

Chapter 3

Grammars: Capsule Summary

Course “Compiler Construction”
Martin Steffen
Spring 2021



Chapter 3

Learning Targets of Chapter "Grammars: Capsule Summary".

1. (context-free) grammars + BNF
2. ambiguity and other properties
3. terminology: tokens, lexemes
4. different trees connected to grammars/parsing
5. derivations, sentential forms

The chapter corresponds to [1, Section 3.1–3.2] (or [2, Chapter 3]).



Chapter 3

Outline of Chapter "Grammars: Capsule Summary".

Introduction

Context-free grammars & BNF notation

Syntax-trees, concrete and abstract

Ambiguity

Chomsky hierarchy



Section

Introduction

Chapter 3 “Grammars: Capsule Summary”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Grammars and parsers



Syntax

- 2 chapters: grammars and parsers
- language specification resp. generation $\Leftrightarrow$ parsing

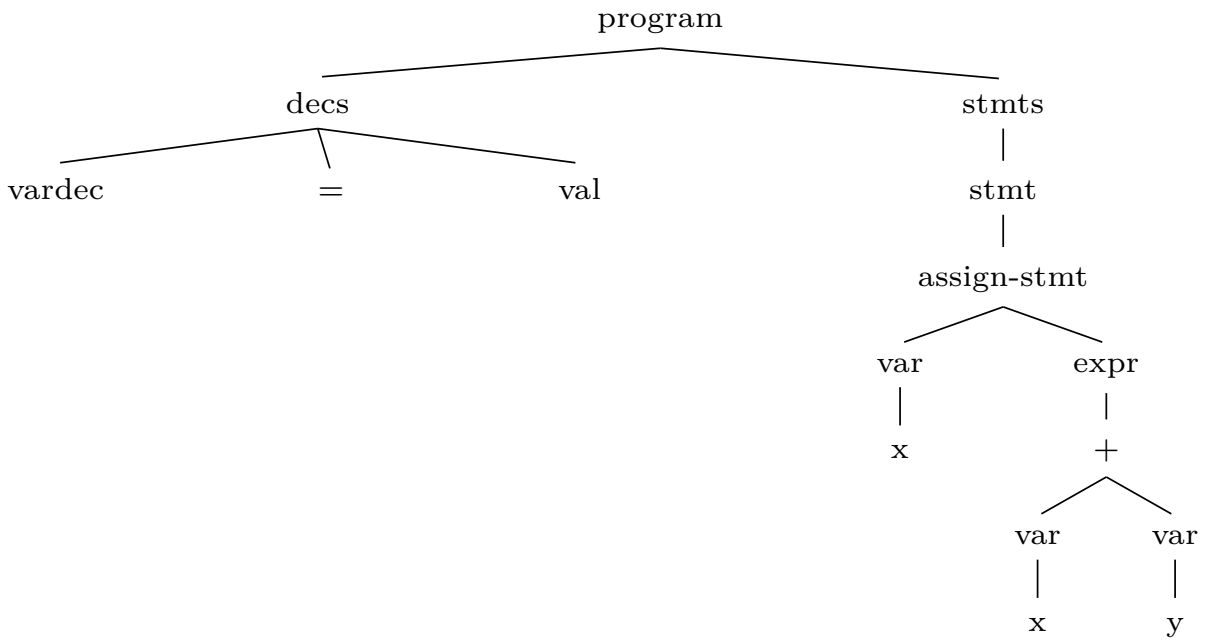
Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Bird's eye view on a parser



The task of **parsing** is, given a stream of "symbols" w and a grammar G, find a *derivation* from G that produces w.

Trees, trees, and more trees



abstract syntax trees $\Leftrightarrow$ parse trees/concrete syntax trees





Section

Context-free grammars & BNF notation

Chapter 3 “Grammars: Capsule Summary”
Course “Compiler Construction”
Martin Steffen
Spring 2021

(Context-free) grammars

Slogan: A CFG describes the syntax of a programming language. [a]

[a] And some say, regular expressions describe its microsyntax.



Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Definition

Definition (CFG)

A *context-free grammar* G is a 4-tuple $G = (\Sigma_T, \Sigma_N, S, P)$:

1. 2 disjoint finite alphabets of terminals Σ_T and
2. non-terminals Σ_N
3. 1 start-symbol $S \in \Sigma_N$ (a non-terminal)
4. productions P = finite subset of $\Sigma_N \times (\Sigma_N + \Sigma_T)^*$

tokens = terminals of the grammar



Notation for grammars: BNF

Backus-Naur form: What regular expressions are for regular languages, is BNF for context-free languages.



BNF example: expressions

$$
\begin{aligned}
exp &\rightarrow exp\ \ op\ \ exp \mid (\ exp\ ) \mid \textbf{number} \\
op &\rightarrow + \mid - \mid *
\end{aligned}
$$

$$
\begin{aligned}
exp &\rightarrow exp\ op\ exp \\
exp &\rightarrow (\ exp\ ) \\
exp &\rightarrow \textbf{number} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow *
\end{aligned}
\tag{1}
$$



Variations exist

```
<exp>  ::=   <exp> <op> <exp>
         |   ( <exp> )
         | NUMBER
<op>   ::=   +  |  -  |  *
```

$$
\begin{aligned}
exp \quad \rightarrow \quad & exp\ (\ ”+”\ \mid\ ”-”\ \mid\ ”*”\ )\ exp \\
\mid \quad & ”(”\ exp\ ”)” \ \mid\ ”number”
\end{aligned}
\tag{2}
$$



Grammars as language generators



Convention

We will use $\alpha_1, \alpha_2, \beta \ldots$ for word of terminals and nonterminals, $w, v_1 \ldots$ for words of *terminals*, and $A, B' \ldots$ for a single non-terminal. We use also X for arbitrary grammar symbols, a terminal, or non-terminal, or ϵ.

$$\mathcal{L}(G) = \{s \mid start \Rightarrow^* s \text{ and } s \in \Sigma_T^*\}$$

Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Derivation

$$
\begin{aligned}
\underline{exp} &\Rightarrow \underline{exp}\ op\ exp \\
&\Rightarrow (\underline{exp})\ op\ exp \\
&\Rightarrow (\underline{exp}\ op\ exp)\ op\ exp \\
&\Rightarrow (\mathbf{n}\ \underline{op}\ exp)\ op\ exp \\
&\Rightarrow (\mathbf{n}-\underline{exp})\ op\ exp \\
&\Rightarrow (\mathbf{n}-\mathbf{n})\underline{op}\ exp \\
&\Rightarrow (\mathbf{n}-\mathbf{n})*\underline{exp} \\
&\Rightarrow (\mathbf{n}-\mathbf{n})*\mathbf{n}
\end{aligned}
$$

left-most



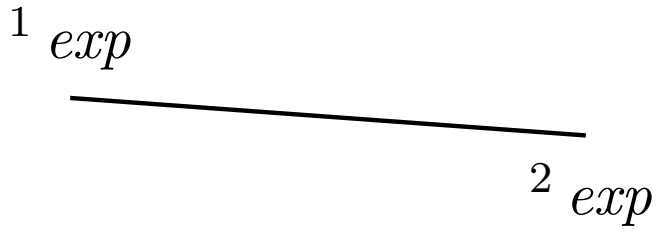
Parse tree

$^1\ exp$

right-most



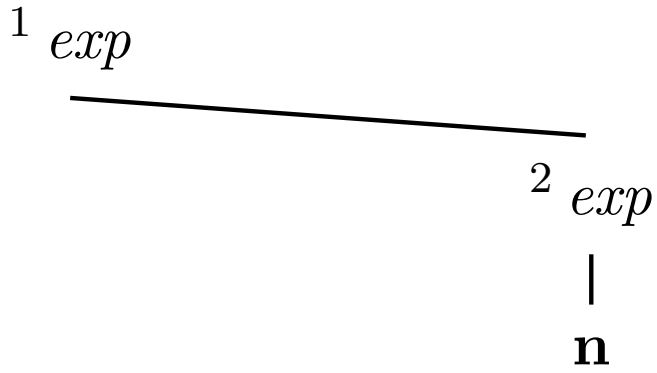
Parse tree



right-most



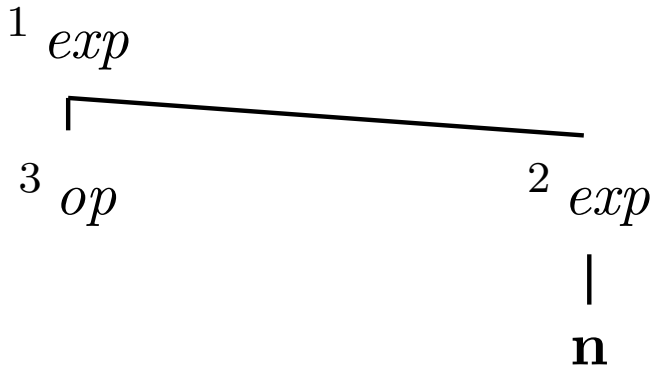
Parse tree



right-most



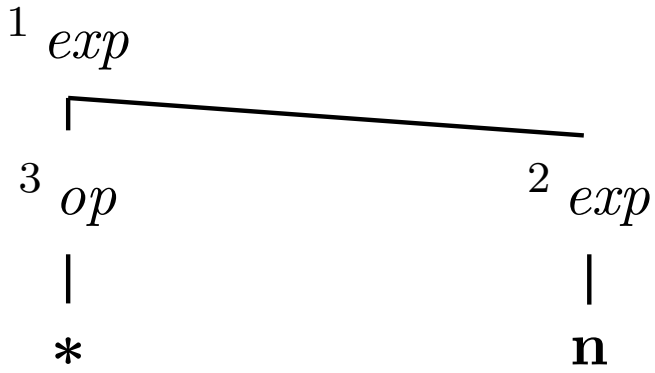
Parse tree



right-most



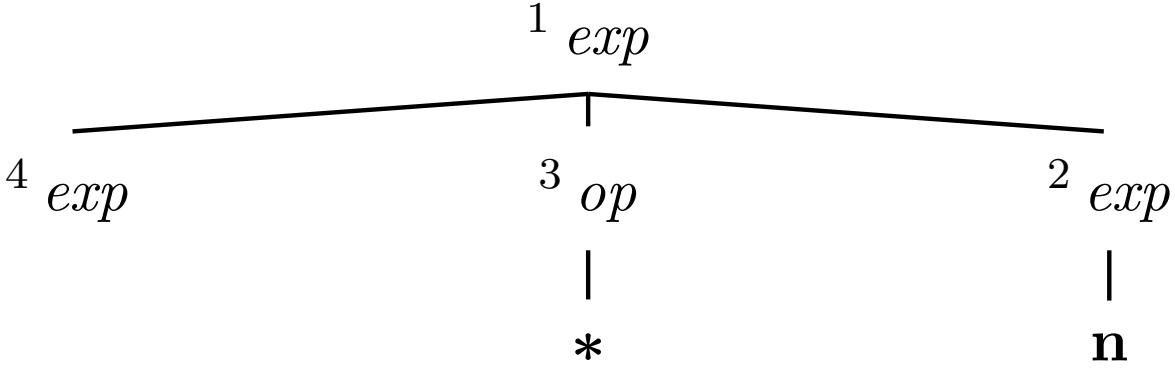
Parse tree



right-most



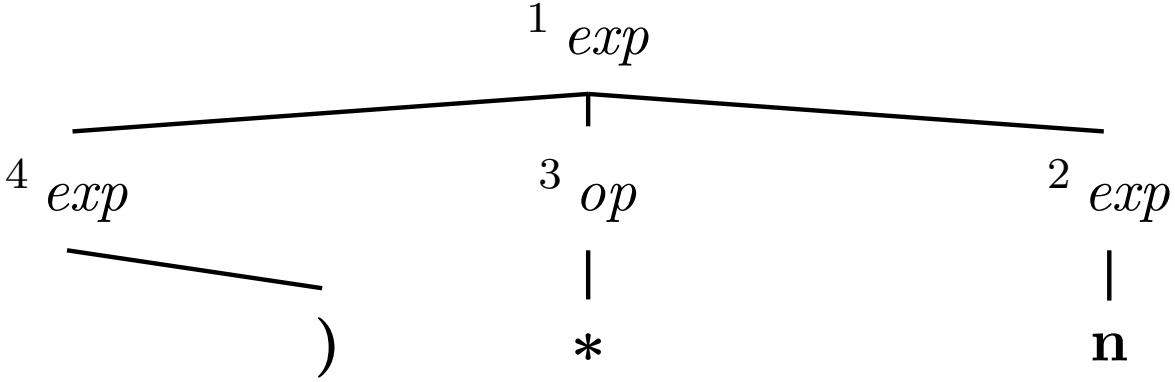
Parse tree



right-most

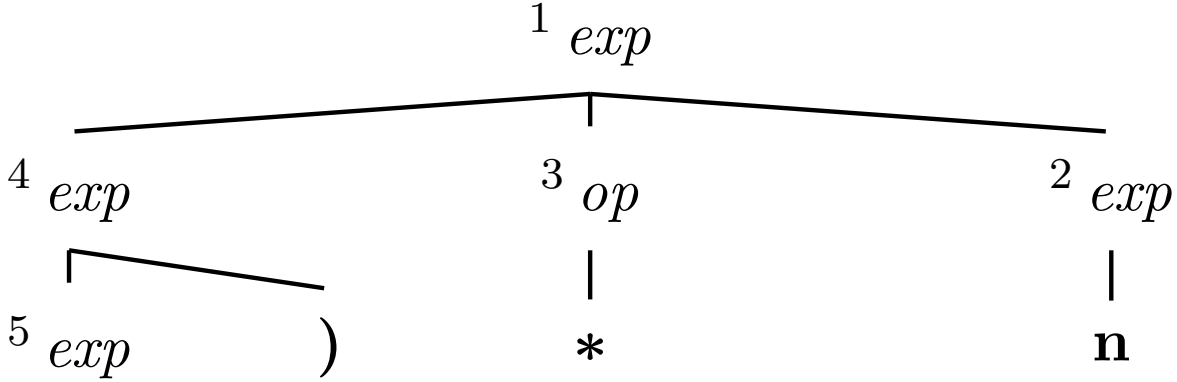


Parse tree



right-most

Parse tree



right-most



Targets & Outline

Introduction

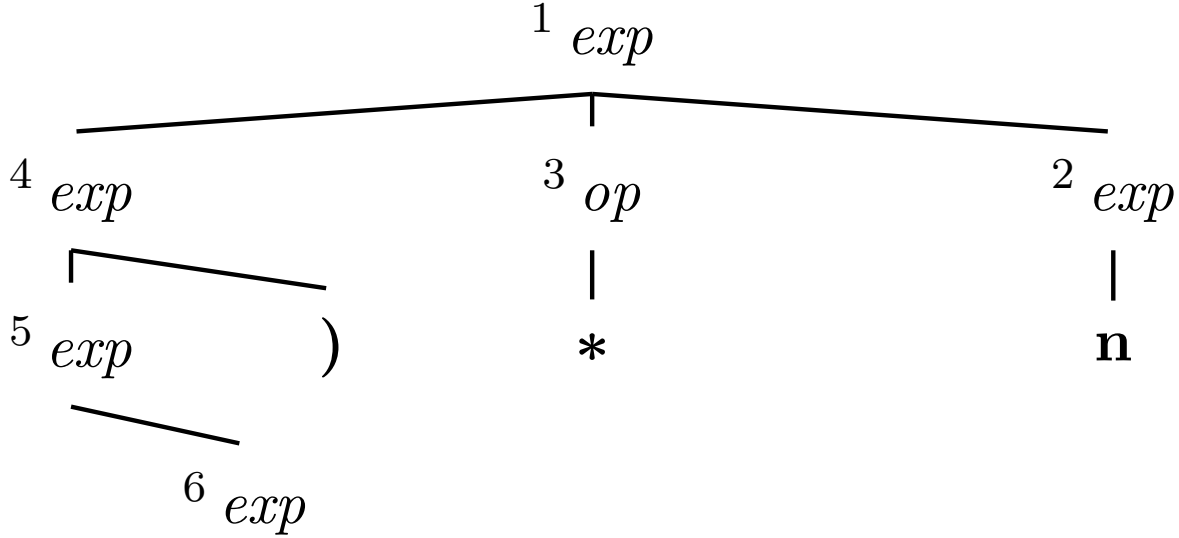
Context-free grammars & BNF notation

Syntax-trees, concrete and abstract

Ambiguity

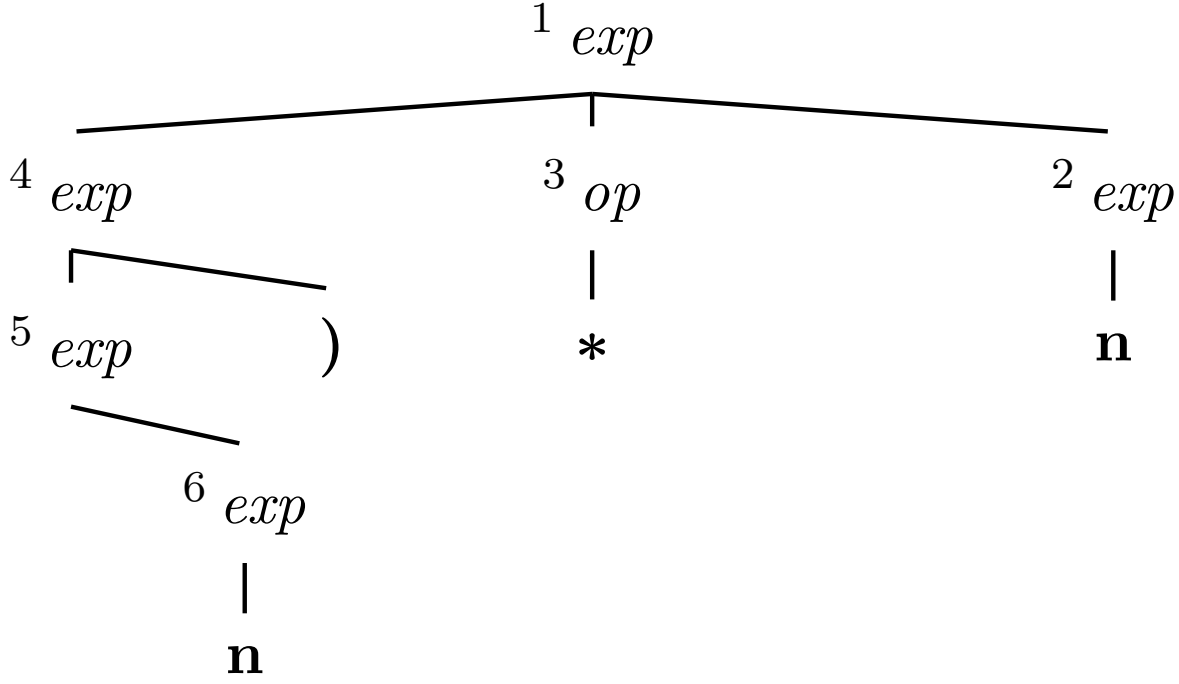
Chomsky hierarchy

Parse tree



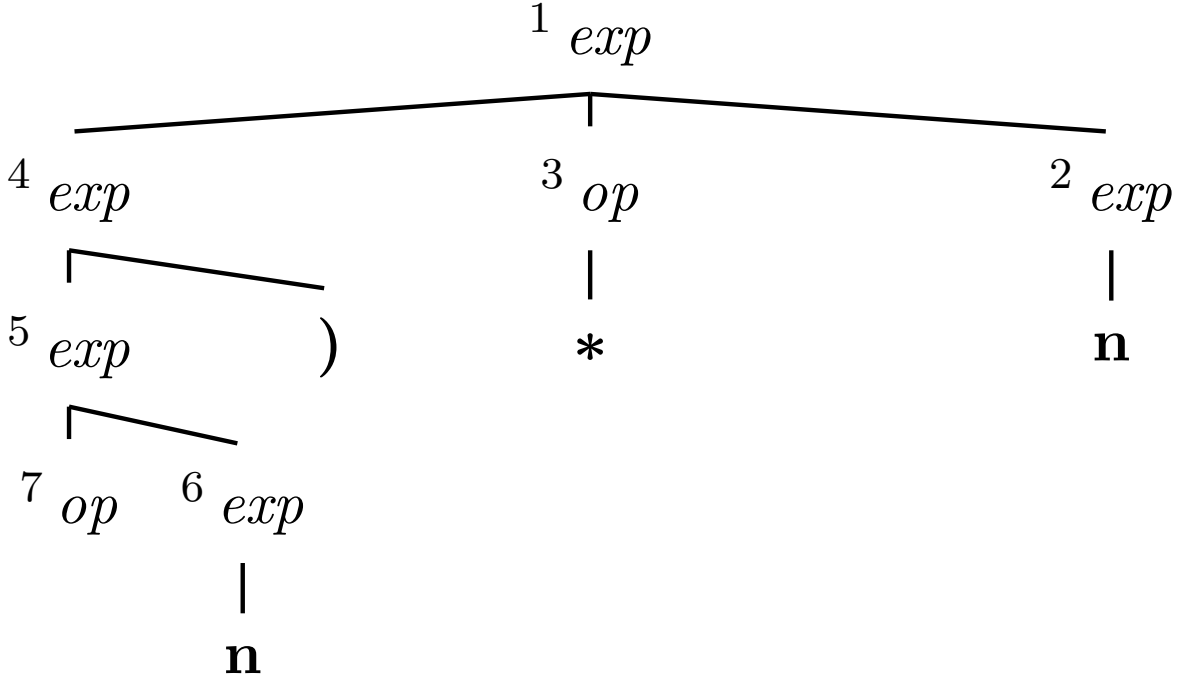
right-most

Parse tree



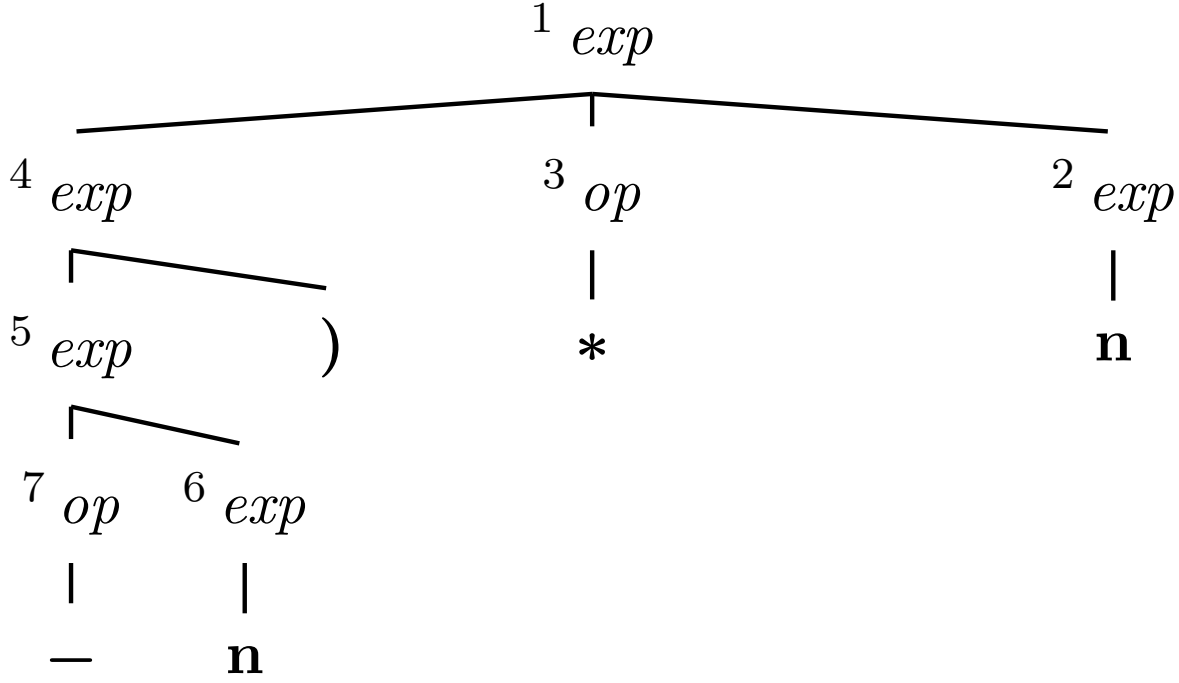
right-most

Parse tree



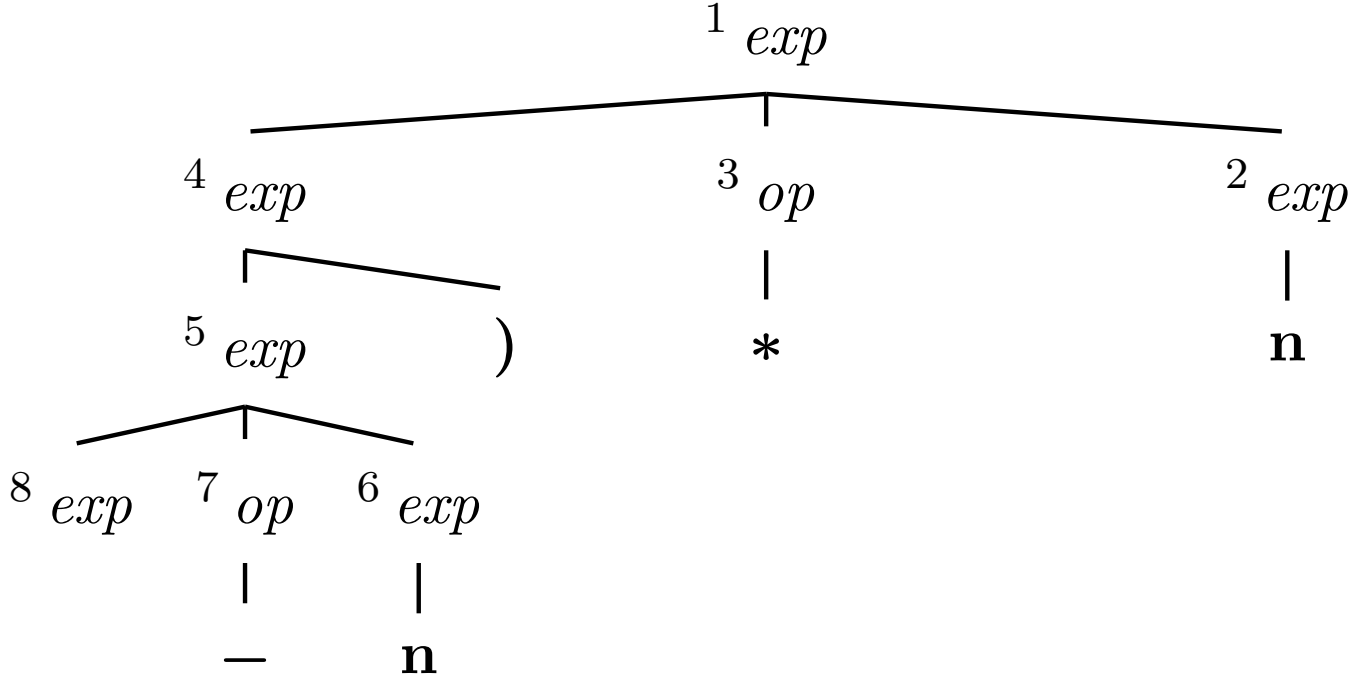
right-most

Parse tree



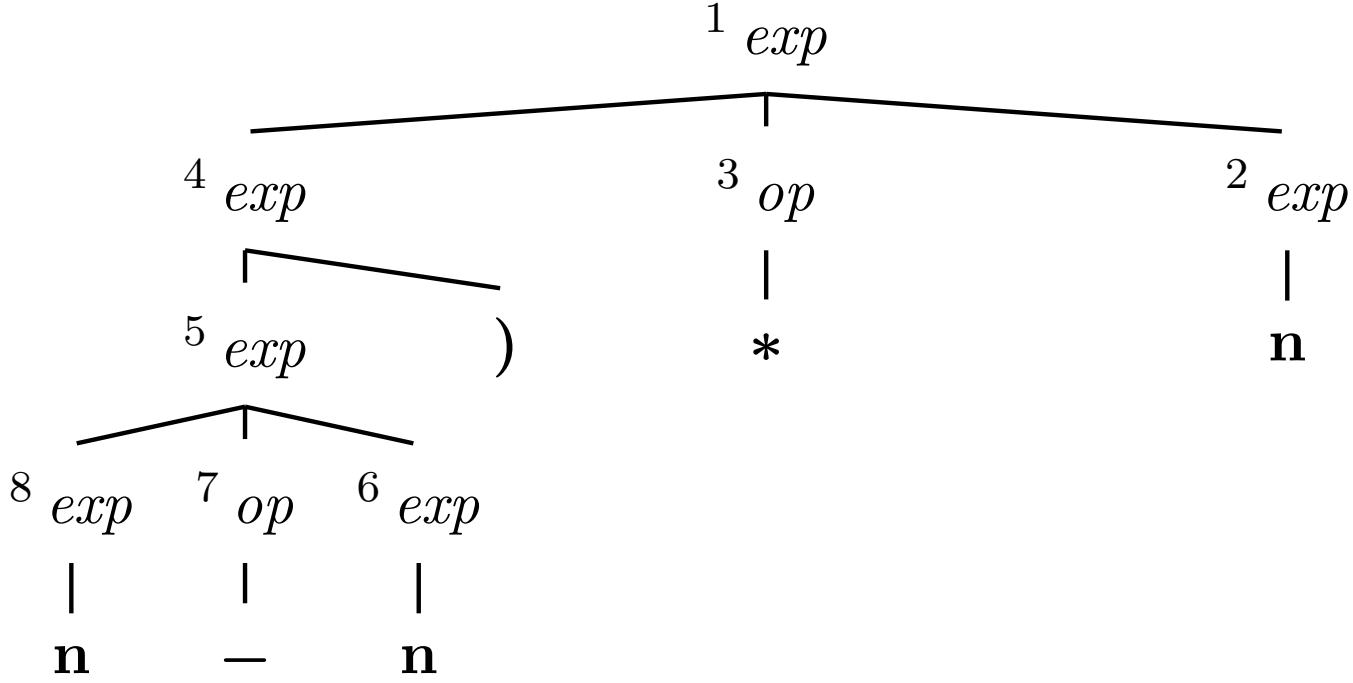
right-most

Parse tree



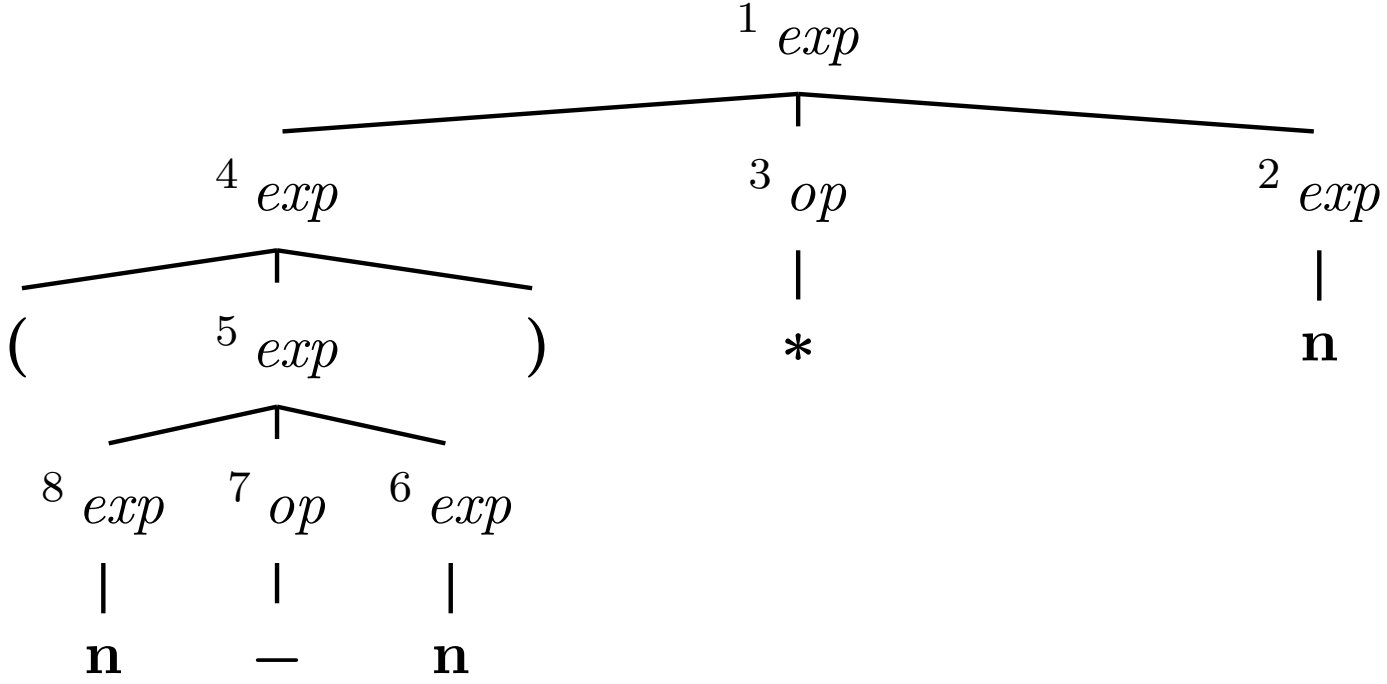
right-most

Parse tree



right-most

Parse tree



right-most

Common-sense sanitary conditions

- not formalized, just common-sense
- no “dead-code”





Section

Syntax-trees, concrete and abstract

Chapter 3 “Grammars: Capsule Summary”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Concrete and abstract syntax



Trees

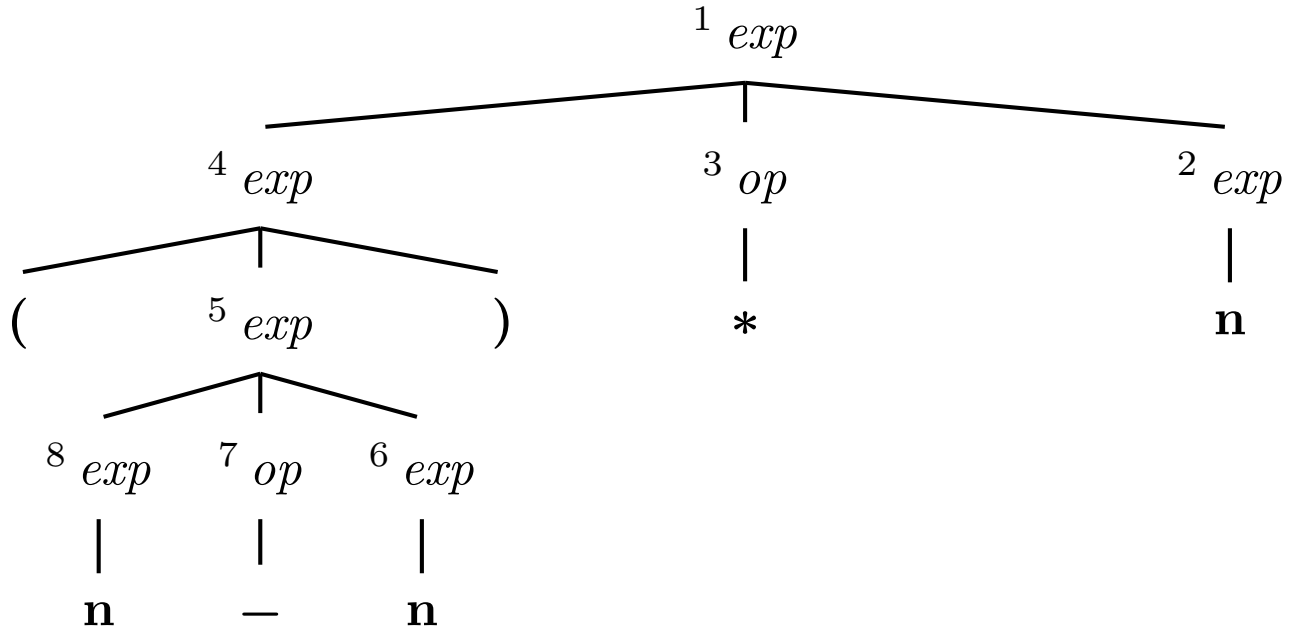
Abstract syntax tree vs. concrete syntax tree (=parse tree)

CST: reflecting the (essence of) the derivation process

AST: tree data structure, given back by the parser, intended to reflect the essence of the parse tree

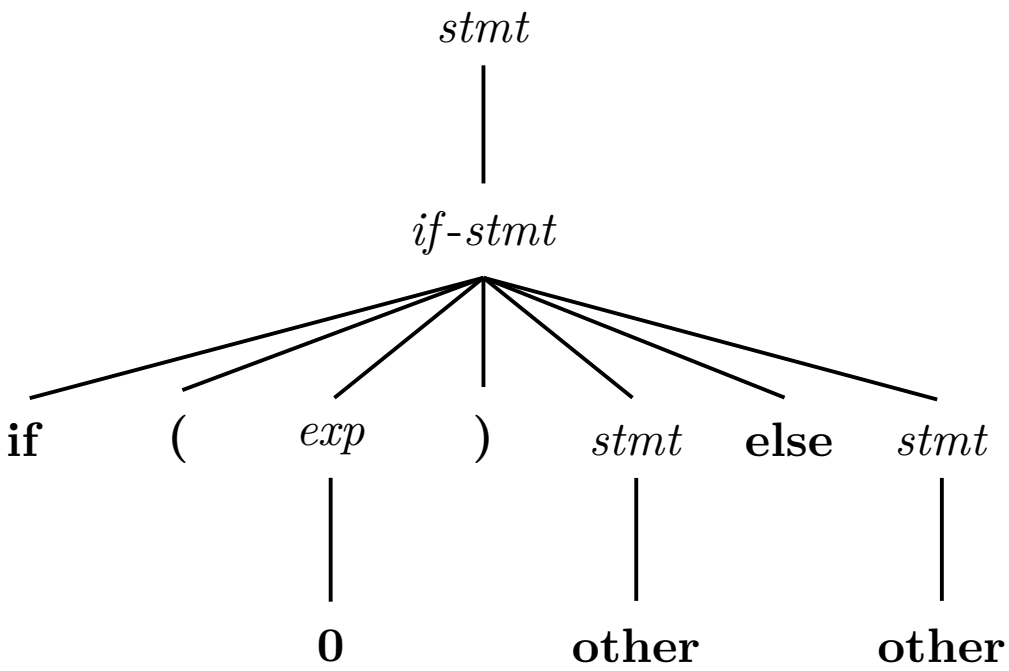
Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Parse tree



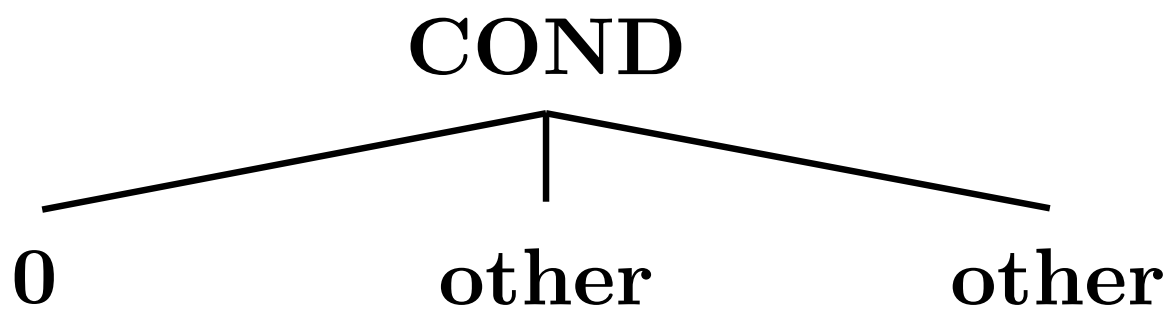
Trees

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{if\text{-}stmt} \mid \textbf{other} \\
\mathit{if\text{-}stmt} & \rightarrow & \textbf{if} \; (\; \mathit{exp} \;) \; \mathit{stmt} \\
 & \mid & \textbf{if} \; (\; \mathit{exp} \;) \; \mathit{stmt} \; \textbf{else} \; \mathit{stmt} \\
\mathit{exp} & \rightarrow & \textbf{0} \mid \textbf{1}
\end{array}
$$



Trees

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{if\text{-}stmt} \mid \textbf{other} \\
\mathit{if\text{-}stmt} & \rightarrow & \textbf{if} \; (\; \mathit{exp} \;) \; \mathit{stmt} \\
 & \mid & \textbf{if} \; (\; \mathit{exp} \;) \; \mathit{stmt} \; \textbf{else} \; \mathit{stmt} \\
\mathit{exp} & \rightarrow & \textbf{0} \mid \textbf{1}
\end{array}
$$





Section

Ambiguity

Chapter 3 “Grammars: Capsule Summary”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Ambiguity

Definition (Ambiguous grammar)

A grammar is *ambiguous* if there exists a word with *two different* parse trees.

- important concept
- "determinism"
- unwanted, but common

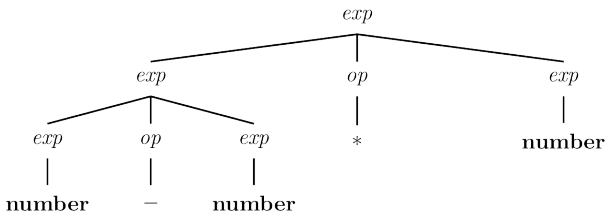


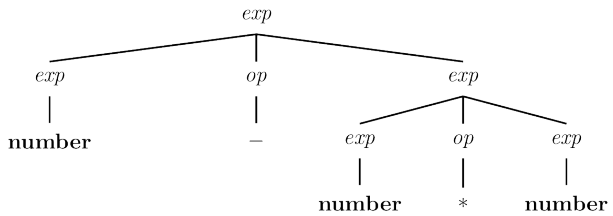
Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Expressions

What's that supposed to mean?

$$34 - 3 * 42$$





Ways to disambiguate

- reformulate grammar: **precendence cascade**
- specify **precedence** and **associativity** (outside the grammar)
- make syntax more verbose[1]
- (live with it), non-essential ambiguity

[1] Lisp, anyone ...



Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Precedence + associativity

binary op's	precedence	associativity
$+, -$	low	left
$\times, /$	higher	left
$\uparrow$	highest	right

cf. also *dangling-else*



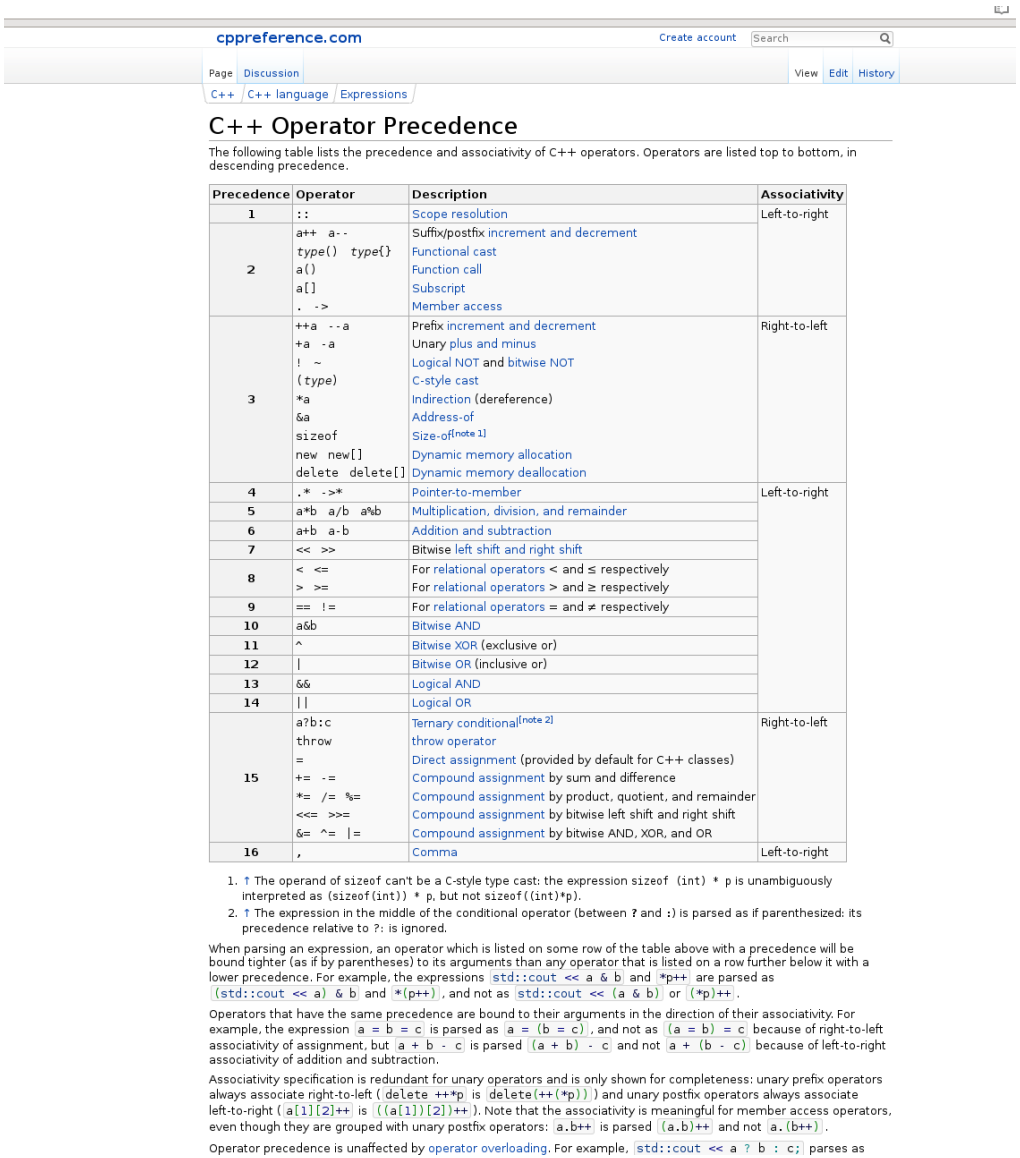
Precedence cascade

$$
\begin{array}{rcl}
exp & \rightarrow & exp\ addop\ term \mid term \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & term\ mulop\ factor \mid factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
$$



Targets & Outline
Introduction
Context-free grammars & BNF notation
Syntax-trees, concrete and abstract
Ambiguity
Chomsky hierarchy

Precedence + associativity: C++



cppreference.com

Create account | Search

Page | Discussion | View | Edit | History

C++ | C++ language | Expressions

C++ Operator Precedence

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
9	`== !=`	For relational operators = and ≠ respectively	
10	`a&b`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	
15	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Compound assignment by sum and difference	
	`*= /= %=`	Compound assignment by product, quotient, and remainder	
	`<<= >>=`	Compound assignment by bitwise left shift and right shift	
	`&= ^= |=`	Compound assignment by bitwise AND, XOR, and OR	
16	`,`	Comma	Left-to-right

1. ↑ The operand of `sizeof` can't be a C-style type cast: the expression `sizeof (int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
2. ↑ The expression in the middle of the conditional operator (between `?` and `:`) is parsed as if parenthesized: its precedence relative to `?:` is ignored.

When parsing an expression, an operator which is listed on some row of the table above with a precedence will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it with a lower precedence. For example, the expressions `std::cout << a & b` and `*p++` are parsed as `(std::cout << a) & b` and `*(p++)`, and not as `std::cout << (a & b)` or `(*p)++`.

Operators that have the same precedence are bound to their arguments in the direction of their associativity. For example, the expression `a = b = c` is parsed as `a = (b = c)`, and not as `(a = b) = c` because of right-to-left associativity of assignment, but `a + b - c` is parsed `(a + b) - c` and not `a + (b - c)` because of left-to-right associativity of addition and subtraction.

Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity is meaningful for member access operators, even though they are grouped with unary postfix operators: `a.b++` is parsed `(a.b)++` and not `a.(b++)`.

Operator precedence is unaffected by operator overloading. For example, `std::cout << a ? b : c;` parses as



Targets & Outline

Introduction

Context-free grammars & BNF notation

Syntax-trees, concrete and abstract

Ambiguity

Chomsky hierarchy



Section

Chomsky hierarchy

Chapter 3 “Grammars: Capsule Summary”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Chomsky hierarchy

	rule format	languages	machines	closed
3	$A \to aB$, $A \to a$	regular	NFA, DFA	all
2	$A \to \alpha_1 \beta \alpha_2$	CF	pushdown automata	$\cup$, $*$, $\circ$
1	$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$	context-sensitive	(linearly restricted automata)	all
0	$\alpha \to \beta$, $\alpha \neq \epsilon$	recursively enumerable	Turing machines	all, except complement

References I



Bibliography

[1] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

[2] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.