CSE 401/M501 – Compilers

Parsing & Context-Free Grammars

Spring 2022

Administrivia

- Reminders:
  - Project partner signup. Please fill out the form
    - ASAP, please, but by tomorrow, 11 pm in any case
    - Who's still looking for a partner?
      - Post to ed discussion thread. Mingle at end of class?
  - hw1 due Thur. night (regexps, etc.) via gradescope
    - ***** vs *: be clear about regexp operators vs characters. Avoid messy \e\s\c\a\p\e\s – I suggest <u>*</u>, [*] (underlined or bracketed for terminal) vs * (plain for operator). Add a short explanation (sentence or 2) to help grader with notation.
- In-person Office Hours
    - Leave doors open, avoid crowding, etc

Agenda for Today

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

Regular expressions have limits

- Famous example: $\{ a^n b^n \mid n \geq 0 \}$ is *not* regular
- Why care? Because stuff like this isn't either:

```
while(i<j) {
   if(a && (b > (c+exp(-d[e/f[g]])))){
     i = (i+(j-k))/(l*m/n-o);
   }}
```

Hmmm..., did I count all those ({ [ ] })'s correctly?

- To the rescue: Context-Free Grammars

Context-free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CFG)
- Compromise between
  – REs: can't nest (parens, e.g.) 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

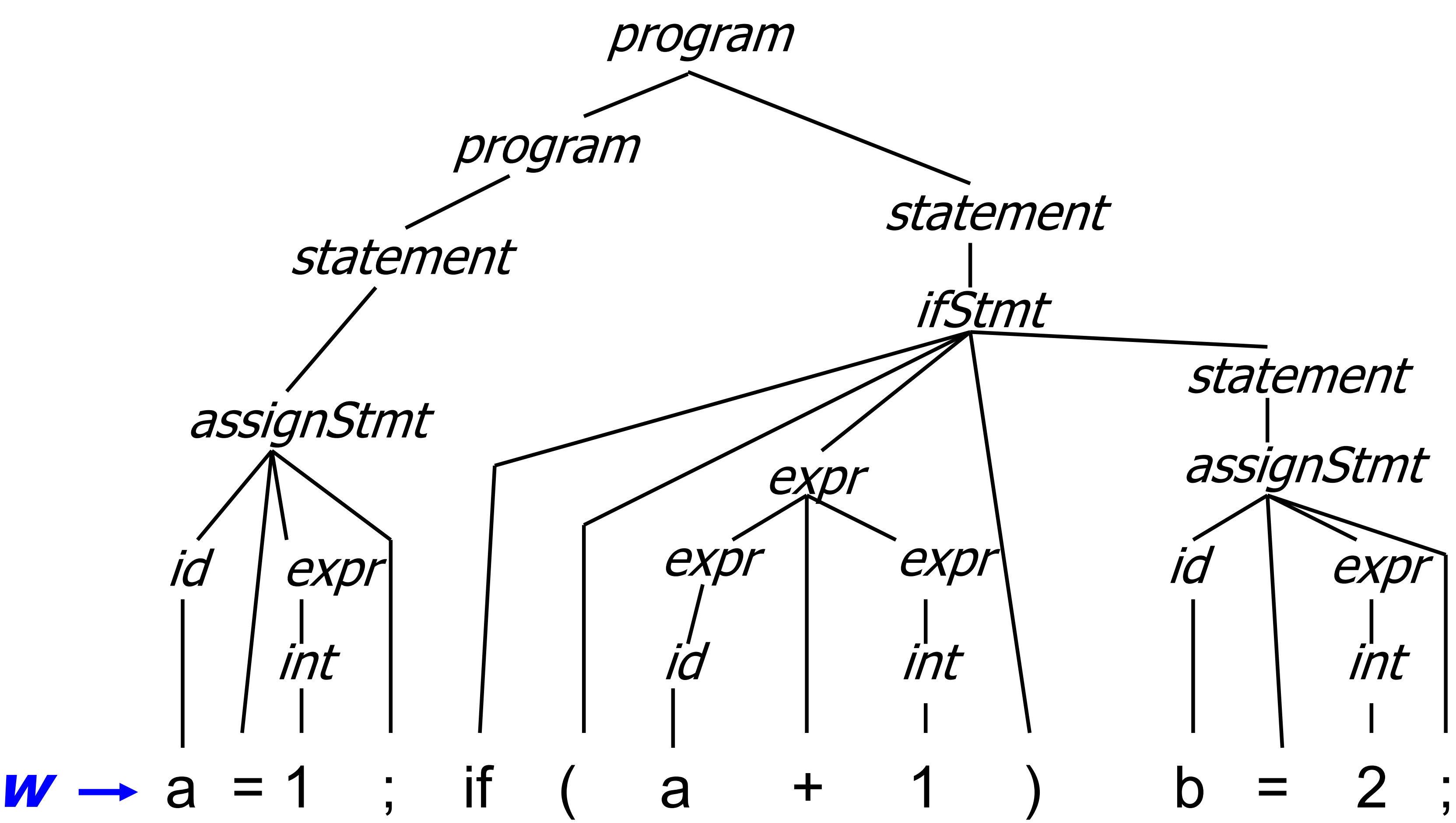
Grammars / Syntax Analysis / Parsing

- Use CFG to specify *syntax* of a programming language
- Syntax analysis/parsing
  - Establishes validity of input
  - Imposes useful *structure* on otherwise flat token stream
- Concrete syntax tree – *exactly* as per CFG
- Abstract syntax tree (AST):
  - Captures program structure, minus nits like "(", ")", ";"
  - Primary data structure for later phases of compilation
- Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing and building ASTs

Concrete syntax

G

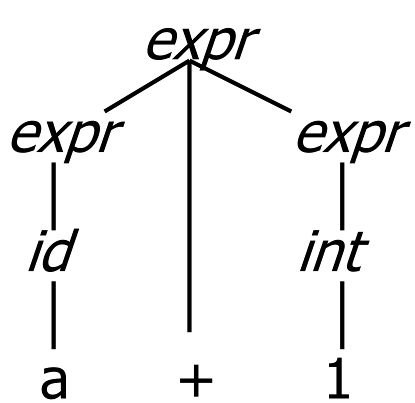
program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* = *expr* ;
ifStmt ::= if (*expr*) *statement*
expr ::= *id* | *int* | *expr* + *expr*
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



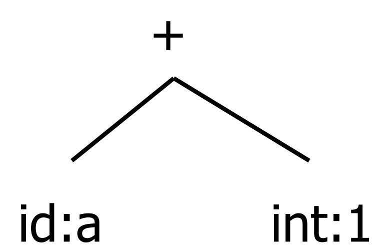
Concrete vs Abstract Syntax

- The full (concrete) parse tree includes all derivation details. Abstract Syntax Tree (AST) omits information that is necessary to parse the input, but not for later processing
- Example:

Concrete Syntax



Abstract Syntax



Context-Free Grammars

- Formally, a *grammar* G is a tuple $<N,\Sigma,P,S>$ where
  - N is a finite set of *non-terminal* symbols
  - Σ is a finite set of *terminal* symbols (alphabet)
  - P is a finite set of *productions*
    - A finite 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 Σ

w, x, y, z elements of Σ^*

A, B, C elements of N

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

α, β, γ elements of $(N \cup \Sigma)^*$

$A \rightarrow \alpha$ or $A ::= \alpha$ if (A, α) 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 α
  - transitive closure of $\Rightarrow$

Derivation Relations (2)

- $w\ A\ \gamma \Rightarrow_{lm} w\ \beta\ \gamma$ iff $A ::= \beta$ in P
  - derives leftmost (recall, by convention, w in Σ^*)
- $\alpha\ A\ w \Rightarrow_{rm} \alpha\ \beta\ w$ iff $A ::= \beta$ in P
  - derives rightmost (ditto)
- We will only be interested in leftmost and rightmost derivations – not random orderings
- Derivations vs trees: $\Rightarrow_{lm}$ is basically preorder traversal of tree; $\Rightarrow_{rm}$ is its mirror.

Languages

- For A in N, define $L(\mathrm{A}) = \{ \mathrm{w} \in \Sigma^* \mid \mathrm{A} \Rightarrow^* \mathrm{w} \}$
- $L(G) = L(S)$, where S is the start symbol of G
  - 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

Derivations and Parse Trees

- Derivation: a sequence of expansion steps, beginning with the start symbol and leading to a sequence of terminals
- Convenient formalism / textual representation
- Parsing Tree: convenient graphical representation and compiler data structure

Ambiguity

- Grammar G is *unambiguous* iff every w in L(G) has a unique leftmost (or rightmost) derivation
  - unique leftmost or unique rightmost implies the other
  - equivalent to saying “unique parse tree”
- A grammar without this property is *ambiguous*
  - But other grammars that generate the same language might be unambiguous
- We want unambiguous grammars for parsing, and for interpretability of the program

Example: Ambiguous Grammar for Arithmetic Expressions

expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*

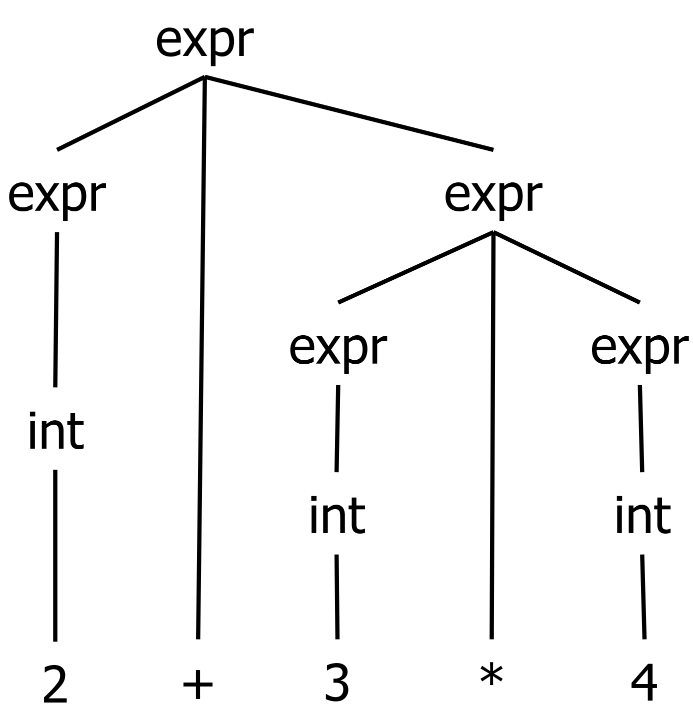
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 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

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

Example (cont)

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

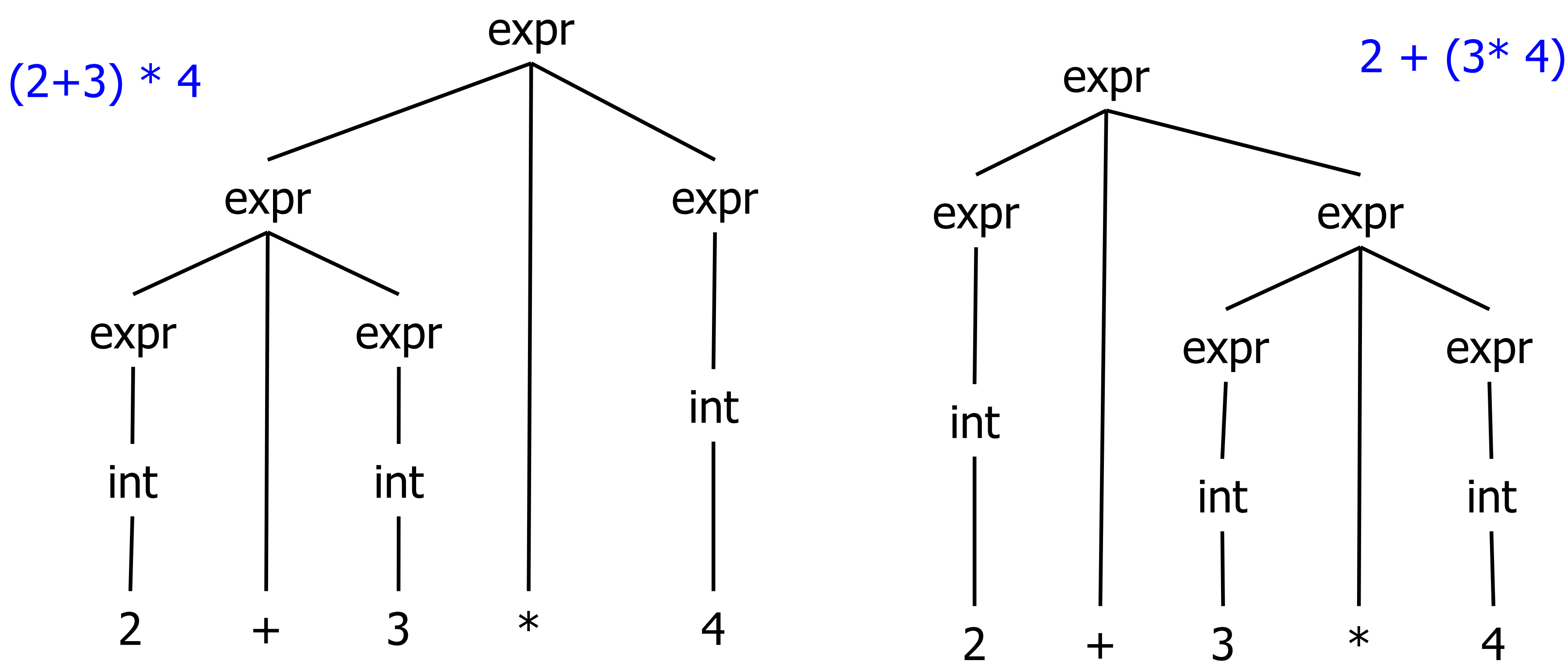


Example (cont)

expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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



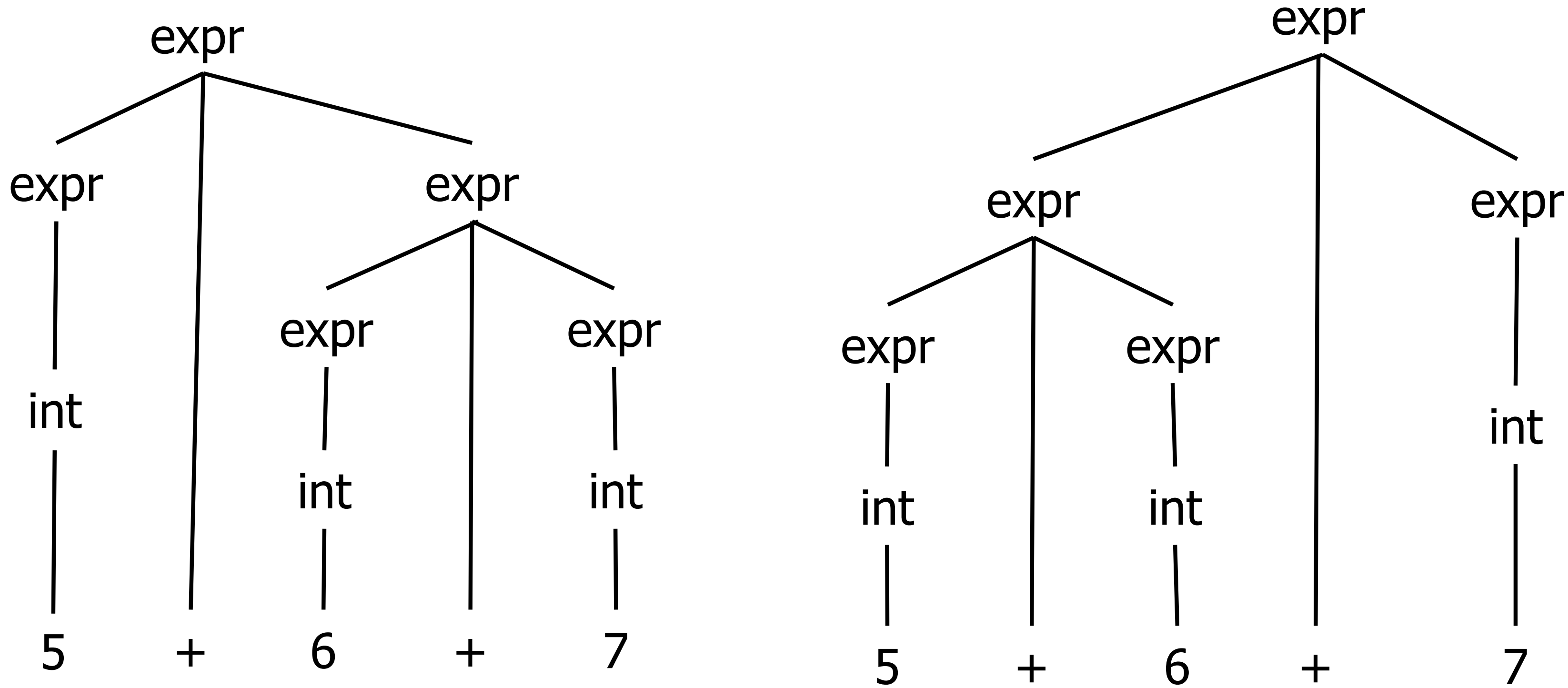
Another example

expr ::= *expr* + *expr* | *expr* - *expr*
| *expr* * *expr* | *expr* / *expr* | *int*

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 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
  - Forces the parser to recognize higher precedence subexpressions first
  - Use left- or right-recursion for left- or right-associative operators (non-associative operators are not recursive)

Classic Expression Grammar
(first used in ALGOL 60)

expr ::= *expr* + *term* | *expr* – *term* | *term*

term ::= *term* * *factor* | *term* / *factor* | *factor*

factor ::= *int* | (*expr*)

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

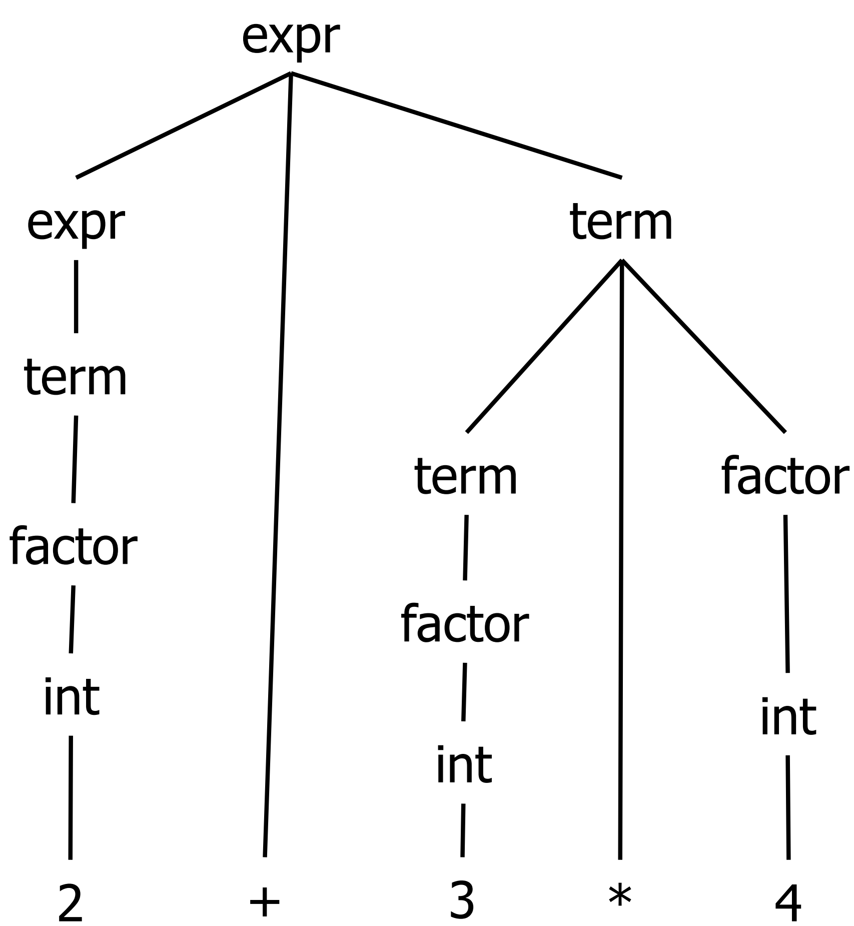
Check:
Derive 2 + 3 * 4

expr ::= *expr* + *term* | *expr* – *term* | *term*

term ::= *term* * *factor* | *term* / *factor* | *factor*

factor ::= *int* | (*expr*)

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7



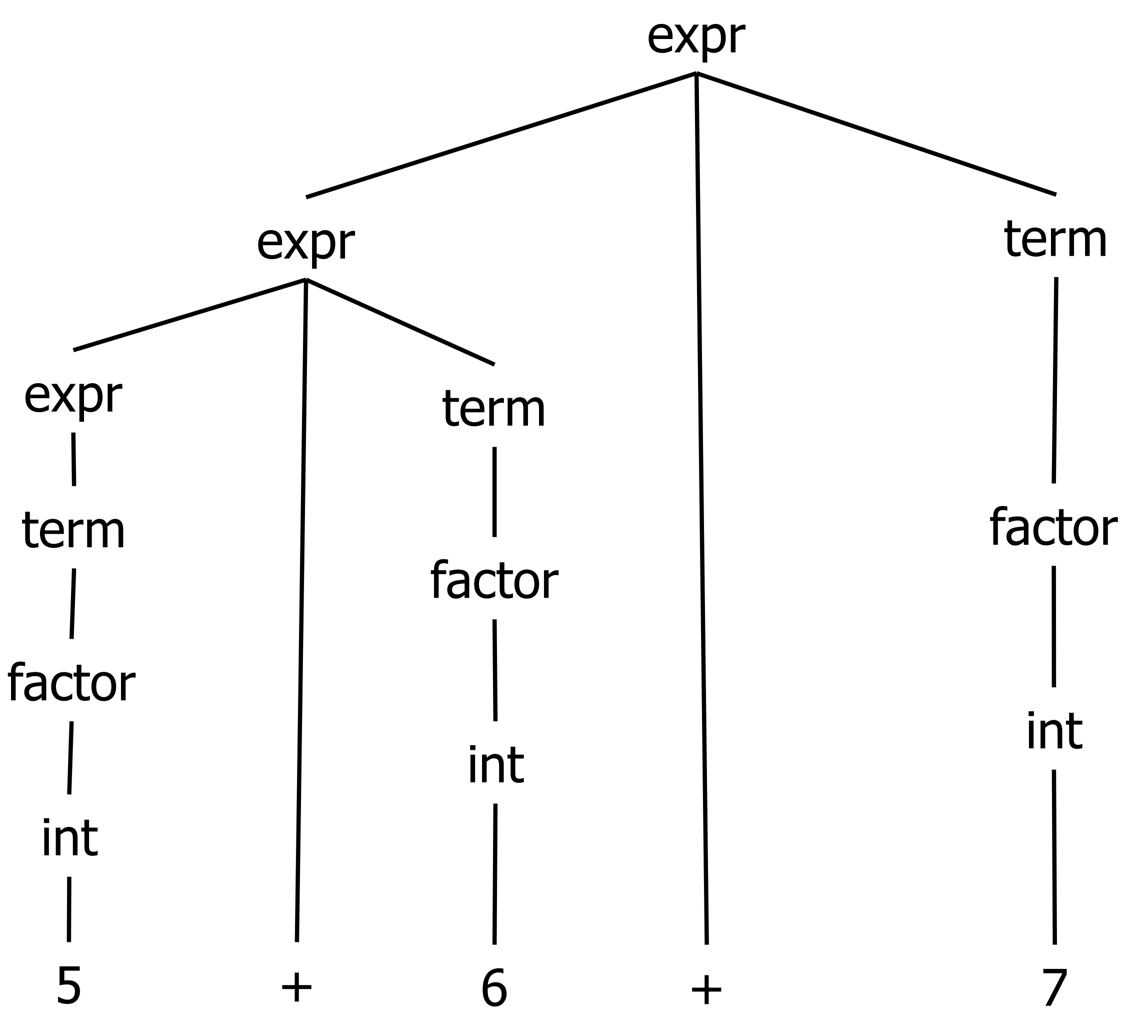
Separation of non-terminals enforces precedence *expr* ::= *expr* + *term* | *expr* – *term* | *term*

term ::= *term* * *factor* | *term* / *factor* | *factor*

factor ::= *int* | (*expr*)

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Check: Derive 5 + 6 + 7



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

Check: Derive 5 + (6 + 7)

expr ::= *expr* + *term* | *expr* – *term* | *term*

term ::= *term* * *factor* | *term* / *factor* | *factor*

factor ::= *int* | (*expr*)

int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

(left as an exercise ☺)

Another Classic: The Dangling "else"

- Grammar for conditional statements

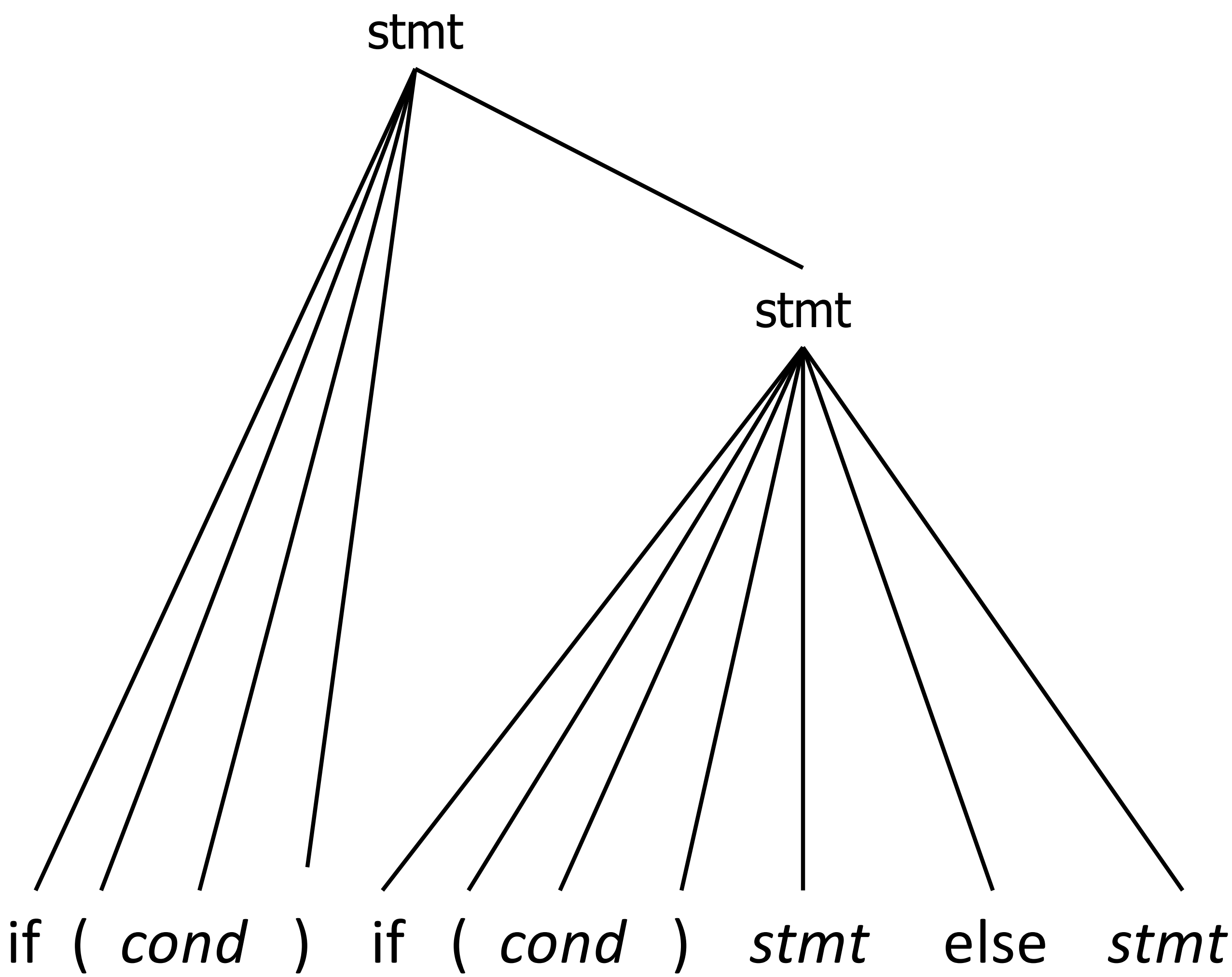
  stmt ::= if (*cond*) *stmt*

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

  - Exercise: show that this is ambiguous
    - How?

$stmt$ ::= if ($cond$) $stmt$
| if ($cond$) $stmt$ else $stmt$

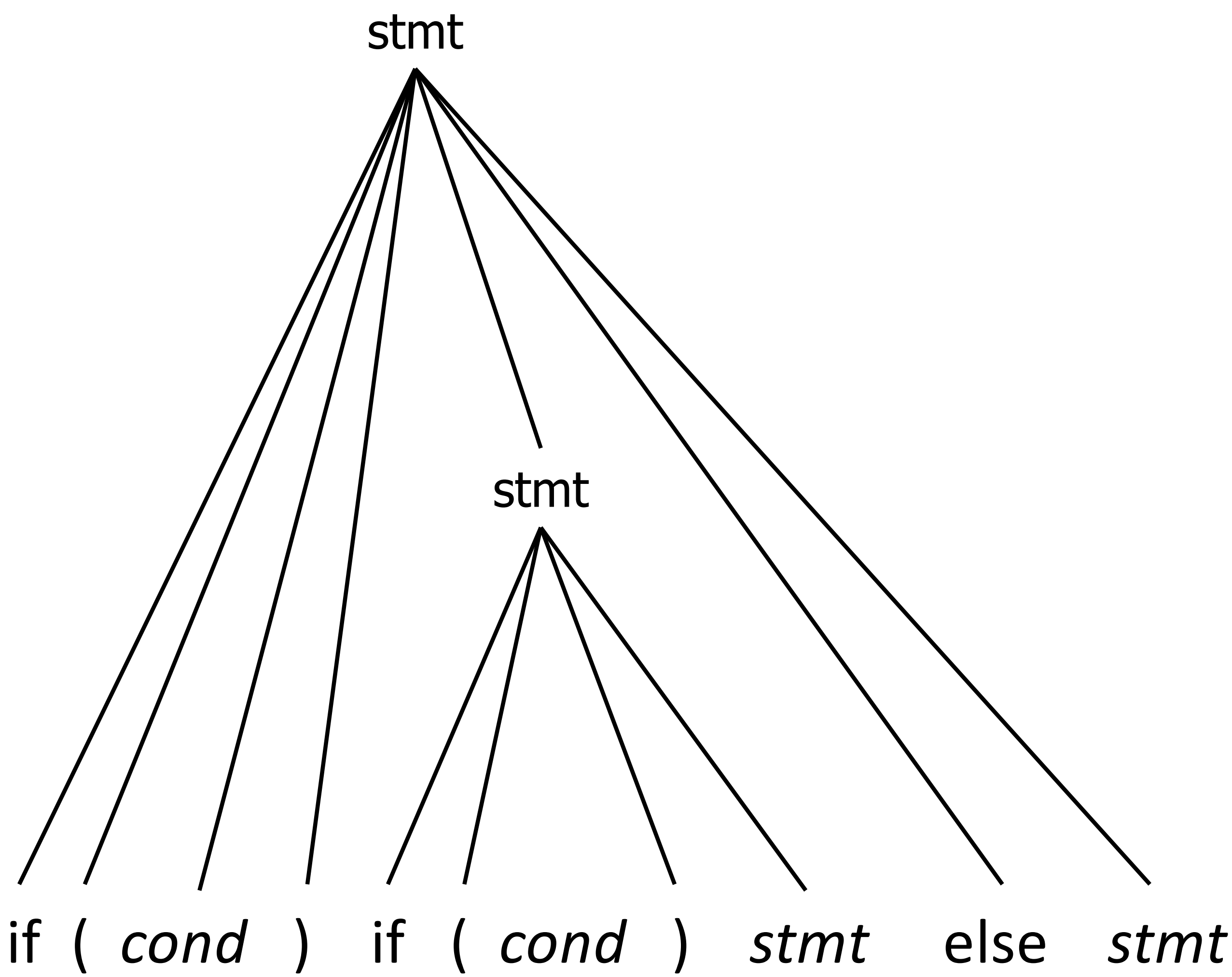
One Derivation



if (cond)
    if (cond)
        stmt
    else
        stmt

stmt ::= if (*cond*) *stmt*
| if (*cond*) *stmt* else *stmt*

Another Derivation




```
if (cond)
    if (cond)
        stmt
else
    stmt
```

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?)

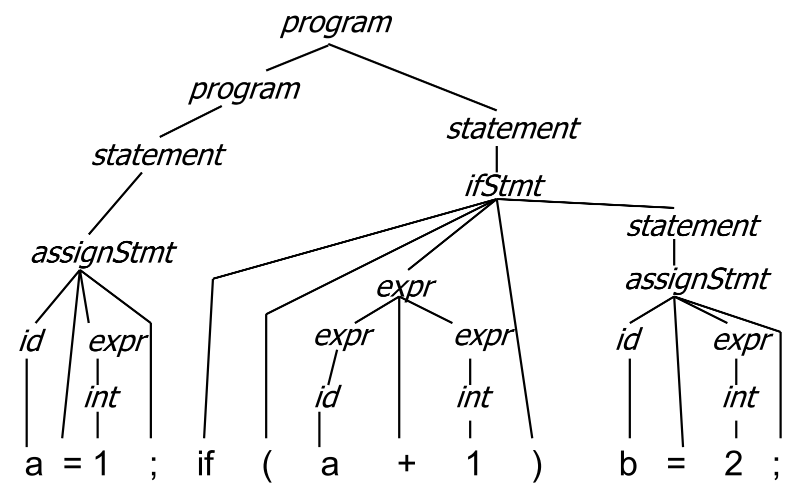
Parsing

- 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.)

At every step in the derivation, replace the *left-(right-) most* nonterminal

"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 compiler; not common in production compilers)
    - Can't generate great code in one pass, but useful if you need a quick 'n dirty working compiler

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., 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 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 else and 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