INF5110 – Compiler Construction

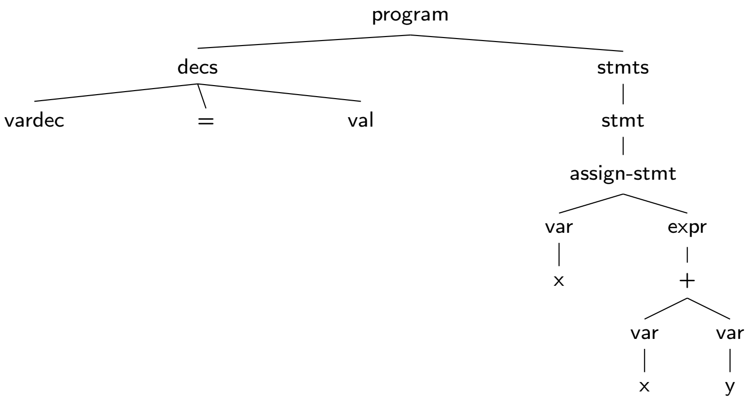
Grammars

Spring 2016

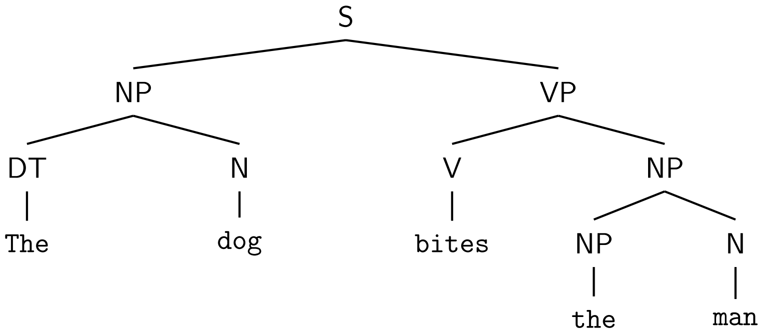
Bird eye's view of a parser

- *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
- 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, it's 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)

- 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 will reject some syntactically correct programs, whose violations *cannot* be captured by CFGs.

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

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

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

- "$\rightarrow$" 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: *expr*
- 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 $\rightarrow$.

- 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*

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

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

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

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

- still: we count 6 productions

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**

$$
\begin{array}{lcl}
\underline{\mathit{exp}} & \Rightarrow & \underline{\mathit{exp}}\ \mathit{op}\ \mathit{exp} \\
 & \Rightarrow & (\underline{\mathit{exp}})\ \mathit{op}\ \mathit{exp} \\
 & \Rightarrow & (\underline{\mathit{exp}}\ \mathit{op}\ \mathit{exp})\ \mathit{op}\ \mathit{exp} \\
 & \Rightarrow & (\textbf{number}\ \underline{\mathit{op}}\ \mathit{exp})\ \mathit{op}\ \mathit{exp} \\
 & \Rightarrow & (\textbf{number} - \underline{\mathit{exp}})\ \mathit{op}\ \mathit{exp} \\
 & \Rightarrow & (\textbf{number} - \textbf{number})\underline{\mathit{op}}\ \mathit{exp} \\
 & \Rightarrow & (\textbf{number} - \textbf{number}) * \underline{\mathit{exp}} \\
 & \Rightarrow & (\textbf{number} - \textbf{number}) * \textbf{number}
\end{array}
$$

- 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

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

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

Some easy requirements for reasonable grammars

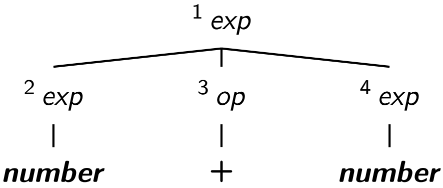
- all symbols (terminals and non-terminals): should occur in a word derivable from the start symbol
- words containing only non-terminals should be derivable
- an example of a silly grammar G (start-symbol A)

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

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

Parse tree

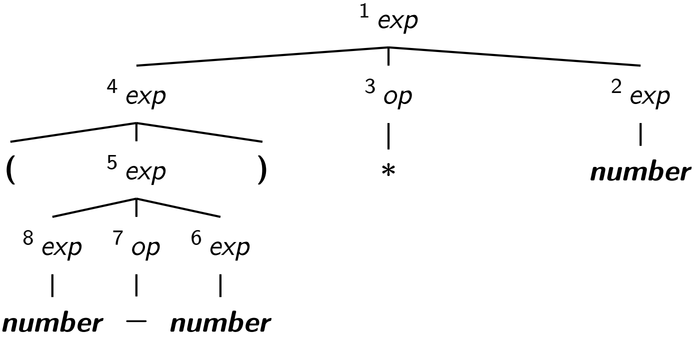
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- parse tree: structure for the *essence* of derivation
- also called *concrete* syntax tree.[3]



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

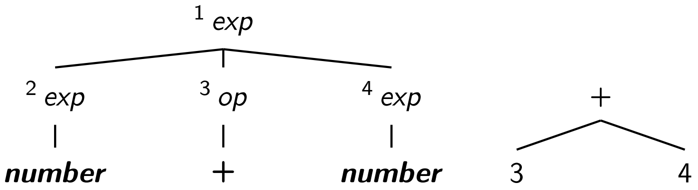
[3]there will be *abstract* syntax trees as well.

Another parse tree (numbers for rightmost derivation)



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 attibute values also (e.g.: full token for token class ***number*** may contain lexeme like "42" ...)

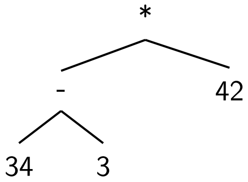


AST vs. CST

- parse tree
  - important *conceptual* structure, to talk about grammars . . . ,
  - most likely *not explicitly implemented* in a parser
- AST is a *concrete* datastructure
  - important IR of the syntax of the language to be 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 as schematic only
  - produced by the parser, used by later phases (often by more than one)
  - 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)



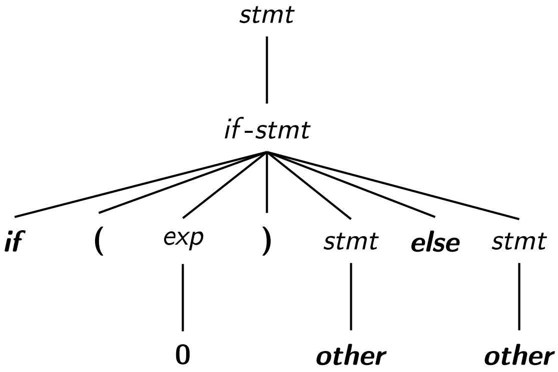
- this AST: rather "simplified" version of the CST
- an AST closer to the CST (just dropping the parentheses): under certain circumstances nothing wrong with it either.

Conditionals G_1

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

Parse tree

if* (0) *other else other



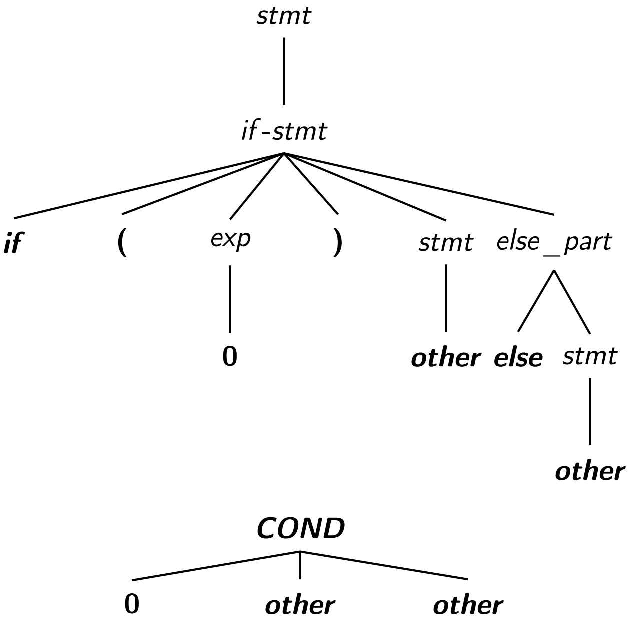
Another grammar for conditionals

Conditionals G_2

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

ϵ = empty word

A further parse tree + an AST



Definition (Ambiguous grammar)

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

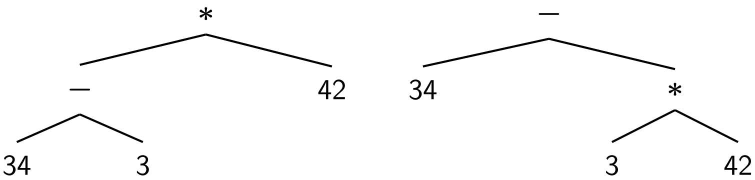
Remember grammar from equation (1):

$$\begin{array}{rcl} exp & \rightarrow & exp\ op\ exp \mid \textbf{(}\ exp\ \textbf{)} \mid \textbf{\textit{number}} \\ op & \rightarrow & \textbf{+} \mid \textbf{−} \mid \textbf{*} \end{array}$$

Consider:

$$\textbf{\textit{number}} - \textbf{\textit{number}} * \textbf{\textit{number}}$$

2 resulting ASTs



different parse trees ⇒ different[4] ASTs ⇒ different[5] 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 most 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)

- 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.:

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

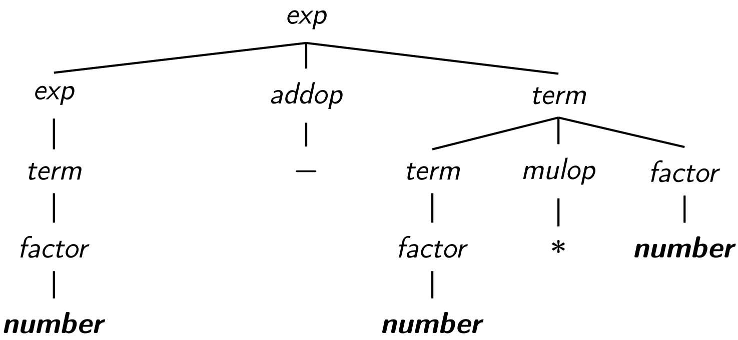
  - *right*-assoc: analogous, but right-recursive
  - *non*-assoc:

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

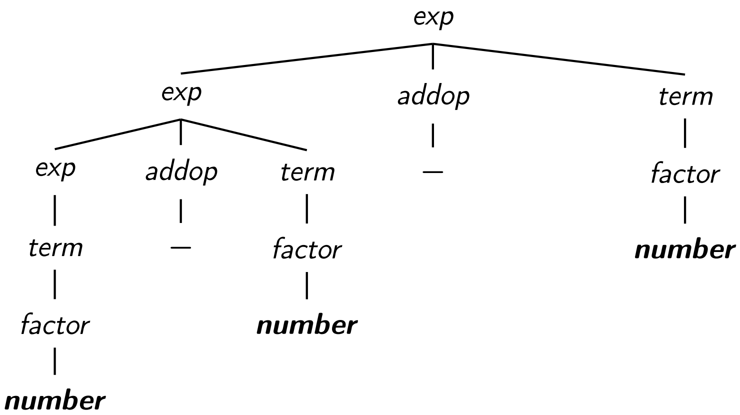
factors and terms

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

$34 - 3 * 42$


exp
exp
addop
term
term
−
term
mulop
factor
factor
factor
*
number
number
number

34 − 3 − 42


exp
exp
addop
term
exp
addop
term
−
factor
term
−
factor
number
factor
number
number

Real life example

left associative

Operator Precedence

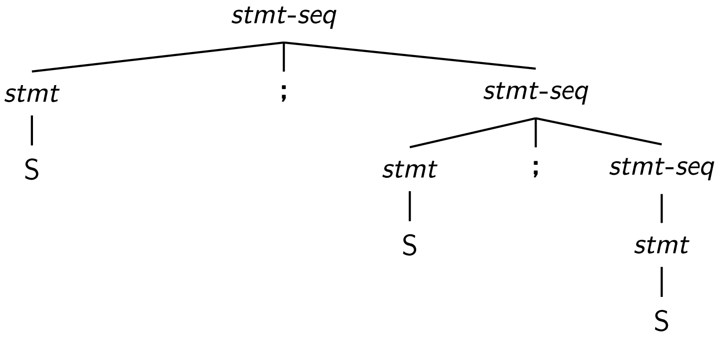
Java performs operations assuming the following ordering (or ***precedence***) rules if parentheses are not used to determine the order of evaluation (operators on the same line are evaluated in left-to-right order subject to the conditional evaluation rule for && and ||). The operations are listed below 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>

Non-essential ambiguity

left-assoc

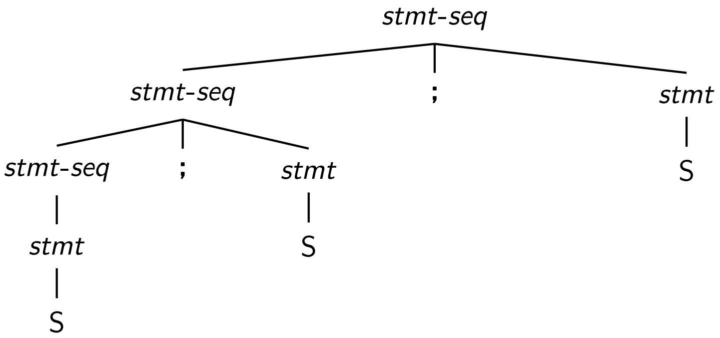
$$\begin{aligned} \textit{stmt-seq} &\rightarrow \textit{stmt-seq} \,\mathbf{;}\, \textit{stmt} \mid \textit{stmt} \\ \textit{stmt} &\rightarrow S \end{aligned}$$



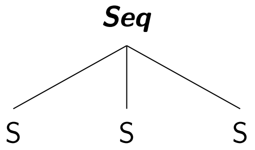
Non-essential ambiguity (2)

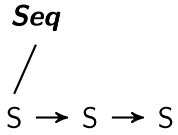
right-assoc representation instead

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



Possible AST representations





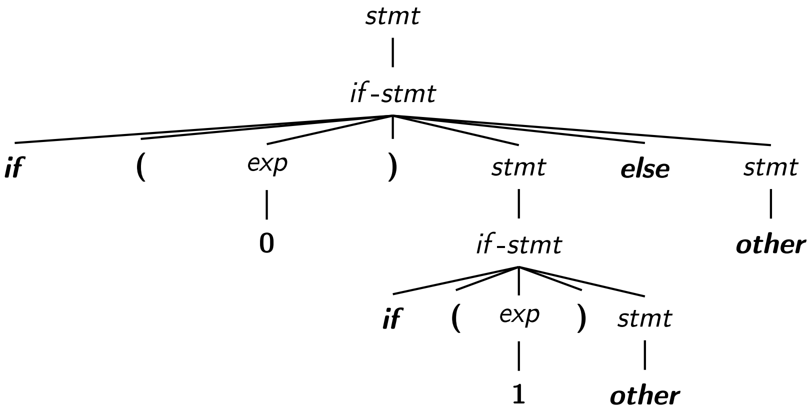
Nested if's

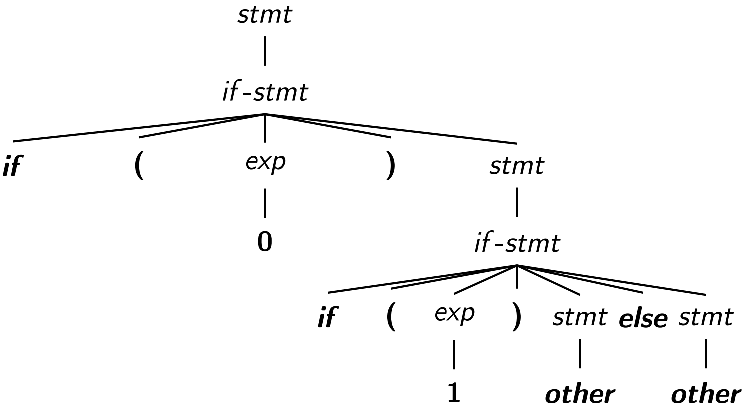
$$\textbf{\textit{if}}\ (\,0\,)\ \textbf{\textit{if}}\ (\,1\,)\ \textbf{\textit{other else other}}$$

Remember grammar from equation (5):

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

Should it be like this





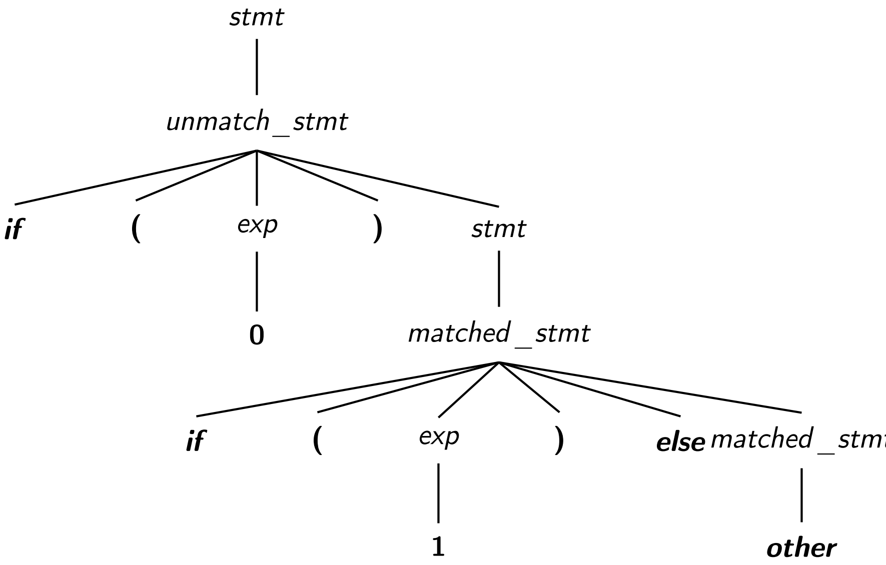
- common convention: connect ***else*** to closest "free" (= dangling) occurrence

Unambiguous grammar

Grammar

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

- 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
1
other

Adding sugar: extended BNF

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

EBNF

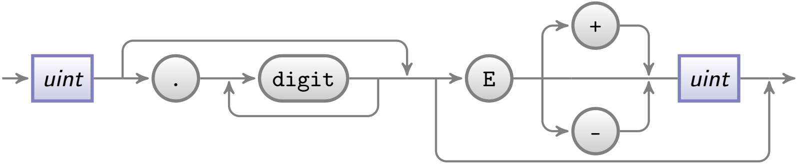
Main additional notational freedom: use regular expressions on the rhs of productions. They can contain terminals and non-terminals

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

$$
\begin{array}{rcll}
A & \rightarrow & \beta\{\alpha\} & \text{for } A \rightarrow A\alpha \mid \beta \\
A & \rightarrow & \{\alpha\}\beta & \text{for } A \rightarrow \alpha A \mid \beta \\
\textit{stmt-seq} & \rightarrow & \textit{stmt}\ \{\textbf{;}\ \textit{stmt}\} & \\
\textit{stmt-seq} & \rightarrow & \{\textit{stmt}\ \textbf{;}\}\ \textit{stmt} & \\
\textit{if-stmt} & \rightarrow & \textbf{\textit{if}}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}[\textbf{\textit{else}}\ \textit{stmt}] &
\end{array}
$$

greek letters: for non-terminals or terminals.

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



The Chomsky hierarchy

- linguist Noam Chomsky [Chomsky, 1956]
- 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 $a, b, \ldots \in \Sigma_N$,
- non-terminals $A, B, \ldots \in \Sigma_T$
- 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

BNF-grammar for *TINY*

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

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

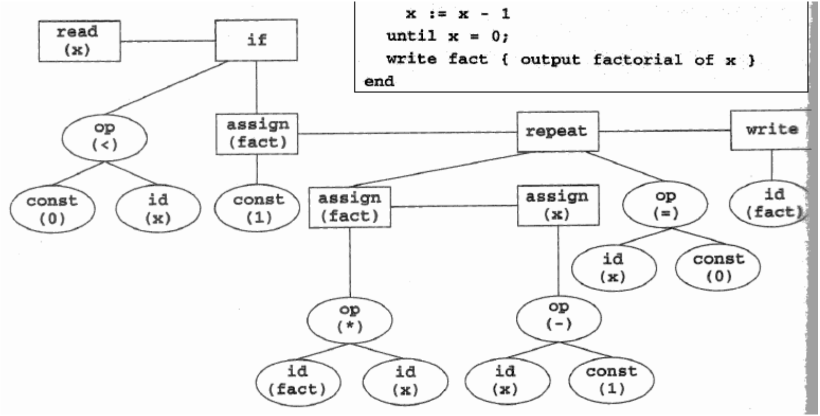
```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0
  write fact   { output factorial of x }
end
```

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



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?

References I

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