# INF5110 – Compiler Construction

Spring 2017

# Outline

# INF5110 – Compiler Construction

Grammars

Spring 2017

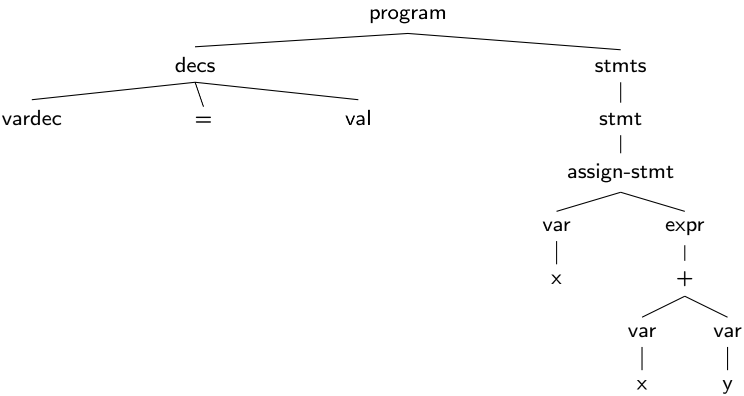
# Outline

# Outline

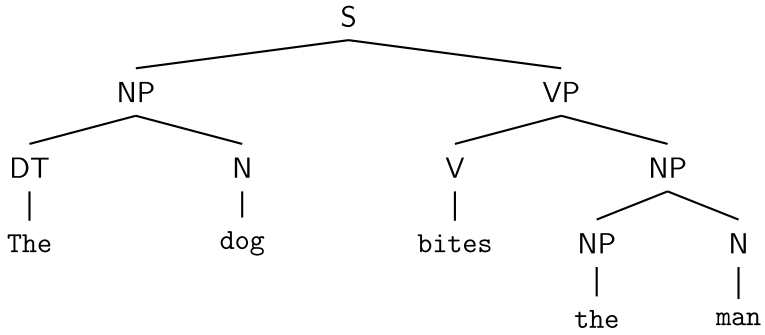
# Bird's eye view of a parser


- *check* that the token sequence correspond to a *syntactically correct* program
  - if yes: yield *tree* as intermediate representation for subsequent phases
  - if not: give *understandable* error message(s)
- we will encounter various kinds of trees
  - derivation trees (derivation in a (context-free) grammar)
  - *parse* tree, *concrete syntax tree*
  - *abstract syntax trees*
- mentioned tree forms hang together, dividing line a bit fuzzy
- result of a parser: typically AST

# Sample syntax tree



# Natural-language parse tree



# "Interface" between scanner and parser

- remember: task of scanner = "chopping up" the input char stream (throw away white space etc) and *classify* the pieces (1 piece = *lexeme*)
- classified lexeme = token
- sometimes we use ⟨`integer`, "42"⟩
  - `integer`: "class" or "type" of the token, also called *token name*
  - "42" : *value of the token attribute* (or just value). Here: directly the *lexeme* (a string or sequence of chars)
- a note on (sloppyness/ease of) terminology: often: the token name is simply just called the token
- for (context-free) grammars: the *token (symbol)* corrresponds there to terminal symbols (or terminals, for short)

# Outline

- in this chapter(s): focus on context-free grammars
- thus here: grammar = CFG
- as in the context of regular expressions/languages: *language* = (typically infinite) set of words
- grammar = formalism to unambiguously specify a language
- intended language: all syntactically correct programs of a given progamming language

### Slogan

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

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

- note: a compiler might reject some syntactically correct programs, whose violations *cannot* be captured by CFGs. That is done by *subsequent* phases (like type checking).

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

- terminal symbols: corresponds to tokens in parser = basic building blocks of syntax
- non-terminals: (e.g. "expression", "while-loop", "method-definition" ...)
- grammar: generating (via "derivations") languages
- parsing: the *inverse* problem

$\Rightarrow$ CFG = specification

# BNF notation

- popular & common format to write CFGs, i.e., describe context-free languages
- named after *pioneering* (seriously) work on Algol 60
- notation to write productions/rules + some extra meta-symbols for convenience and grouping

### Slogan: Backus-Naur form

What regular expressions are for regular languages is BNF for context-free languages.

# "Expressions" in BNF

$$
\begin{aligned}
exp &\rightarrow exp \; op \; exp \mid (\, exp \,) \mid \mathbf{number} \\
op &\rightarrow + \mid - \mid *
\end{aligned}
\tag{1}
$$

- "$\rightarrow$" indicating productions and " $\mid$ " indicating alternatives [1]
- convention: terminals written **boldface**, non-terminals *italic*
- also simple math symbols like "+" and "(" are meant above as terminals
- start symbol here: *exp*
- remember: terminals like **number** correspond to tokens, resp. token classes. The attributes/token values are not relevant here.

[1] The grammar can be seen as consisting of 6 productions/rules, 3 for *expr* and 3 for *op*, the $\mid$ is just for convenience. Side remark: Often also ::= is used for $\rightarrow$.

# Different notations

- BNF: notationally not 100% "standardized" across books/tools
- "classic" way (Algol 60):

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

- Extended BNF (EBNF) and yet another style

$$
\begin{array}{rcl}
exp & \rightarrow & exp\ (\ "+"\ \mid\ "-"\ \mid\ "*"\ )\ exp \\
 & \mid & "("\ exp\ ")"\ \mid\ "number"
\end{array}
\tag{2}
$$

- note: parentheses as terminals vs. as *metasymbols*

# Different ways of writing the same grammar

- directly written as 6 pairs (6 rules, 6 productions) from $\Sigma_N \times (\Sigma_N \cup \Sigma_T)^*$, with "$\rightarrow$" as nice looking "separator":

$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{exp}\ \mathit{op}\ \mathit{exp} \\
\mathit{exp} & \rightarrow & \textbf{(}\ \mathit{exp}\ \textbf{)} \\
\mathit{exp} & \rightarrow & \textbf{number} \\
\mathit{op} & \rightarrow & \textbf{+} \\
\mathit{op} & \rightarrow & \textbf{-} \\
\mathit{op} & \rightarrow & \textbf{*}
\end{array}
\tag{3}
$$

- choice of non-terminals: irrelevant (except for human readability):

$$
\begin{array}{rcl}
E & \rightarrow & E\ O\ E \mid \textbf{(}\ E\ \textbf{)} \mid \textbf{number} \\
O & \rightarrow & \textbf{+} \mid \textbf{-} \mid \textbf{*}
\end{array}
\tag{4}
$$

- still: we count 6 productions

# Grammars as language generators

## Deriving a word:

Start from start symbol. Pick a "matching" rule to rewrite the current word to a new one; repeat until *terminal* symbols, only.

- *non-deterministic* process
- rewrite relation for derivations:
    - one step rewriting: $w_1 \Rightarrow w_2$
    - one step using rule $n$: $w_1 \Rightarrow_n w_2$
    - many steps: $\Rightarrow^*$ etc.

## Language of grammar $G$

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

# Example derivation for (number−number)∗number

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

- underline the "place" were a rule is used, i.e., an *occurrence* of the non-terminal symbol is being rewritten/expanded
- here: *leftmost* derivation[2]

[2] We'll come back to that later, it will be important.

# Rightmost derivation

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

- other ("mixed") derivations for the same word possible

# Some easy requirements for reasonable grammars

- all symbols (terminals and non-terminals): should occur in a some word derivable from the start symbol
- words containing only non-terminals should be derivable
- an example of a silly grammar $G$ (start-symbol $A$)

$$\begin{array}{lcl} A & \to & B\mathbf{x} \\ B & \to & A\mathbf{y} \\ C & \to & \mathbf{z} \end{array}$$

- $\mathcal{L}(G) = \varnothing$
- those "sanitary conditions": very minimal "common sense" requirements

# Parse tree

- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]

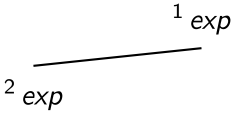
$^{1}$ *exp*

- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3]There will be *abstract* syntax trees, as well.

# Parse tree

- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]

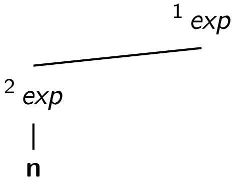



- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3] There will be *abstract* syntax trees, as well.

# Parse tree

- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]

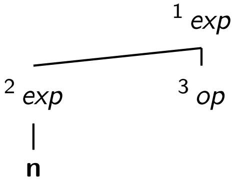



- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3] There will be *abstract* syntax trees, as well.

# Parse tree

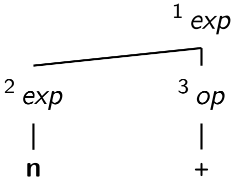
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]




- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3] There will be *abstract* syntax trees, as well.

- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]

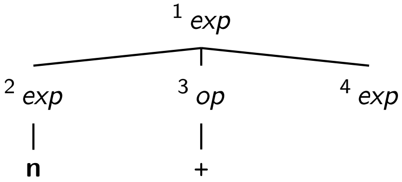



- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3] There will be *abstract* syntax trees, as well.

# Parse tree

- derivation: if viewed as sequence of steps $\Rightarrow$ linear “structure”
- order of individual steps: irrelevant
- $\Rightarrow$ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]

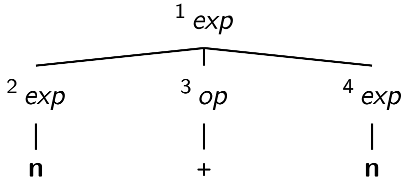



- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3]There will be *abstract* syntax trees, as well.

# Parse tree

- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]



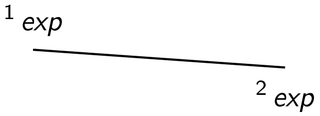

- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

[3] There will be *abstract* syntax trees, as well.

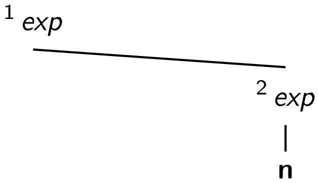
# Another parse tree (numbers for rightmost derivation)

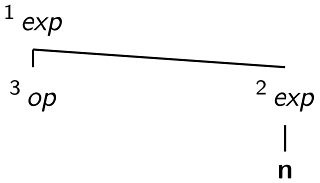
$^{1}\,exp$

# Another parse tree (numbers for rightmost derivation)



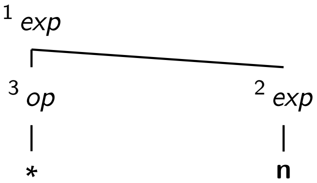
# Another parse tree (numbers for rightmost derivation)



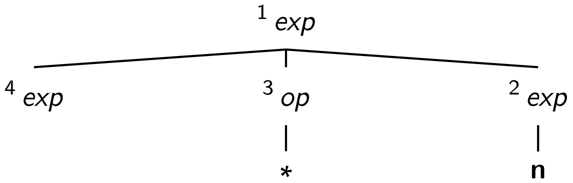


1 exp
3 op
2 exp
n

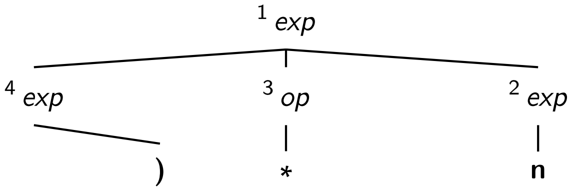
# Another parse tree (numbers for rightmost derivation)



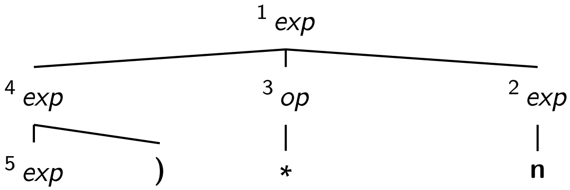
# Another parse tree (numbers for rightmost derivation)

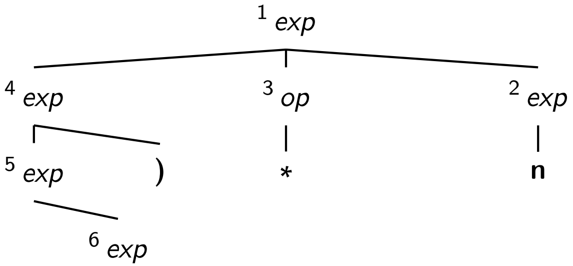


# Another parse tree (numbers for rightmost derivation)

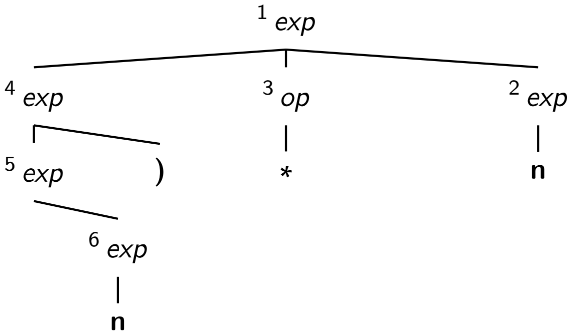


# Another parse tree (numbers for rightmost derivation)

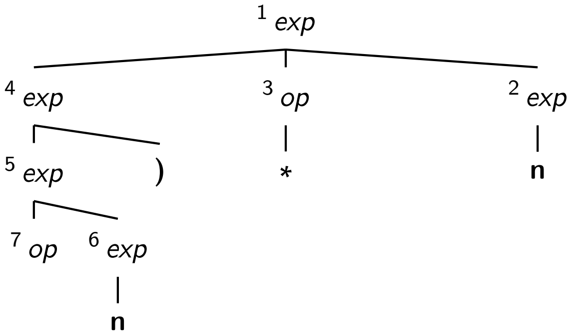



1 exp
4 exp
3 op
2 exp
5 exp
)
*
n
6 exp

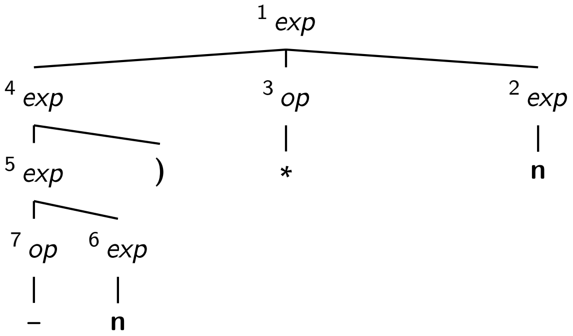
# Another parse tree (numbers for rightmost derivation)



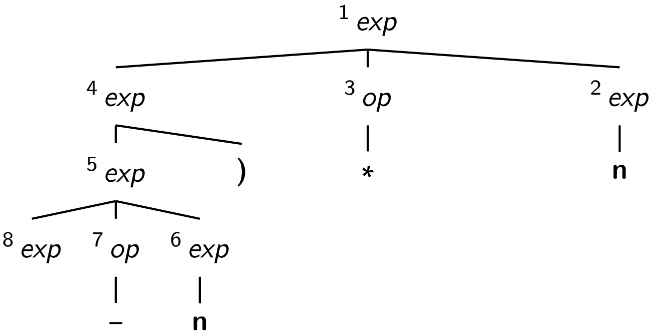
# Another parse tree (numbers for rightmost derivation)



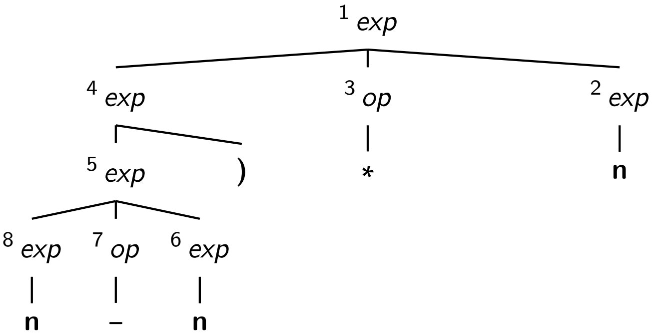
# Another parse tree (numbers for rightmost derivation)



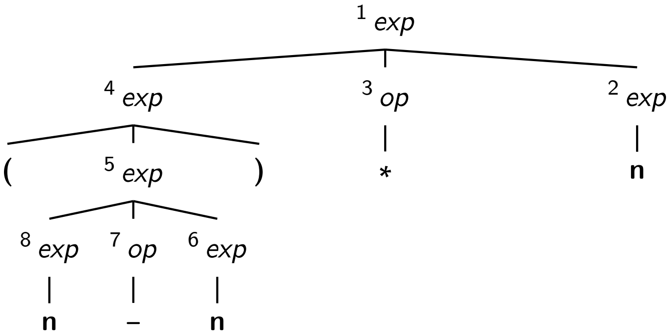
# Another parse tree (numbers for rightmost derivation)



# Another parse tree (numbers for rightmost derivation)

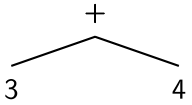


# Another parse tree (numbers for rightmost derivation)

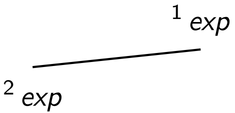


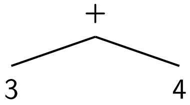
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

$^{1}$ *exp*



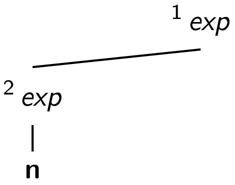
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

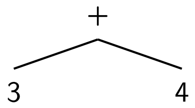


# Abstract syntax tree

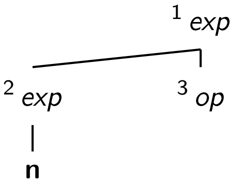
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

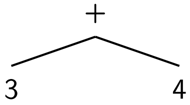


# Abstract syntax tree

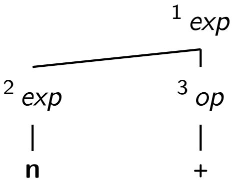
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" . . . )

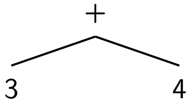


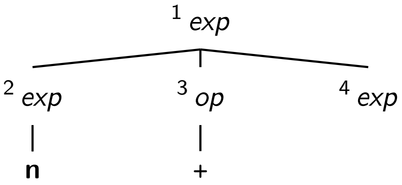
# Abstract syntax tree

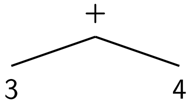
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)



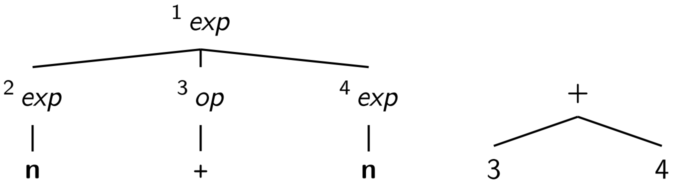
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)



# Abstract syntax tree

- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

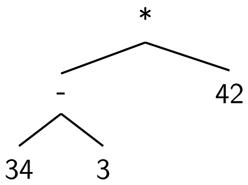


# AST vs. CST

- parse tree
  - important *conceptual* structure, to talk about grammars and derivations. . . ,
  - most likely *not explicitly implemented* in a parser
- AST is a *concrete* data structure
  - important IR of the syntax of the language being implemented
  - written in the meta-language used in the implementation
  - therefore: nodes like + and 3 *are no longer tokens or lexemes*
  - concrete data stuctures in the meta-language (C-structs, instances of Java classes, or what suits best)
  - the figure is meant schematic, only
  - produced by the parser, used by later phases
  - note also: we use 3 in the AST, where lexeme was "3"

  ⇒ at some point the lexeme string (for numbers) is translated to a *number* in the meta-language (typically already by the lexer)

# Plausible schematic AST (for the other parse tree)



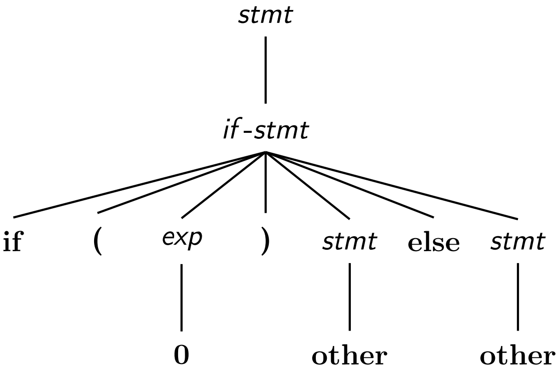

- this AST: rather "simplified" version of the CST
- an AST closer to the CST (just dropping the parentheses): in principle nothing "wrong" with it either

## Conditionals $G_1$

$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & \mathit{if\text{-}stmt} \mid \textbf{other} \\ \mathit{if\text{-}stmt} & \rightarrow & \textbf{if} \; ( \; \mathit{exp} \; ) \; \mathit{stmt} \\ & \mid & \textbf{if} \; ( \; \mathit{exp} \; ) \; \mathit{stmt} \; \textbf{else} \; \mathit{stmt} \\ \mathit{exp} & \rightarrow & \mathbf{0} \mid \mathbf{1} \end{array} \tag{5}$$

# Parse tree

**if ( 0 ) other else other**



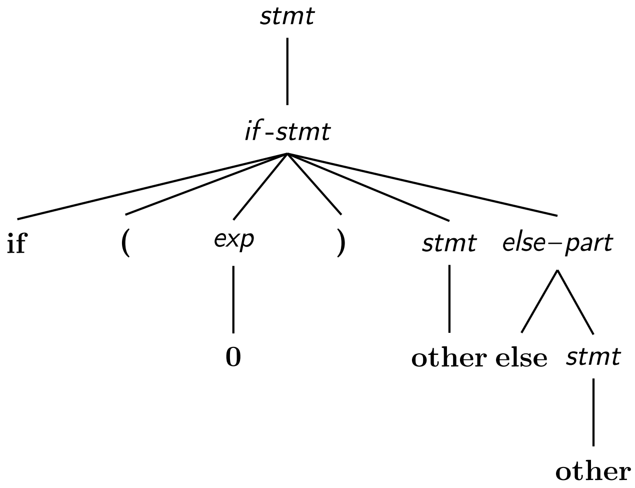
# Another grammar for conditionals

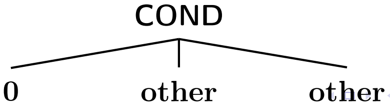
## Conditionals $G_2$

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \mathit{if}\text{-}\mathit{stmt} \mid \mathbf{other} \\
\mathit{if}\text{-}\mathit{stmt} &\rightarrow \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt}\ \mathit{else{-}part} \\
\mathit{else{-}part} &\rightarrow \mathbf{else}\ \mathit{stmt} \mid \epsilon \\
\mathit{exp} &\rightarrow \mathbf{0} \mid \mathbf{1}
\end{aligned}
\tag{6}
$$

$\epsilon$ = empty word

# A further parse tree + an AST



# Outline

# Tempus fugit . . .

picture source: wikipedia

# Ambiguous grammar

## Definition (Ambiguous grammar)

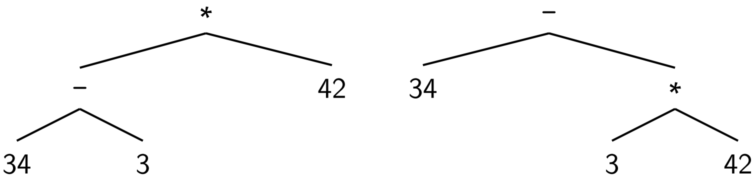
A grammar is *ambiguous* if there exists a word with *two different* parse trees.

Remember grammar from equation (1):

$$\begin{array}{rcl} \mathit{exp} & \rightarrow & \mathit{exp}\ \ \mathit{op}\ \ \mathit{exp} \mid (\,\mathit{exp}\,) \mid \textbf{number} \\ \mathit{op} & \rightarrow & + \mid - \mid * \end{array}$$

Consider:

$$\mathbf{n} - \mathbf{n} * \mathbf{n}$$

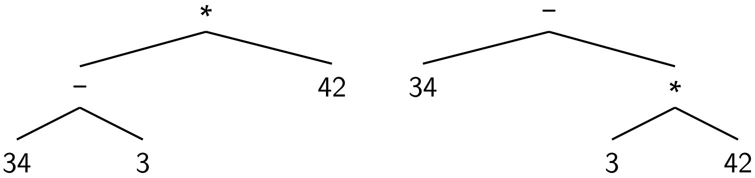



different parse trees ⇒ different[4] ASTs ⇒ different[4] meaning

### Side remark: different meaning

The issue of "different meaning" may in practice be subtle: is $(x + y) - z$ the same as $x + (y - z)$?

[4] At least in many cases.




different parse trees $\Rightarrow$ different[4] ASTs $\Rightarrow$ different[4] meaning

### Side remark: different meaning

The issue of "different meaning" may in practice be subtle: is $(x + y) - z$ the same as $x + (y - z)$? In principle yes, but what about MAXINT ?

[4] At least in many cases.

# Precendence & associativity

- one way to make a grammar unambiguous (or less ambiguous)
- for instance:

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

- $a \uparrow b$ written in standard math as $a^b$:

$$
\begin{aligned}
& 5 + 3/5 \times 2 + 4 \uparrow 2 \uparrow 3 && = \\
& 5 + 3/5 \times 2 + 4^{2^3} && = \\
& (5 + ((3/5 \times 2)) + (4^{(2^3)})) \ .
\end{aligned}
$$

- mostly fine for *binary* ops, but usually also for unary ones (postfix or prefix)

# Unambiguity without associativity and precedence

- removing ambiguity by reformulating the grammar
- precedence for op's: *precedence cascade*
  - some bind stronger than others (* more than +)
  - introduce separate *non-terminal* for each precedence level (here: terms and factors)

- *associativity*
  - *left*-assoc: write the corresponding rules in *left-recursive* manner, e.g.:

$$exp \rightarrow exp\ addop\ term \mid term$$

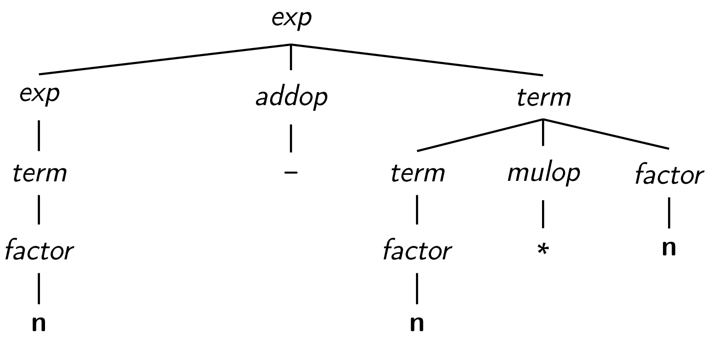
  - *right*-assoc: analogous, but right-recursive
  - *non*-assoc:

$$exp \rightarrow term\ addop\ term \mid term$$

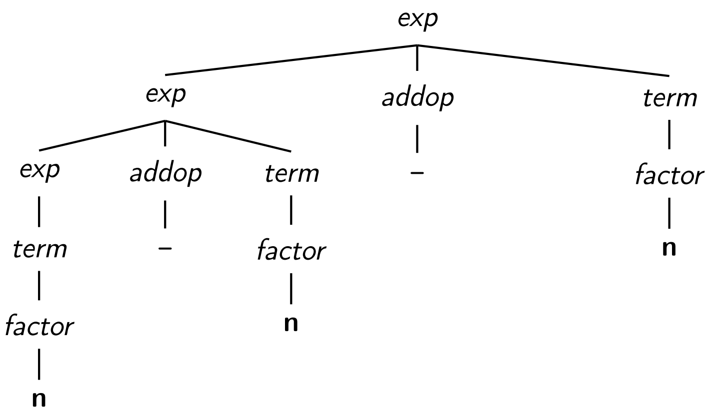
## factors and terms

$$
\begin{array}{rcl}
exp & \rightarrow & exp\ \ addop\ \ term \mid term \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & term\ \ mulop\ \ factor \mid factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{7}
$$

# $34 - 3 * 42$



# 34 – 3 – 42


exp
exp
addop
term
exp
addop
term
–
factor
term
–
factor
n
factor
n
n

# Real life example

left associative

**Operator Precedence**

Java performs operations assuming the following ordering (or ***precedence***) rules if parentheses are not used to determine the order of evaluation (operators on the same line are evaluated in left-to-right order subject to the conditional evaluation rule for && and ||). The operations are listed below from highest to lowest precedence (we use ⟨exp⟩ to denote an atomic or parenthesized expression):

<table>
<tr><td>postfix ops</td><td>[] . (⟨exp⟩) ⟨exp⟩++ ⟨exp⟩−−</td></tr>
<tr><td>prefix ops</td><td>++⟨exp⟩ −−⟨exp⟩ −⟨exp⟩ ˜⟨exp⟩ !⟨exp⟩</td></tr>
<tr><td>creation/cast</td><td>new (⟨type⟩)⟨exp⟩</td></tr>
<tr><td>mult./div.</td><td>* / %</td></tr>
<tr><td>add./subt.</td><td>+ −</td></tr>
<tr><td>shift</td><td><< >> >>></td></tr>
<tr><td>comparison</td><td>< <= > >= instanceof</td></tr>
<tr><td>equality</td><td>== !=</td></tr>
<tr><td>bitwise-and</td><td>&</td></tr>
<tr><td>bitwise-xor</td><td>^</td></tr>
<tr><td>bitwise-or</td><td>|</td></tr>
<tr><td>and</td><td>&&</td></tr>
<tr><td>or</td><td>||</td></tr>
<tr><td>conditional</td><td>⟨bool_exp⟩? ⟨true_val⟩: ⟨false_val⟩</td></tr>
<tr><td>assignment</td><td>=</td></tr>
<tr><td>op assignment</td><td>+= −= *= /= %=</td></tr>
<tr><td>bitwise assign.</td><td>>>= <<= >>>=</td></tr>
<tr><td>boolean assign.</td><td>&= ^= |=</td></tr>
</table>

# Another example

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.

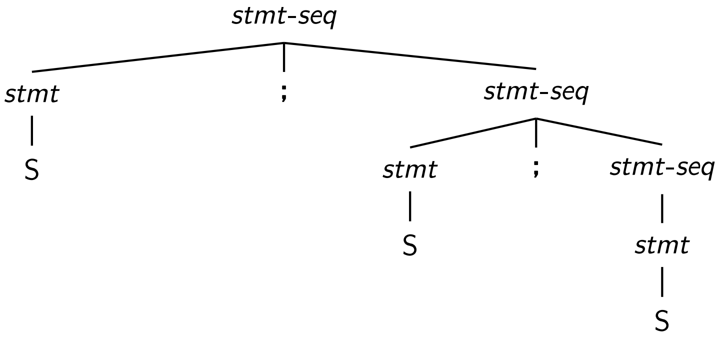
Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity 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

# Non-essential ambiguity

## left-assoc

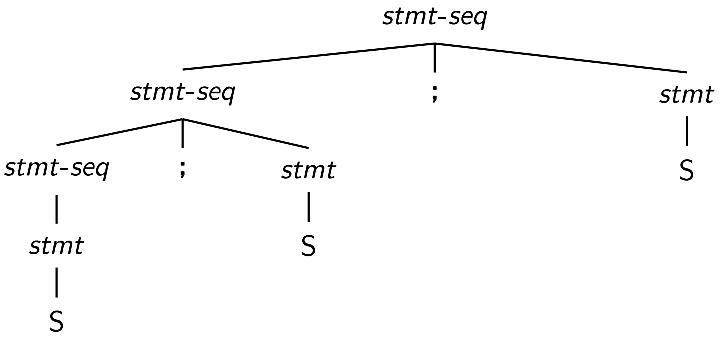
$$
\begin{aligned}
\mathit{stmt\text{-}seq} &\rightarrow \mathit{stmt\text{-}seq}\ \mathbf{;}\ \mathit{stmt} \mid \mathit{stmt} \\
\mathit{stmt} &\rightarrow S
\end{aligned}
$$



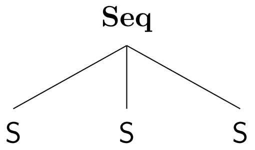
# Non-essential ambiguity (2)

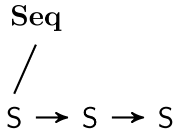
## right-assoc representation instead

$$
\begin{array}{rcl}
\mathit{stmt\text{-}seq} & \rightarrow & \mathit{stmt}\ ;\ \mathit{stmt\text{-}seq} \mid \mathit{stmt} \\
\mathit{stmt} & \rightarrow & S
\end{array}
$$



# Possible AST representations



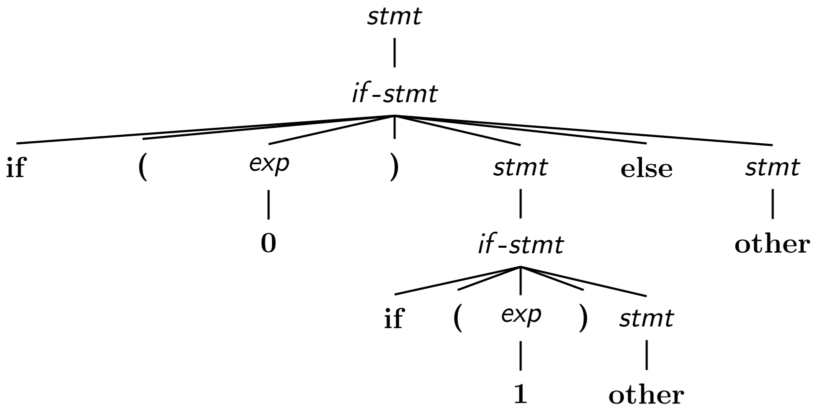
## Nested if's

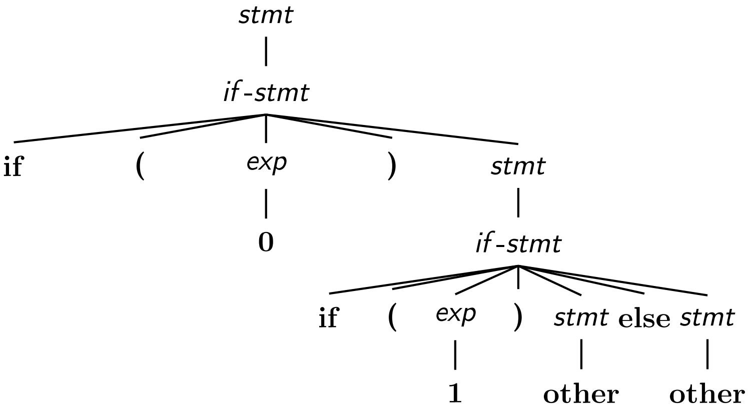
$$\textbf{if}\,(\,\mathbf{0}\,)\,\textbf{if}\,(\,\mathbf{1}\,)\,\textbf{other else other}$$

Remember grammar from equation (5):

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

## Should it be like this . . .





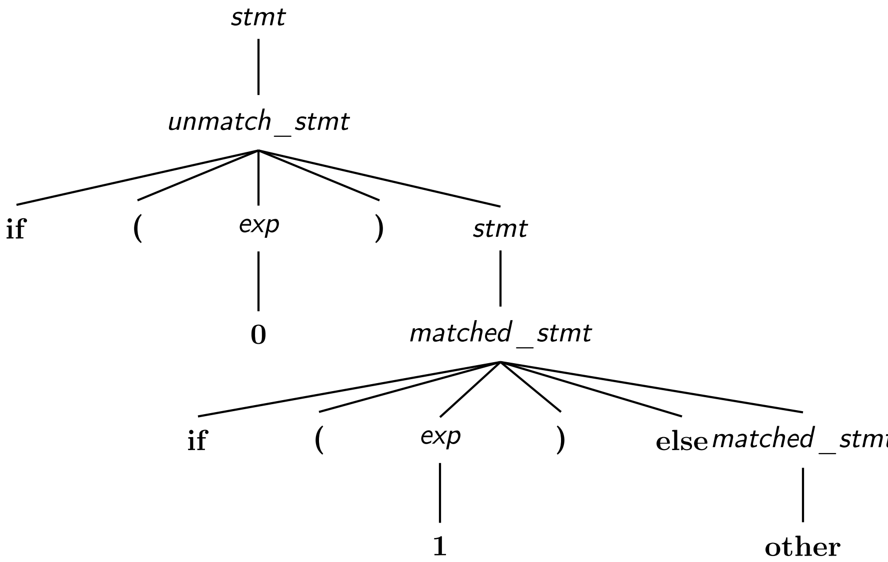

- common convention: connect **else** to closest "free" (= dangling) occurrence

# Unambiguous grammar

## Grammar

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \mid \mathit{unmatch\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{matched\_stmt}\ \mathbf{else}\ \mathit{matched\_stmt} \\
 & \mid & \mathbf{other} \\
\mathit{unmatch\_stmt} & \rightarrow & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt} \\
 & \mid & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{matched\_stmt}\ \mathbf{else}\ \mathit{unmatch\_stmt} \\
\mathit{exp} & \rightarrow & \mathbf{0} \mid \mathbf{1}
\end{array}
$$

- never have an unmatched statement inside a matched
- complex grammar, seldomly used
- instead: ambiguous one, with extra "rule": connect each **else** to closest free **if**
- alternative: *different* syntax, e.g.,
  - *mandatory* **else**,
  - or require **endif**


stmt
unmatch_stmt
if
(
exp
)
stmt
0
matched_stmt
if
(
exp
)
else
matched_stmt
1
other

# Adding sugar: extended BNF

- make CFG-notation more "convenient" (but without more theoretical expressiveness)
- syntactic sugar

## EBNF

Main additional notational freedom: use regular expressions on the rhs of productions. They can contain terminals and non-terminals

- EBNF: officially standardized, but often: all "sugared" BNFs are called EBNF
- in the standard:
  - $\alpha^*$ written as $\{\alpha\}$
  - $\alpha?$ written as $[\alpha]$
- supported (in the standardized form or other) by some parser tools, but not in all
- remember equation (2)

# EBNF examples

$$A \rightarrow \beta\{\alpha\} \qquad \text{for} \quad A \rightarrow A\alpha \mid \beta$$

$$A \rightarrow \{\alpha\}\beta \qquad \text{for} \quad A \rightarrow \alpha A \mid \beta$$

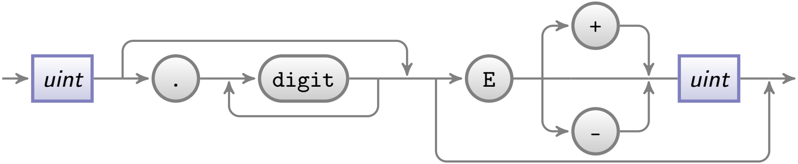
$$\begin{aligned}
\textit{stmt-seq} &\rightarrow \textit{stmt} \; \{\mathbf{;}\, \textit{stmt}\} \\
\textit{stmt-seq} &\rightarrow \{\textit{stmt}\,\mathbf{;}\} \; \textit{stmt} \\
\textit{if-stmt} &\rightarrow \mathbf{if}\ \mathbf{(}\ \textit{exp}\ \mathbf{)}\ \textit{stmt}[\mathrm{else}\ \textit{stmt}]
\end{aligned}$$

greek letters: for non-terminals or terminals.

# Outline

# Syntax diagrams

- graphical notation for CFG
- used for Pascal
- important concepts like ambiguity etc: not easily recognizable
  - not much in use any longer
  - example for floats, using unsigned int's (taken from the TikZ manual):



# Outline

# The Chomsky hierarchy

- linguist Noam Chomsky [?]
- important classification of (formal) languages (sometimes Chomsky-Schützenberger)
- 4 levels: type 0 languages – type 3 languages
- levels related to machine models that generate/recognize them
- so far: regular languages and CF languages

# Overview

| | 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 |

## Conventions

- terminals $a, b, \ldots \in \Sigma_T$,
- non-terminals $A, B, \ldots \in \Sigma_N$
- general words $\alpha, \beta \ldots \in (\Sigma_T \cup \Sigma_N)^*$

## "Simplified" design?

1 big grammar for the whole compiler? Or at least a CSG for the front-end, or a CFG combining parsing and scanning?

theoretically possible, but bad idea:

- efficiency
- bad design
- especially combining scanner + parser in one BNF:
  - grammar would be needlessly large
  - separation of concerns: much clearer/ more efficient design
- for scanner/parsers: regular expressions + (E)BNF: simply the formalisms of choice!
  - front-end needs to do more than checking syntax, CFGs not expressive enough
  - for level-2 and higher: situation gets less clear-cut, plain CSG not too useful for compilers

# Outline

# BNF-grammar for TINY

$$
\begin{array}{rcl}
\textit{program} & \to & \textit{stmt-seq} \\
\textit{stmt-seq} & \to & \textit{stmt-seq}\,\mathbf{;}\,\textit{stmt} \mid \textit{stmt} \\
\textit{stmt} & \to & \textit{if-stmt} \mid \textit{repeat-stmt} \mid \textit{assign-stmt} \\
 & \mid & \textit{read-stmt} \mid \textit{write-stmt} \\
\textit{if-stmt} & \to & \mathbf{if}\ \textit{expr}\ \mathbf{then}\ \textit{stmt}\ \mathbf{end} \\
 & \mid & \mathbf{if}\ \textit{expr}\ \mathbf{then}\ \textit{stmt}\ \mathbf{else}\ \textit{stmt}\ \mathbf{end} \\
\textit{repeat-stmt} & \to & \mathbf{repeat}\ \textit{stmt-seq}\ \mathbf{until}\ \textit{expr} \\
\textit{assign-stmt} & \to & \mathbf{identifier}\ \mathbf{:=}\ \textit{expr} \\
\textit{read-stmt} & \to & \mathbf{read}\ \mathbf{identifier} \\
\textit{write-stmt} & \to & \mathbf{write}\ \textit{expr} \\
\textit{expr} & \to & \textit{simple-expr}\ \textit{comparison-op}\ \textit{simple-expr} \mid \textit{simple-expr} \\
\textit{comparison-op} & \to & < \mid = \\
\textit{simple-expr} & \to & \textit{simple-expr}\ \textit{addop}\ \textit{term} \mid \textit{term} \\
\textit{addop} & \to & + \mid - \\
\textit{term} & \to & \textit{term}\ \textit{mulop}\ \textit{factor} \mid \textit{factor} \\
\textit{mulop} & \to & * \mid / \\
\textit{factor} & \to & (\ \textit{expr}\ ) \mid \mathbf{number} \mid \mathbf{identifier}
\end{array}
$$

# Syntax tree nodes

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type; /* for type checking of exps */
```

# Comments on C-representation

- typical use of `enum` type for that (in C)
- `enum`'s in C can be very efficient
- `treeNode` struct (records) is a bit "unstructured"
- newer languages/higher-level than C: better structuring advisable, especially for languages larger than Tiny.
- in Java-kind of languages: inheritance/subtyping and abstract classes/interfaces often used for better structuring

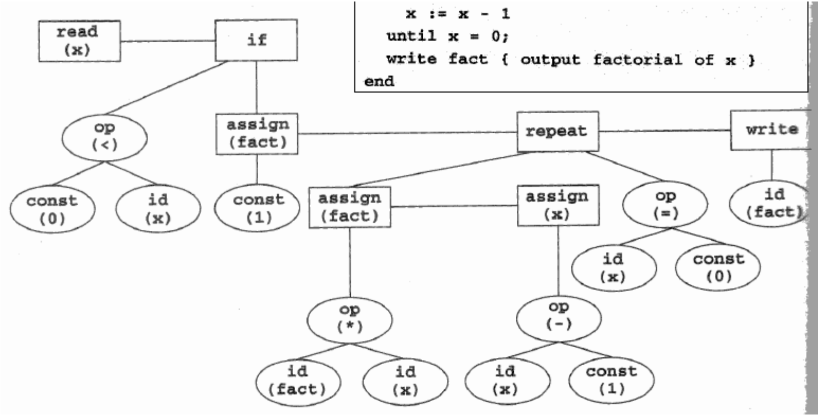
# Sample Tiny program

```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact   { output factorial of x }
end
```

# Same Tiny program again

```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact   { output factorial of x }
end
```

- *keywords* / *reserved words* highlighted by bold-face type setting
- reserved syntax like 0, :=, ... is not bold-faced
- comments are italicized


x := x - 1
until x = 0;
write fact { output factorial of x }
end
read
(x)
if
op
(<)
assign
(fact)
repeat
write
const
(0)
id
(x)
const
(1)
assign
(fact)
assign
(x)
op
(=)
id
(fact)
id
(x)
const
(0)
op
(*)
op
(-)
id
(fact)
id
(x)
id
(x)
const
(1)

later given as assignment

- is the grammar unambiguous?
- How can we change it so that the Tiny allows empty statements?
- What if we want semicolons *in between* statements and not *after*?
- What is the precedence and associativity of the different operators?

# Outline

# References I


[Appel, 1998] Appel, A. W. (1998).
*Modern Compiler Implementation in ML/Java/C.*
Cambridge University Press.

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