Chapter 3

Grammars

Course "Compiler Construction"
Martin Steffen
Spring 2021

Chapter 3

Learning Targets of Chapter “Grammars”.

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 [2, Section 3.1–3.2] (or [3, Chapter 3]).

Chapter 3

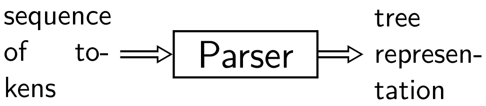
Outline of Chapter “Grammars”.

Section

Introduction

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

Bird's eye view of a parser



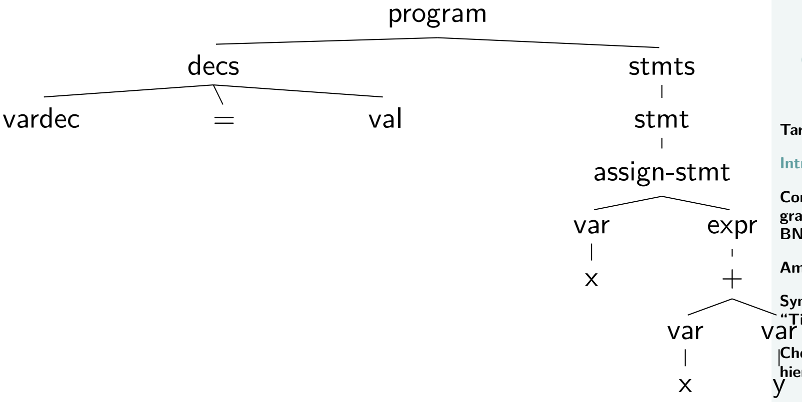
- *check* that the token sequence correspond to a *syntactically correct* program
  - if yes: yield *tree* as intermediate representation for subsequent phases
  - if not: give *understandable* error message(s)
- we will encounter various kinds of trees
  - derivation trees (derivation in a (context-free) 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

(Context-free) grammars

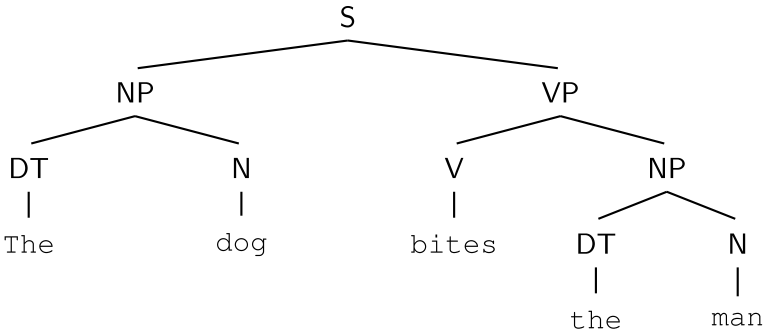
- specifies the *syntactic structure* of a language
- here: grammar means CFG
- G derives word w

Parsing

Given a stream of "symbols" w and a grammar G, find a *derivation* from G that produces w.


program
decs
stmts
vardec
=
val
stmt
assign-stmt
var
expr
x
+
var
var
x
y

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 $\langle \texttt{integer}, \text{"42"} \rangle$
  - `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)

Section

Context-free grammars and BNF notation

Chapter 3 "Grammars"
Course "Compiler Construction"
Martin Steffen
Spring 2021

Grammars

- 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. [1]

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

Context-free grammar

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 $\Sigma_N \times (\Sigma_N + \Sigma_T)^*$

- 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

Further notions

- sentence and sentential form
- productions (or rules)
- derivation
- *language* of a grammar $\mathcal{L}(G)$
- parse tree

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

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

- "$\to$" indicating productions and " $\mid$ " indicating alternatives
- 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.

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

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

- 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":

$$
\begin{aligned}
exp &\rightarrow exp\ op\ exp \\
exp &\rightarrow (\ exp\ ) \\
exp &\rightarrow \mathbf{number} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow *
\end{aligned} \tag{3}
$$

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

$$
\begin{aligned}
E &\rightarrow E\ O\ E \mid (\ E\ ) \mid \mathbf{number} \\
O &\rightarrow + \mid - \mid *
\end{aligned} \tag{4}
$$

- 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 $(\mathbf{number}-\mathbf{number})*\mathbf{number}$

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

- underline the "place" where 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

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

- 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)

$$\begin{array}{rcl} A & \rightarrow & B\mathbf{x} \\ B & \rightarrow & A\mathbf{y} \\ C & \rightarrow & \mathbf{z} \end{array}$$

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

Parse tree

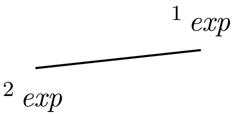
- derivation: if viewed as sequence of steps $\Rightarrow$ linear "structure"
- order of individual steps: irrelevant
- $\Rightarrow$ order not needed for subsequent phases
- parse tree: structure for the *essence* of derivation
- also called concrete syntax tree.

$$^{1}\ exp$$

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

Parse tree

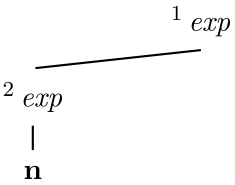
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent phases
- parse tree: structure for the *essence* of derivation
- also called concrete syntax tree.



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

Parse tree

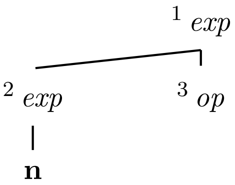
- derivation: if viewed as sequence of steps $\Rightarrow$ linear "structure"
- order of individual steps: irrelevant
- $\Rightarrow$ order not needed for subsequent phases
- parse tree: structure for the *essence* of derivation
- also called concrete syntax tree.



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

Parse tree

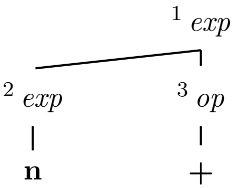
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent phases
- parse tree: structure for the *essence* of derivation
- also called concrete syntax tree.



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

Parse tree

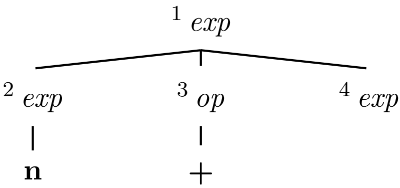
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent phases
- parse tree: structure for the *essence* of derivation
- also called concrete syntax tree.



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

Parse tree

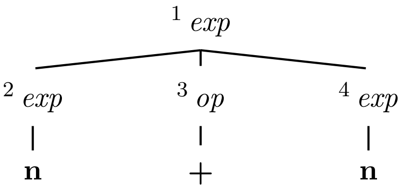
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent phases
- parse tree: structure for the *essence* of derivation
- also called concrete syntax tree.



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

Parse tree

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

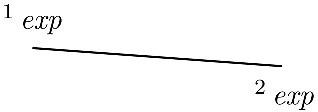


- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

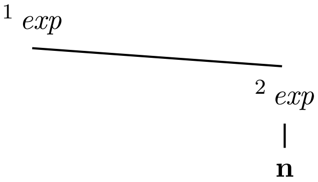
Another parse tree (numbers for rightmost derivation)

$^{1}\ exp$

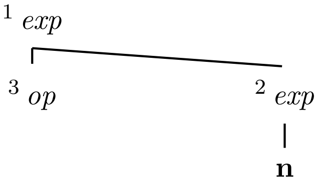
Another parse tree (numbers for rightmost derivation)



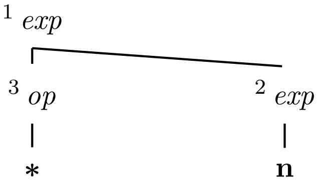
Another parse tree (numbers for rightmost derivation)



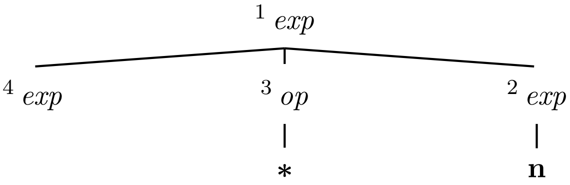
Another parse tree (numbers for rightmost derivation)



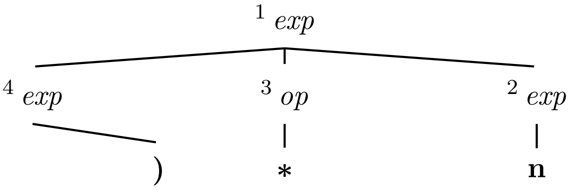
Another parse tree (numbers for rightmost derivation)



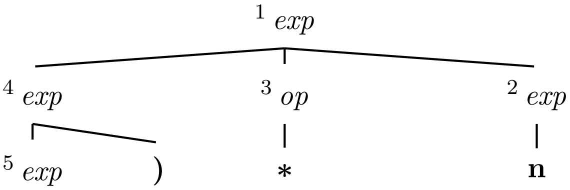
Another parse tree (numbers for rightmost derivation)



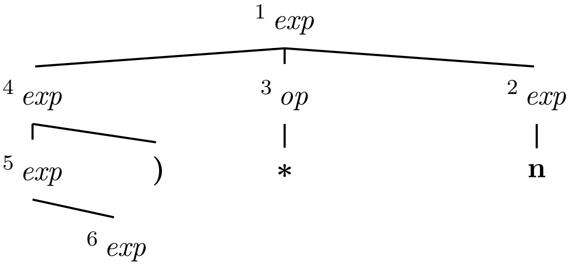
Another parse tree (numbers for rightmost derivation)



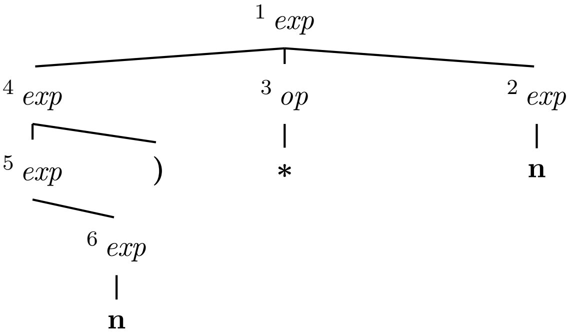
Another parse tree (numbers for rightmost derivation)



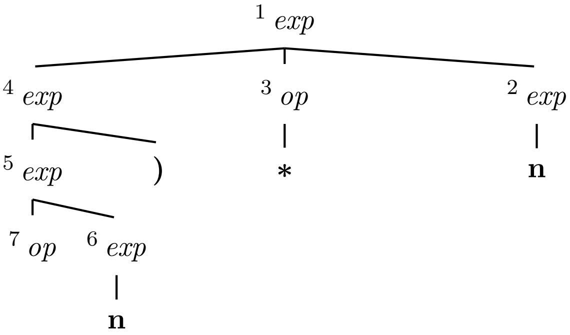
Another parse tree (numbers for rightmost derivation)



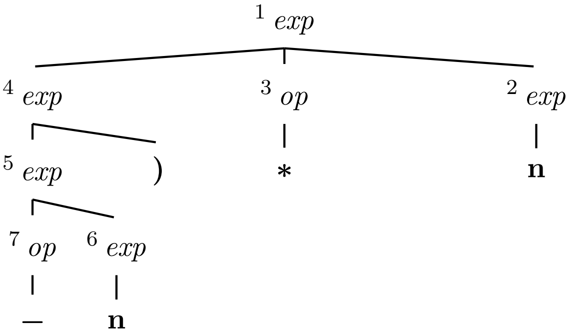
Another parse tree (numbers for rightmost derivation)



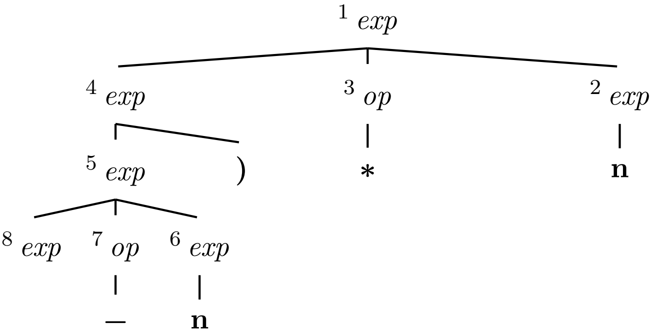
Another parse tree (numbers for rightmost derivation)



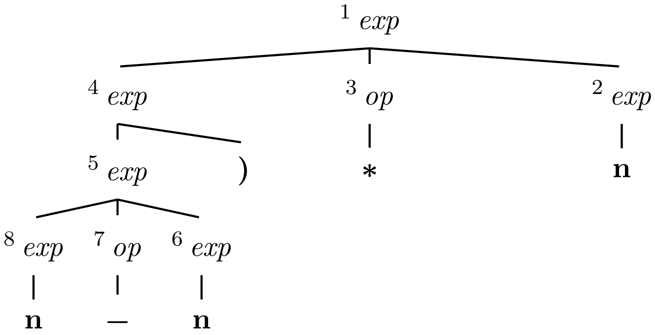
Another parse tree (numbers for rightmost derivation)



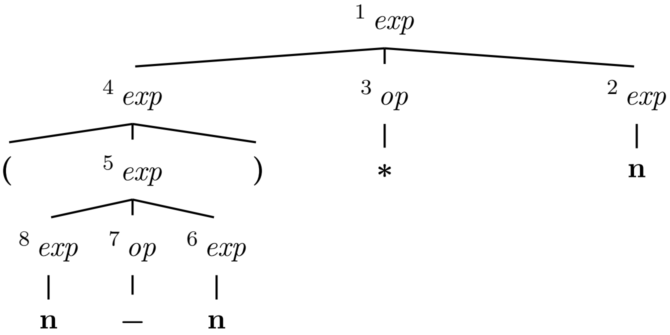
Another parse tree (numbers for rightmost derivation)



Another parse tree (numbers for rightmost derivation)



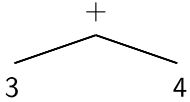
Another parse tree (numbers for rightmost derivation)



Abstract syntax tree

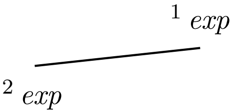
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

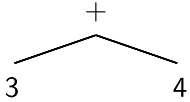
[1] *exp*



Abstract syntax tree

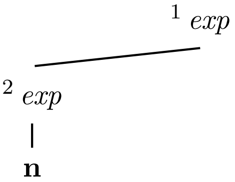
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class $\mathbf{n}$ may contain lexeme like "42" ...)

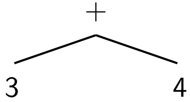




Abstract syntax tree

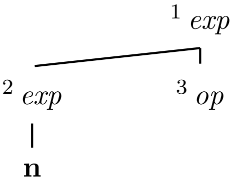
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

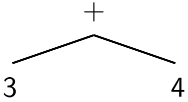




Abstract syntax tree

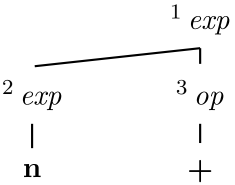
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

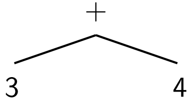




Abstract syntax tree

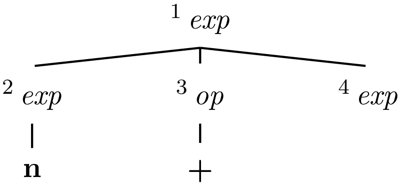
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class $\mathbf{n}$ may contain lexeme like "42" . . .)

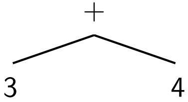




Abstract syntax tree

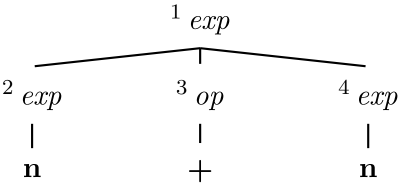
- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)

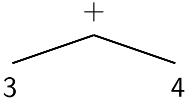




Abstract syntax tree

- 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 $\mathbf{n}$ may contain lexeme like "42" ...)



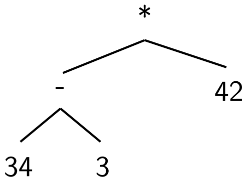


AST vs. CST

- 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 (for the language being implemented)
  - written in the meta-language
  - 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

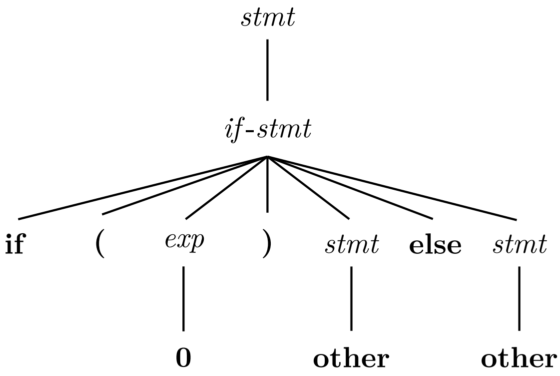
Conditionals

Conditionals G_1

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

Parse tree

if (0) other else other



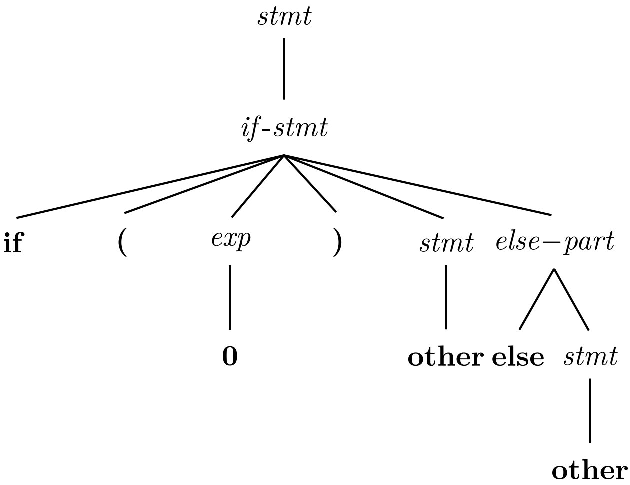
Another grammar for conditionals

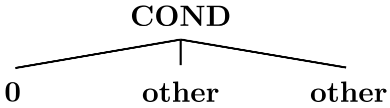
Conditionals G_2

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

ϵ = empty word

A further parse tree + an AST





Section

Ambiguity

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

Tempus fugit . . .

picture source: wikipedia

Ambiguous grammar

Definition (Ambiguous grammar)

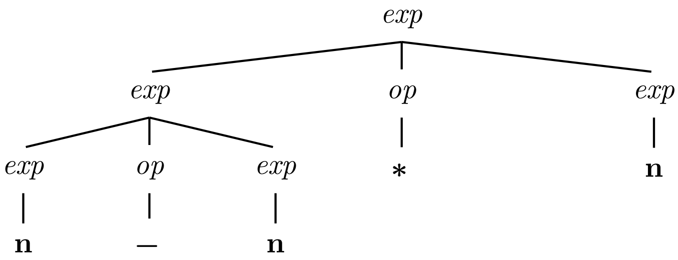
A grammar is *ambiguous* if there exists a word with *two different* parse trees.

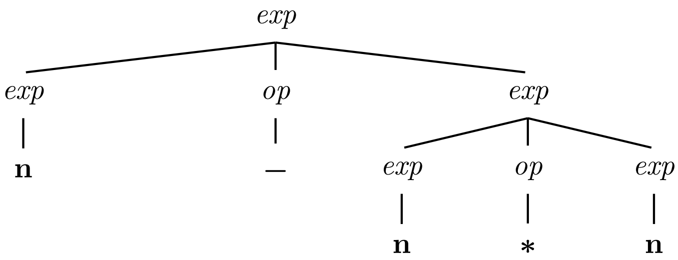
Remember grammar from equation (1):

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

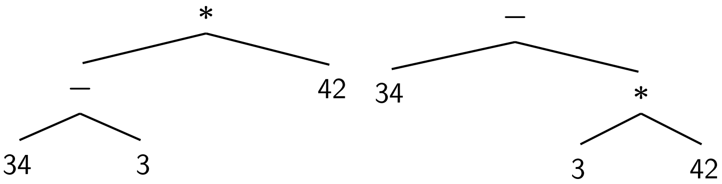
Consider:

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


exp
exp
op
exp
exp
op
exp
*
n
n
–
n


exp
exp
op
exp
n
–
exp
op
exp
n
*
n

2 resulting ASTs

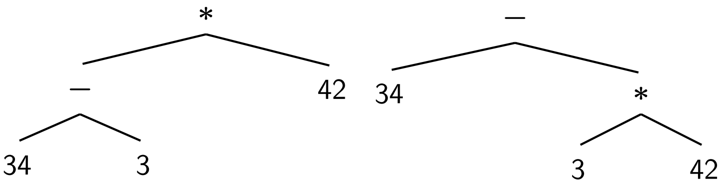


different parse trees ⇒ different ASTs ⇒ different meaning

Side remark: different meaning

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

2 resulting ASTs



different parse trees ⇒ different ASTs ⇒ different 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 ?

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:

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

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

Unambiguity without imposing explicit 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)

Expressions, revisited

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

$$exp \to exp\ addop\ term \mid term$$

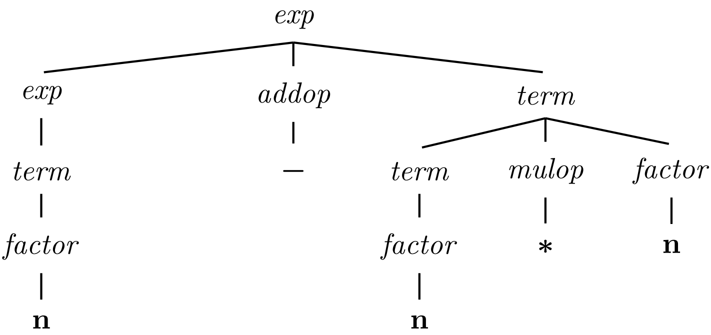
  - *right*-assoc: analogous, but right-recursive
  - *non*-assoc:

$$exp \to term\ addop\ term \mid term$$

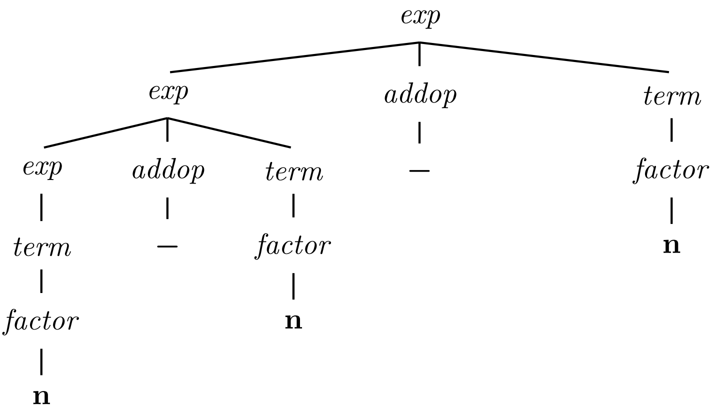
factors and terms

$$\begin{aligned} exp &\to exp\ \ addop\ \ term \mid term \qquad (7)\\ addop &\to + \mid - \\ term &\to term\ \ mulop\ \ factor \mid factor \\ mulop &\to * \\ factor &\to (\ exp\ ) \mid \mathbf{number} \end{aligned}$$

$34 - 3 * 42$



$34 - 3 - 42$


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):

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

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--`
	type`()` *type*`{}`
	`a()`
	`a[]`
	`. ->`
		Suffix/postfix increment and decrement
		Functional cast
		Function call
		Subscript
		Member access	

3	`++a --a`
	`+a -a`
	`! ~`
	`(`*type*`)`
	`*a`
	`&a`
	`sizeof`
	`new new[]`
	`delete delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style cast
		Indirection (dereference)
		Address-of
		Size-of[note 1]
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

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`
	`throw`
	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		Ternary conditional[note 2]
		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
			Right-to-left

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.

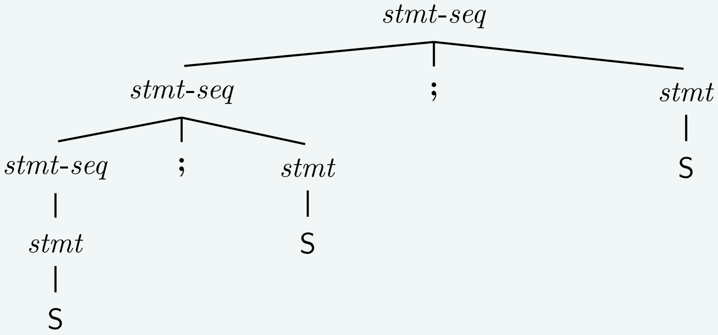
Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity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

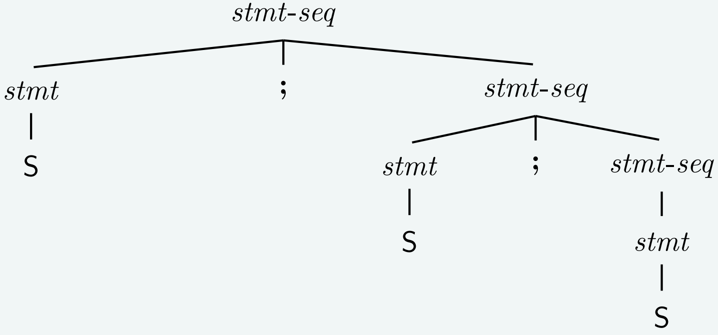
$$
\begin{aligned}
\textit{stmt-seq} &\rightarrow \textit{stmt-seq}\,;\textit{stmt} \mid \textit{stmt} \\
\textit{stmt} &\rightarrow S
\end{aligned}
$$



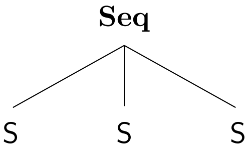
Non-essential ambiguity (2)

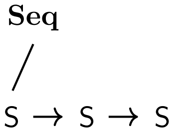
right-assoc representation instead

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



Possible AST representations





Dangling else

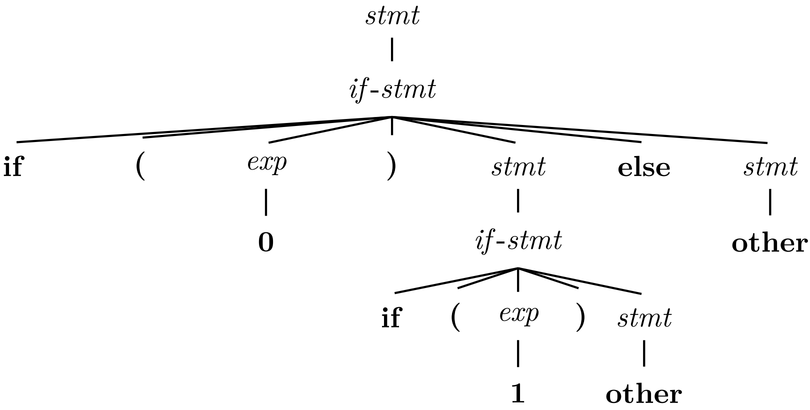
Nested if's

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

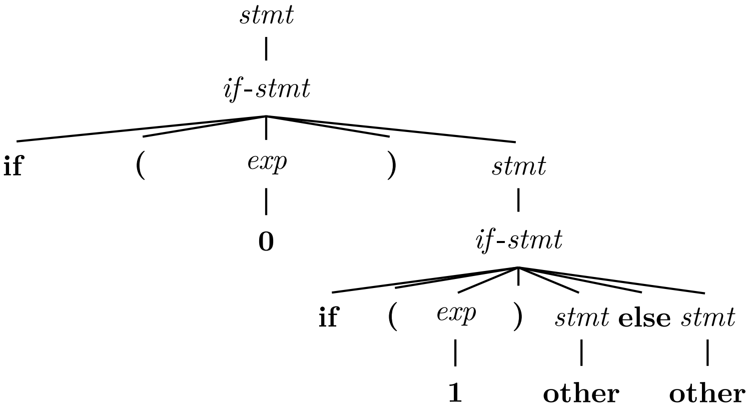
Remember grammar from equation (5):

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

Should it be like this . . .



. . . or like this



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

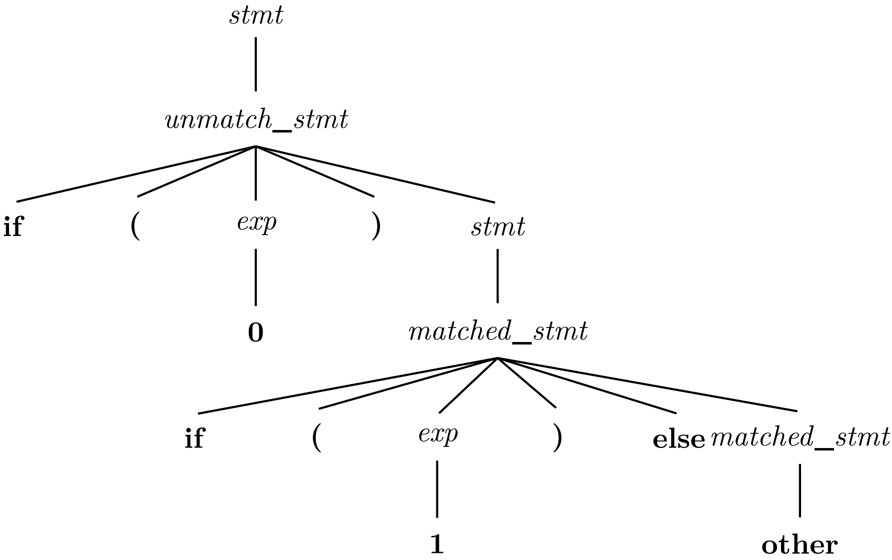
Unambiguous grammar

Grammar

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

- never have an unmatched statement inside a matched one
- 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**

CST



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:
  - α^* 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

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

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

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

greek letters: for non-terminals or terminals.

Some yacc style grammar

```
/* Infix notation calculator--calc */
%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
input:    /* empty string */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

Section

Syntax of a “Tiny” language

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

BNF-grammar for TINY

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

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

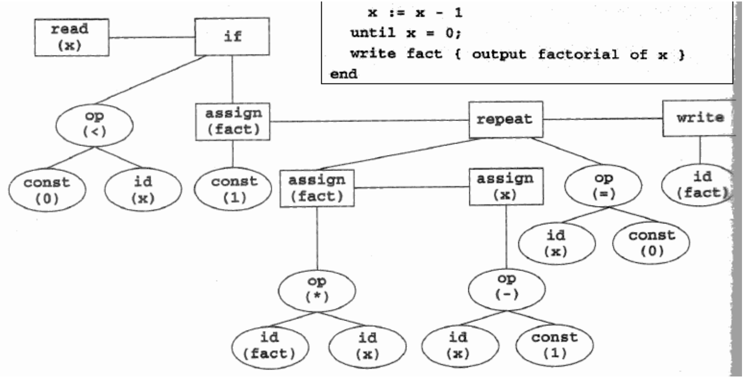
```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact   { output factorial of x }
end
```

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

Abstract syntax tree for a tiny program



Some questions about the Tiny grammy

- 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?

Section

Chomsky hierarchy

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

The Chomsky hierarchy

- linguist Noam Chomsky [1]
- 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 $\alpha, \beta \ldots \in (\Sigma_T \cup \Sigma_N)^*$

Phases of a compiler & hierarchy

"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

References I

Bibliography

[1] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(113–124).

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

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