse (concrete syntax) tree

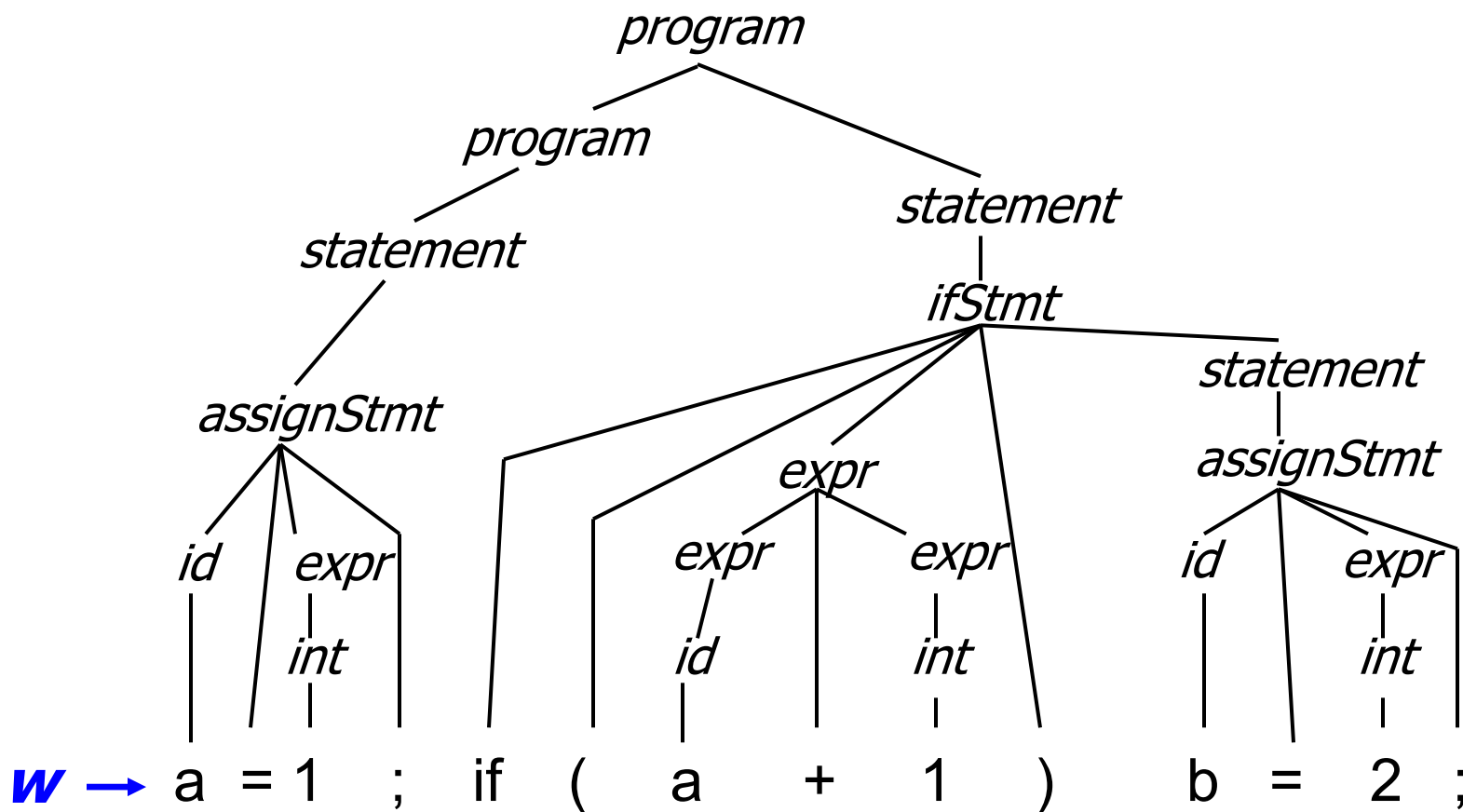


# Old Example

*G*

```

program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) statement
expr ::= id | int | expr + expr
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    
```



# Parsing

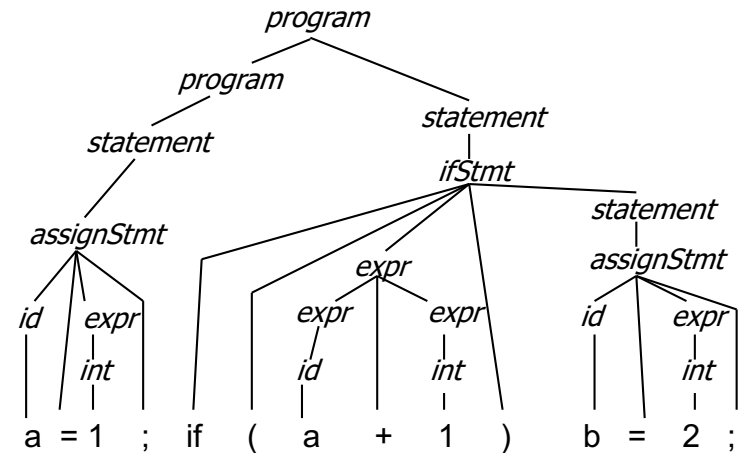
- Parsing: Given a grammar  $G$  and a sentence  $w$  in  $L(G)$ , traverse the derivation (parse tree) for  $w$  in some *standard order* and do *something useful* at each node
  - The tree might not be produced explicitly, but the control flow of the parser will correspond to a traversal

# “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 Orderings

- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  - LL(k), recursive-descent
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost 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
  - Construct nodes of full parse tree (rare)
  - Construct abstract syntax tree (AST) (common)
  - Construct linear, lower-level representation (often produced by traversing initial AST in later phases of production compilers)
  - Generate target code on the fly (done in 1-pass compilers; not common in production compilers)
    - Can’t generate great code in one pass, but can be useful and good-enough if you need a quick ‘n dirty working compiler

# Context-Free Grammars

- Formally, a *grammar*  $G$  is a tuple  $\langle N, \Sigma, P, S \rangle$  where
  - $N$  is a finite set of *non-terminal* symbols
  - $\Sigma$  is a finite set of *terminal* symbols (alphabet)
  - $P$  is a finite set of *productions*
    - A subset of  $N \times (N \cup \Sigma)^*$
  - $S$  is the *start symbol*, a distinguished element of  $N$ 
    - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production

# Standard Notations

$a, b, c$  elements of  $\Sigma$

$w, x, y, z$  elements of  $\Sigma^*$

$A, B, C$  elements of  $N$

$X, Y, Z$  elements of  $N \cup \Sigma$

$\alpha, \beta, \gamma$  elements of  $(N \cup \Sigma)^*$

$A \rightarrow \alpha$  or  $A ::= \alpha$  if  $\langle A, \alpha \rangle$  in  $P$

# Derivation Relations (1)

- $\alpha A \gamma \Rightarrow \alpha \beta \gamma$  iff  $A ::= \beta$  in  $P$ 
  - derives
- $A \Rightarrow^* \alpha$  if there is a chain of productions starting with  $A$  that generates  $\alpha$ 
  - transitive closure



## Derivation Relations (2)

- $w A \gamma \Rightarrow_{lm} w \beta \gamma$  iff  $A ::= \beta$  in  $P$ 
  - derives **leftmost**
- $\alpha A w \Rightarrow_{rm} \alpha \beta w$  iff  $A ::= \beta$  in  $P$ 
  - derives **rightmost**
- We will only be interested in leftmost and rightmost derivations – not random orderings

# Languages

- For  $A$  in  $N$ , define  $L(A) = \{ w \mid A \Rightarrow^* w \}$
- If  $S$  is the start symbol of grammar  $G$ , define  $L(G) = L(S)$ 
  - Nonterminal on left of first rule is taken to be the start symbol if one is not specified explicitly

# Reduced Grammars

- Grammar  $G$  is *reduced* iff for every production  $A ::= \alpha$  in  $G$  there is a derivation
$$S \Rightarrow^* x A z \Rightarrow x \alpha z \Rightarrow^* xyz$$
  - i.e., no production is useless
- Convention: we will use only reduced grammars
  - There are algorithms for pruning useless productions from grammars – see a formal language or compiler book for details

# Ambiguity

- Grammar  $G$  is *unambiguous* iff every  $w$  in  $L(G)$  has a unique leftmost (or rightmost) derivation
  - Fact: unique leftmost or unique rightmost implies the other
- 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 for parsing

## Example: Ambiguous Grammar for Arithmetic Expressions

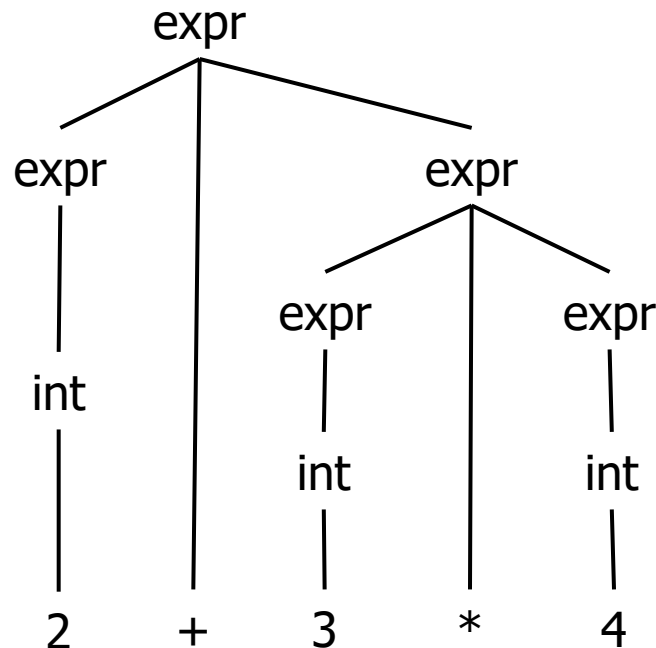
$$\begin{aligned} \text{expr} ::= & \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \\ & \mid \text{expr} * \text{expr} \mid \text{expr} / \text{expr} \mid \text{int} \end{aligned}$$
$$\text{int} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- Exercise: show that this is ambiguous
  - How? Show two different leftmost or rightmost derivations for the same string
  - Equivalently: show two different parse trees for the same string

## Example (cont)

$expr ::= expr + expr \mid expr - expr$   
 $\mid expr * expr \mid expr / expr \mid int$   
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

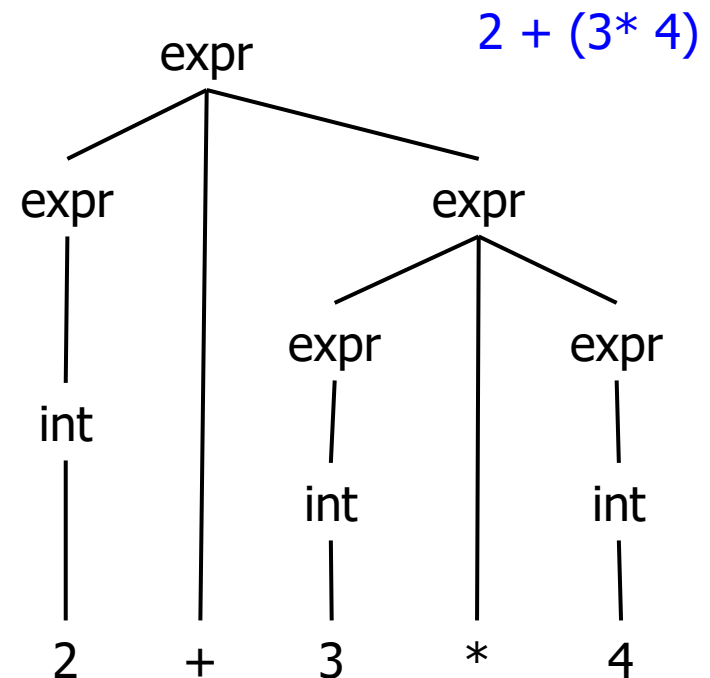
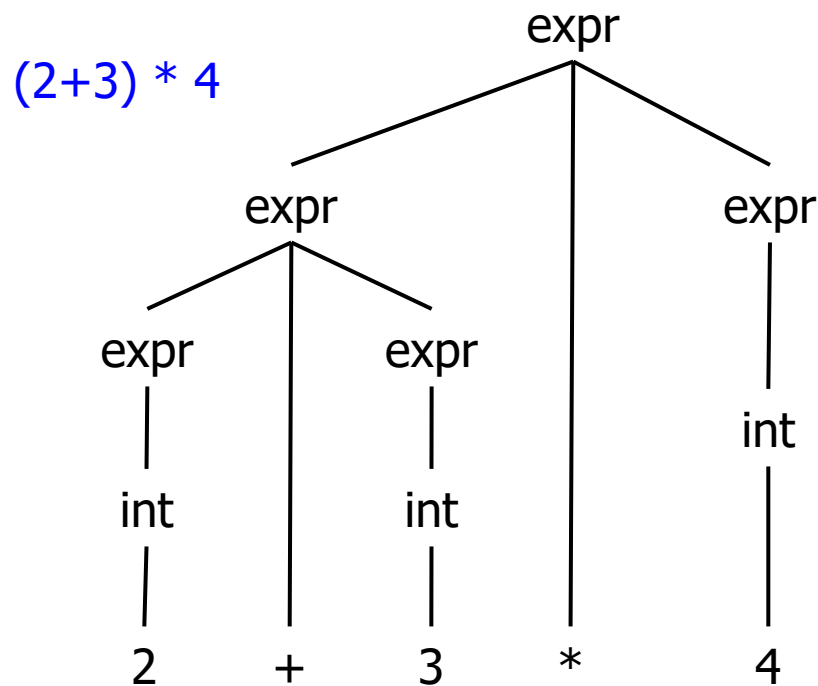
- Give a leftmost derivation of  $2+3*4$  and show the parse tree



## Example (cont)

$expr ::= expr + expr \mid expr - expr$   
 $\mid expr * expr \mid expr / expr \mid int$   
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- Give a different leftmost derivation of  $2+3*4$  and show the parse tree

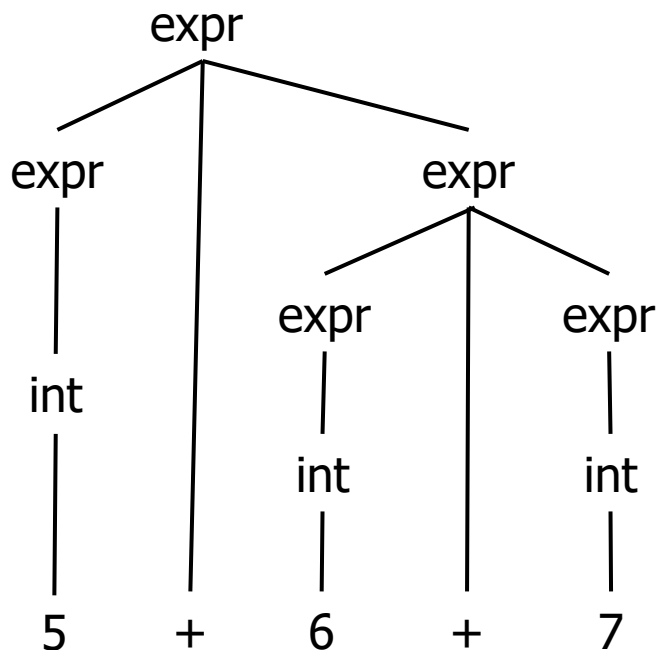


# Another example

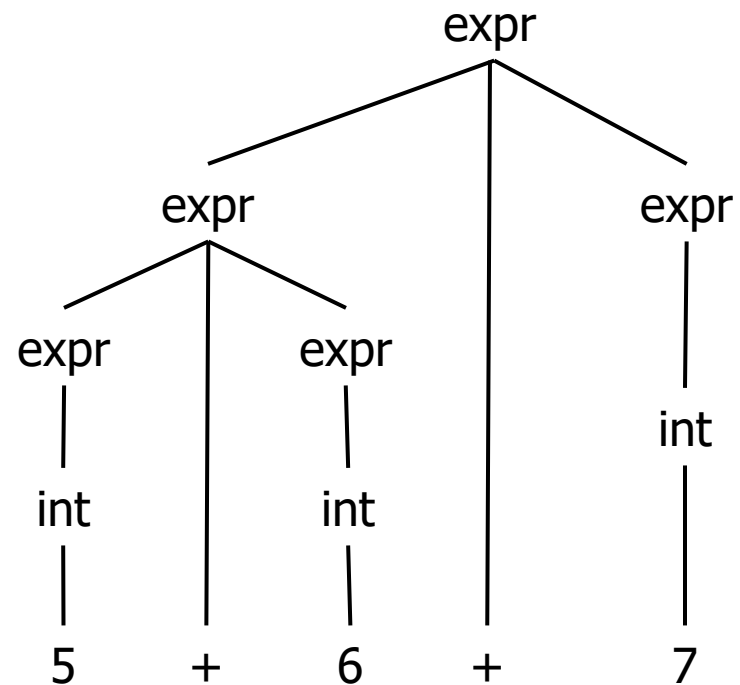
$expr ::= expr + expr \mid expr - expr$   
 $\mid expr * expr \mid expr / expr \mid int$   
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- Give two different derivations of  $5+6+7$

$5 + (6+7)$



$(5+6) + 7$





# What's going on here?

- The grammar has no notion of precedence or associativity
- Traditional solution
  - Create a non-terminal for each level of precedence
  - Isolate the corresponding part of the grammar
  - Force the parser to recognize higher precedence subexpressions first
  - Use left- or right-recursion for left- or right-associative operators

# Classic Expression Grammar

(first used in ALGOL 60)

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

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

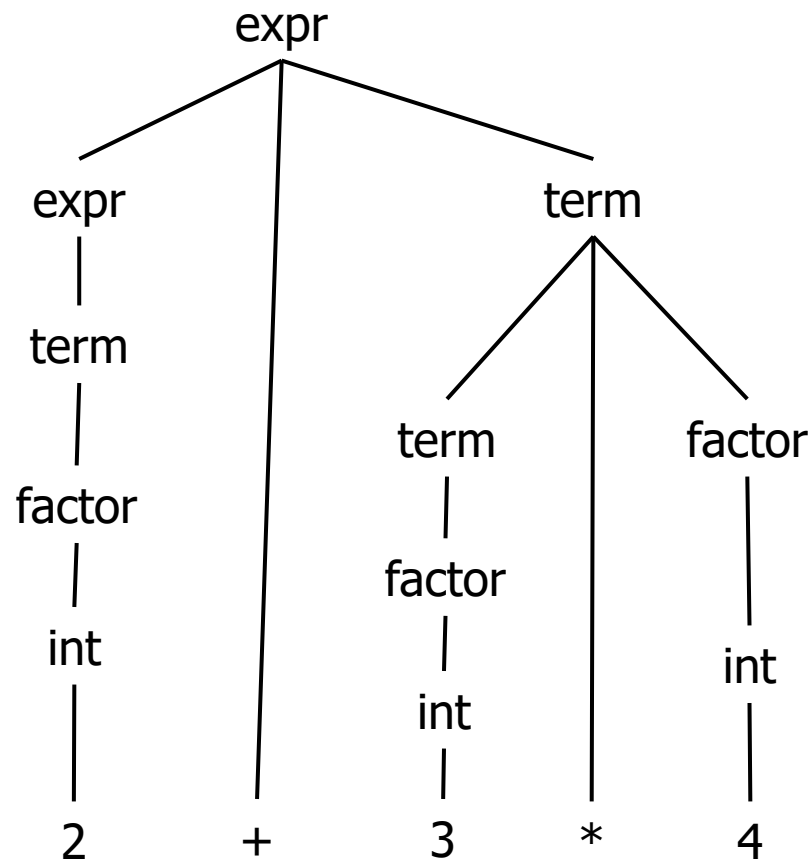
$factor ::= int \mid ( expr )$

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

# Check:

## Derive $2 + 3 * 4$

$expr ::= expr + term \mid expr - term \mid term$   
 $term ::= term * factor \mid term / factor \mid factor$   
 $factor ::= int \mid ( expr )$   
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$

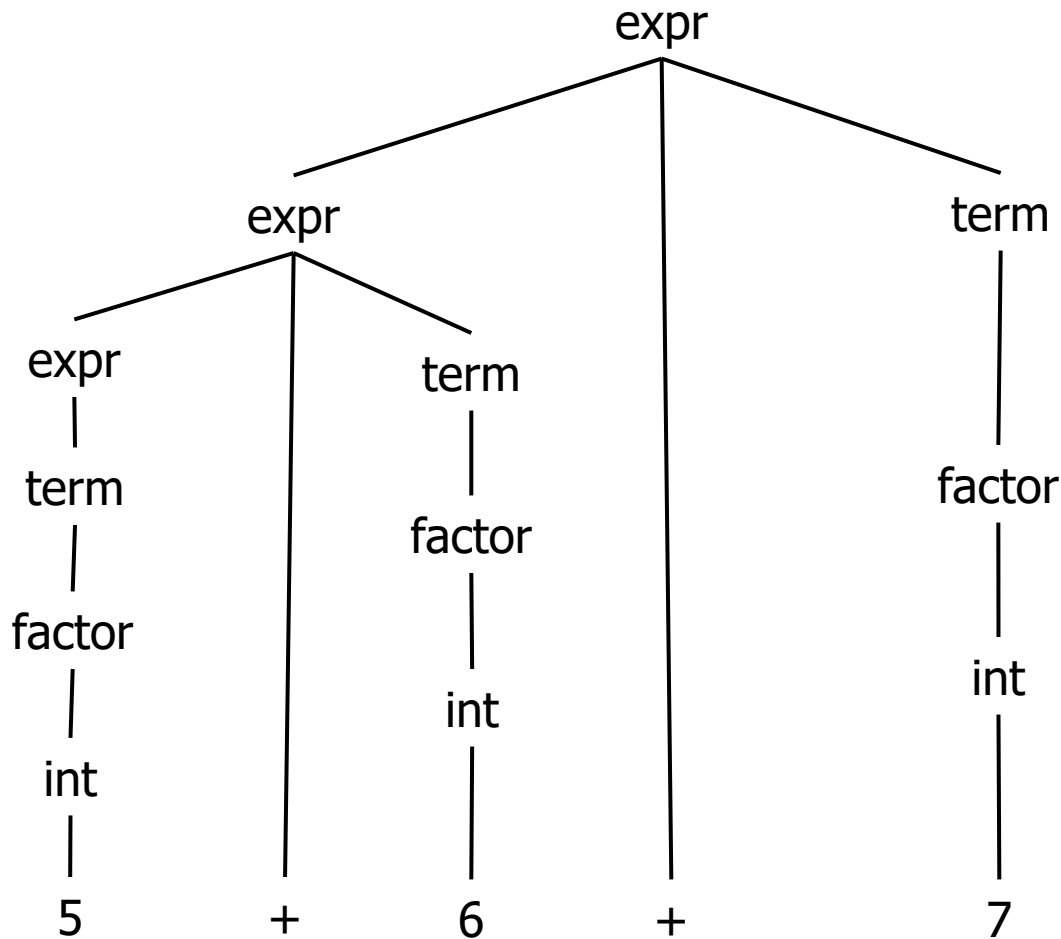


Separation of non-terminals enforces precedence

# Check:

## Derive $5 + 6 + 7$

$expr ::= expr + term \mid expr - term \mid term$   
 $term ::= term * factor \mid term / factor \mid factor$   
 $factor ::= int \mid ( expr )$   
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$



Note interaction  
between left- vs  
right-recursive  
rules and resulting  
associativity

Check:

Derive  $5 + (6 + 7)$

$expr ::= expr + term \mid expr - term \mid term$   
 $term ::= term * factor \mid term / factor \mid factor$   
 $factor ::= int \mid ( expr )$   
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$

(left as an exercise 😊)

# Another Classic Example

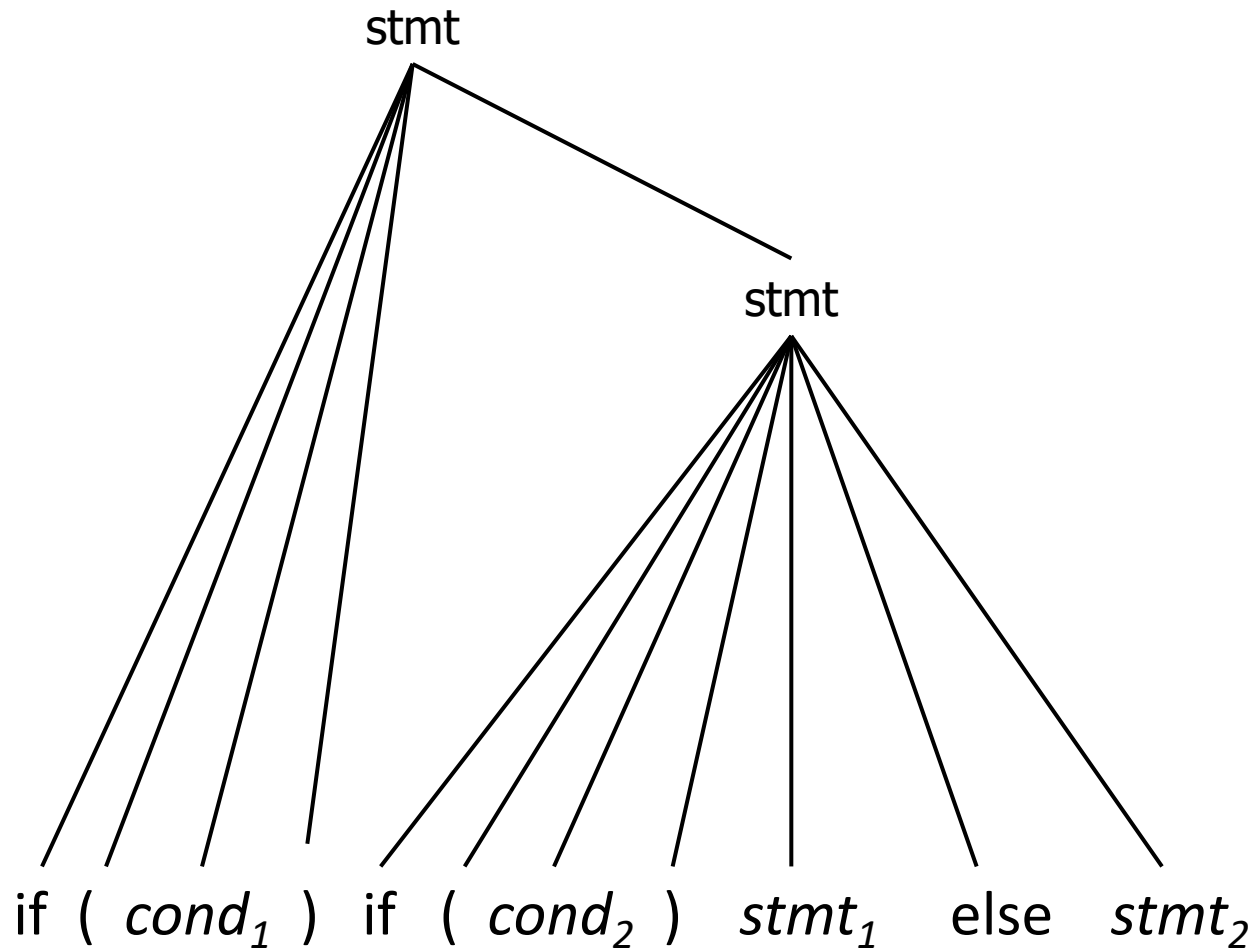
- Grammar for conditional statements

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

(This is the “dangling else” problem found in many, many grammars for languages, beginning with Algol 60)

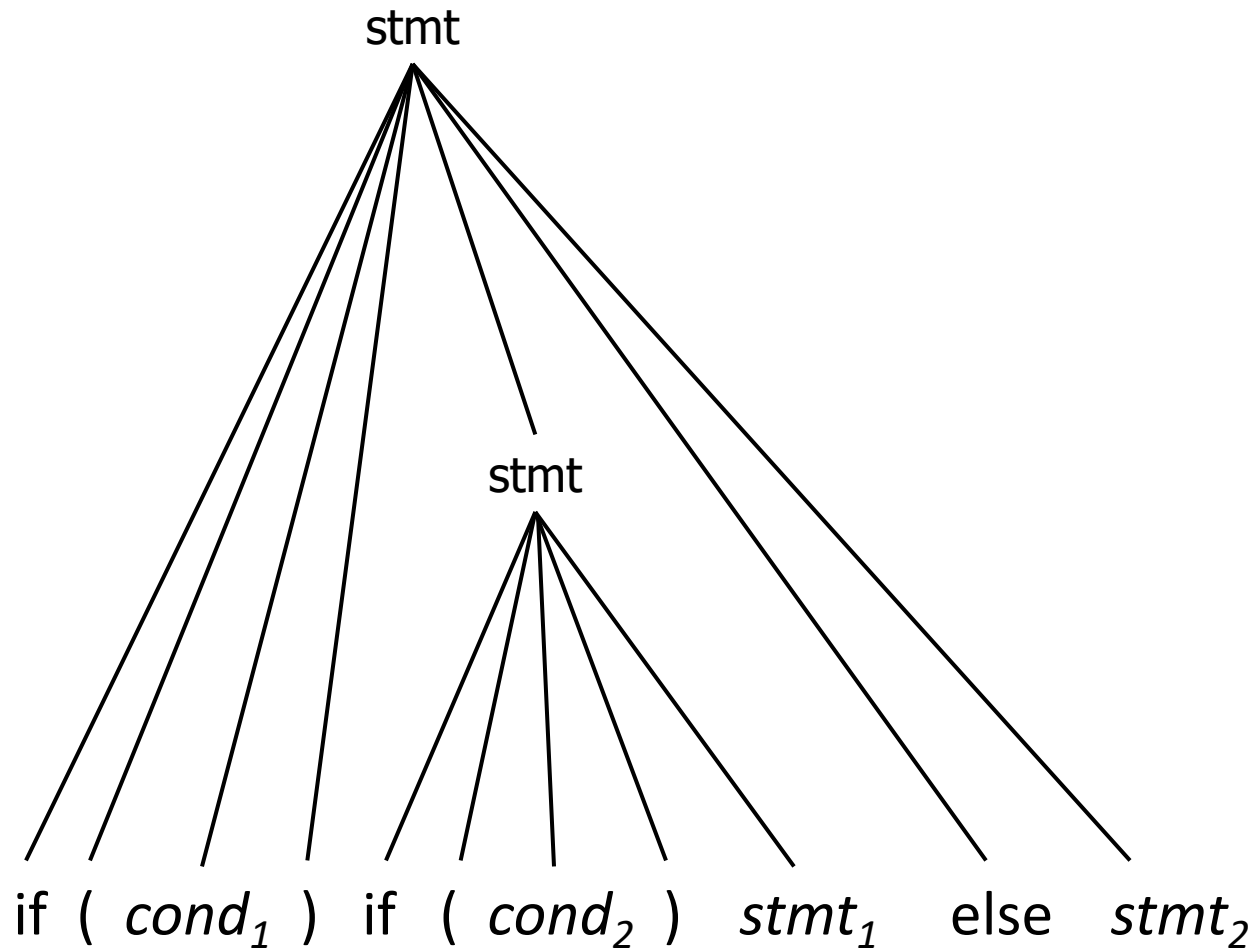
- Exercise: show that this is ambiguous
  - How?

# One Derivation

$$\begin{aligned} stmt &::= \text{if } ( cond ) stmt \\ &\quad | \text{if } ( cond ) stmt \text{ else } stmt \end{aligned}$$


```
if (cond1)  
  if (cond2)  
    stmt1  
  else  
    stmt2
```

# Another Derivation

$$\begin{aligned} stmt &::= \text{if } ( cond ) stmt \\ &\quad | \text{if } ( cond ) stmt \text{ else } stmt \end{aligned}$$


```
if (cond1)
  if (cond2)
    stmt1
  else
    stmt2
```



# Solving “if” Ambiguity

- Fix the grammar to separate if statements with else clause and if statements with no else
  - Done in Java reference grammar
  - Adds lots of non-terminals
- or, Change the language
  - But it’d better be ok with the language’s community to do this
- or, Use some ad-hoc rule in the parser
  - “else matches closest unpaired if”

# Resolving Ambiguity with Grammar (1)

Stmt ::= MatchedStmt | UnmatchedStmt

MatchedStmt ::= ... |

**if** ( Expr ) MatchedStmt **else** MatchedStmt

UnmatchedStmt ::= ... |

**if** ( Expr ) Stmt |

**if** ( Expr ) MatchedStmt **else** UnmatchedStmt

- formal, no additional rules beyond syntax
- can be more obscure than original grammar

# Check

```
Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... |
    if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
    if ( Expr ) MatchedStmt else UnmatchedStmt
```

(exercise 😊)

*if ( cond ) if ( cond ) stmt else stmt*

# Resolving Ambiguity with Grammar (2)

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

```
Stmt ::= ... |  
       if Expr then Stmt end |  
       if Expr then Stmt else Stmt end
```

- formal, clear, elegant
- allows sequence of Stmts in then and else branches, no { } needed
- extra end required for every if  
(But maybe this is a good idea anyway?)

# Parser Tools and 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.,  $\text{expr} ::= \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid \dots$  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 and Ambiguous Grammars

- 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 if and a few similar things)
- Parser tools normally allow for this
  - But be sure that what the tool does is really what you want
    - And that it's part of the permanent tool spec, so that v2 won't do something different (that you *don't* want!)

# Coming Attractions

- Next topic: LR parsing
  - Continue reading ch. 3