Compiler construction

Martin Steffen

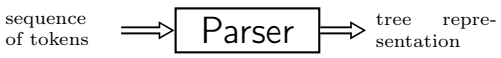
January 31, 2016

Contents

1 Grammars

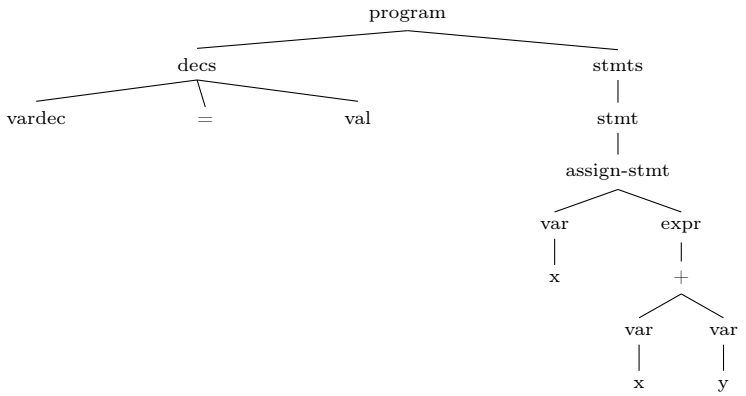
1.1 Introduction

1. 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

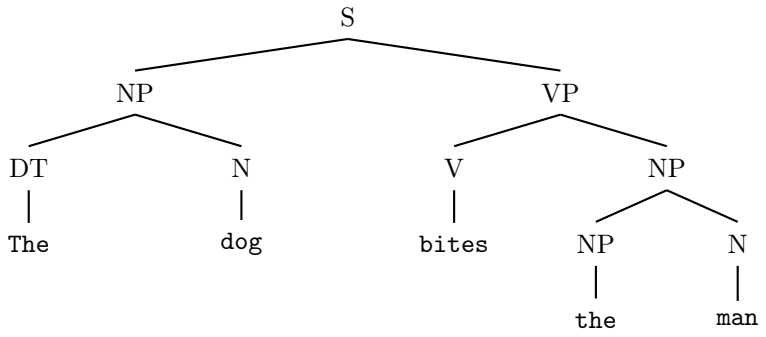
2. Sample syntax tree



(a) Syntax tree

The displayed syntax tree is meant "impressionistic" rather then formal. Neither is it a sample syntax tree of a real programming language, nor do we want to illustrate for instance special features of an *abstract* syntax tree vs.\ a *concrete* syntax tree (or a parse tree). Those notions are closely related and corresponding trees might all looks similar to the tree shown. There might, however, be subtle conceptual and representational differences in the various classes of trees. Those are not relevant yet.

3. Natural-language parse tree



4. "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**
- sometime we use $\langle$`integer`, "42"$\rangle$
  - `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)

(a) Token names and terminals

Remark 1 (Token (names) and terminals)**.** *We said, that sometimes one uses the name "token" just to mean token symbol, ignoring its value (like "42" from above). Especially, in the conceptual discussion and treatment of context-free grammars, which form the core of the specifications of a parser, the token value* is basically irrelevant*. Therefore, one simply identifies "tokens = terminals of the grammar" and silently ignores the presence of the value. In an implementation, and in lexer/parser generators, the value* "42" *of an integer-representing token must obviously* not *be forgotten, though . . . The grammar maybe the core of the specification of the syntactical analysis, but the result of the scanner, which resulted in the lexeme* "42" *must nevertheless not be thrown away, it's only not really part of the parser's tasks.*

(b) Notations

Remark 2. *Writing a compiler, especially a compiler front-end comprising a scanner and a parser, but to a lesser extent also for later phases, is about implementing representation of syntactic structures. The slides here don't* implement *a lexer or a parser or similar, but* describe *in a hopefully unambiguous way the principles of how a compiler front end works and is implemented. To describe that, one needs "language" as well, such as English language*

(mostly for intuitions) but also "mathematical" notations such as regular expressions, or in this section, context-free grammars. Those mathematical definitions have themselves a particular syntax; *one can see them as formal* domain-specific languages *to describe (other) languages. One faces therefore the (unavoidable) fact that one deals with two levels of languages: the language who is described (or at least whose* syntax *is described) and the language used to descibe that language. The situation is, of course, analogous when implementing a language: there is the language used to implement the compiler on the one hand, and the language for which the compiler is written for. For instance, one may choose to implement a C^{++}-compiler in C. It may increase the confusion, if one chooses to write a C compiler in C . . . . Anyhow, the language for describing (or implementing) the language of interest is called the* meta-language, *and the other one described therefore just "the language".*

When writing texts or slides about such syntactic issues, typically one wants to make clear to the reader what is meant. One standard way nowadays are typographic *conventions, i.e., using specific typographic fonts. I am stressing "nowadays" because in classic texts in compiler construction, sometimes the typographic choices were limited. []*

1.2 Context-free grammars and BNF notation

1. 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

   (a) Slogan A CFG describes the syntax of a programming language. [1]

   (b) Rest

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

   (c) Remarks on grammars

   Sometimes, the word "grammar" is synonymously for context-free grammars, as CFG are so central. However, context-sensitive and Turing-expressive grammars exists, both more expressive than CFG. Also a restricted class of CFG corresponds to regular expressions/languages. Seen as a grammar, regular expressions correspond so-called left-linear grammars (or alternativelty, right-linear grammars), which are a special form of context-free grammars.

2. Context-free grammar

   Definition 1 (CFG)**.** A *context-free grammar* G is a 4-tuple $G = (\Sigma_T, \Sigma_N, S, P)$:

   (a) 2 disjoint finite alphabets of terminals Σ_T and

   (b) non-terminals Σ_N

   (c) 1 start-symbol $S \in \Sigma_N$ (a non-terminal)

   (d) 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

3. BNF notation

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

- 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

(a) Slogan: Backus-Naur form
What regular expressions are for regular languages is BNF for context-free languages.

4. "Expressions" in BNF

$$\begin{array}{rcl} \mathit{exp} & \rightarrow & \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid (\ \mathit{exp}\ ) \mid \boldsymbol{number} \\ \mathit{op} & \rightarrow & + \mid - \mid * \end{array} \tag{1}$$

- "$\rightarrow$" indicating productions and " | " indicating alternatives. [2]
- 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 are not relevant here.

(a) Terminals
Conventions are not 100% followed, often bold fonts for symbols such as + or (are unavailable. The alternative using for instance ***PLUS*** and ***LPAREN*** looks ugly. Even if this might reminisce to the situation in concrete parser implementation, where + might by implemented by a concrete class named `Plus` —classes or identifiers named + are typically not available— most texts don't follow conventions so slavishly and hope of intuitive understanding of the educated reader.

5. 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

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

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

(a) "Standard" BNF
Specific and unambiguous notation is important, in particular if you *implement* a concrete language on a computer. On the other hand: understanding the underlying concepts by *humans* is at least equally important. In that way, bueaucratically fixed notations may distract from the core, which is understanding the principles. BTW: XML, anyone? Most textbooks (and we) rely on simple typographic conventions (boldfaces, italics). For "implementations" of BNF specification (as in tools like yacc), the notations, based mostly on ASCII, cannot rely on such typographic conventions.

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

(b) Syntax of BNF

BNF and its variations is a notation to describe "languages", more precisely the "syntax" of context-free languages. Of course, BNF notation, when exactly defined, is a language in itself, namely a domain-specific language to describe context-free languages. It may be instructive to write a grammar for BNF in BNF, i.e., using BNF as meta-language to describe BNF notation (or regular expressions). Is it possible to use regular expressions as meta-language to describe regular expression?

6. 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":

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

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

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

   - still: we count 6 productions

7. Grammars as language generators

   (a) 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.

   (b) Rest

   - *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.

   (c) **language** of grammar G

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

8. Example derivation for $(\boldsymbol{number} - \boldsymbol{number}) * \boldsymbol{number}$

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

   - underline the "place" were a rule is used, i.e., an *occurrence* of the non-terminal symbol is being rewritten/expanded
   - here: *leftmost* derivation[3]

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

9. Rightmost derivation

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

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

10. 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

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

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

    (a) Remarks

    Remark 3. *There can be many more that the one mentioned. A CFG that derives ultimately only 1 word of terminals (or a finite set of those) does not make much sense either.*

    Remark 4 ("Easy" sanitary conditions for CFGs)**.** *We stated a few conditions to avoid grammars which technically qualify as CFGs but don't make much sense; there are easier ways to describe an empty set ...*

    There's a catch, though: it might not immediately be obvious that, for a given G, the question $\mathcal{L}(G) =^{?} \emptyset$ is decidable!

    Whether a regular expression describes the empty language is trivially decidable immediately. Whether a finite state automaton descibes the empty language or not is, if not trivial, then at least a very easily decidable question. For context-sensitive grammars (which are more expressive than CFG but not yet Turing complete), the emptyness question turns out to be undecidable. Also, other interesting questions concerning CFGs are, in fact, undecidable, like: given two CFGs, do they describe the same language? Or: given a CFG, does it actually describe a regular language? Most disturbingly perhaps: given a grammar, it's undecidable whether the grammar is ambiguous or not. So there are interesting and relevant properties concerning CFGs which are undecidable. Why that is, is not part of the pensum of this lecture (but we will at least encounter the concept of grammatical ambiguity later). Coming back for the initial question: fortunately, the emptyness problem for CFGs is *decidable.*

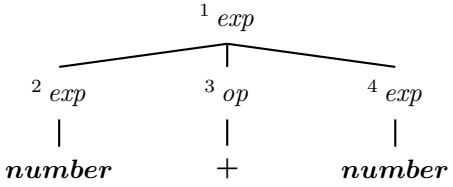
    Questions concerning decidability may seem not too relevant at first sight. Even if some grammars can be constructed to demonstrate difficult questions, for instance related to decidability or worst-case complexity, the designer of a language will not intentionally try to achieve an obscure set of rules whose status is unclear, but hopefully strive to capture in a clear manner the syntactic principles of a equally hopefully clearly structured language. Nonetheless: grammars for real language may become large and complex, and, even if conceptually clear, may contain unexcepted bugs which makes them behave different from expectation (for instance caused by a simple typo in one of the many rules).

    In general, the implementor of a parser will mostly rely on automatic tools ("parser generators") which take as an input a CFG and turns it in into an implementation of a recognizer, which does the syntactic analysis. Such tools obviously can reliably and accurately help the implementor of the parser automatically only for problems which are decidable. *For undecidable problems, one could still achieve things automatically, provided one would compromise by not insisting*

that the parser always terminates (but that's generally seens as unacceptable), or at the price of approximative *answers. It should also be mentioned that parser generators typcially won't tackle CFGs in their* full generality *but are tailor-made for well-defined and well-understood subclasses thereof, where efficient recognizers are automaticlly generatable.*

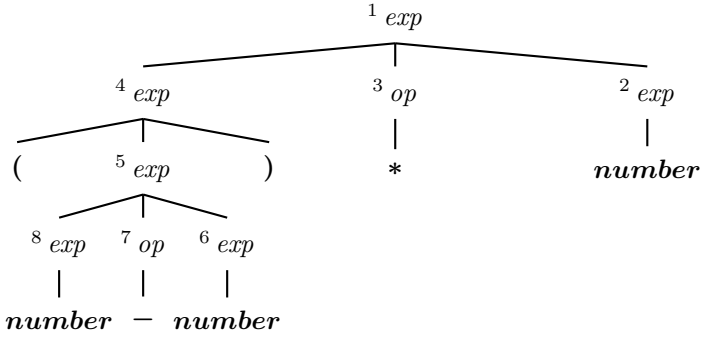
11. Parse tree

    - 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.[4]



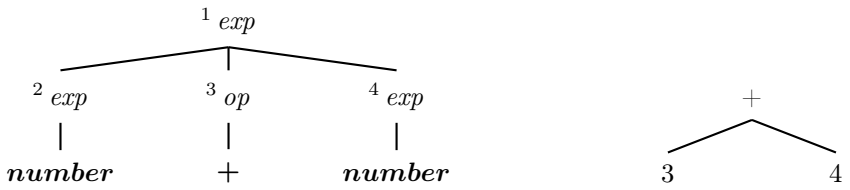
    - numbers in the tree
        - *not* part of the parse tree, indicate order of derivation, only
        - here: lefttmost derivation

12. Another parse tree (numbers for rightmost derivation)



13. 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`" ...)



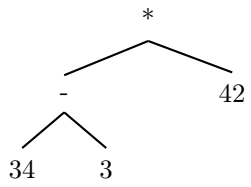
14. AST vs. CST

    - **parse tree**

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

  - 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 (e.g., when producing the AST)

15. 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): nothing wrong with it either.

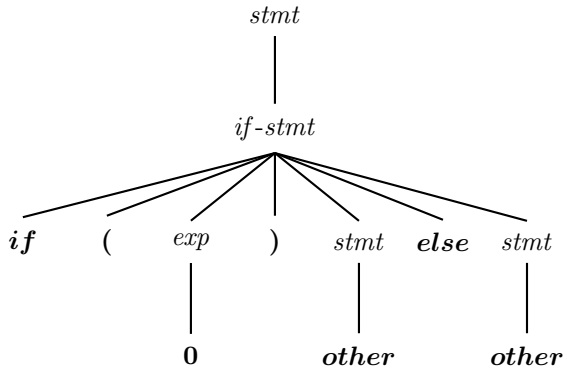
16. Conditionals

(a) Conditionals G_1

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

17. Parse tree

if* (0) *other* *else* *other



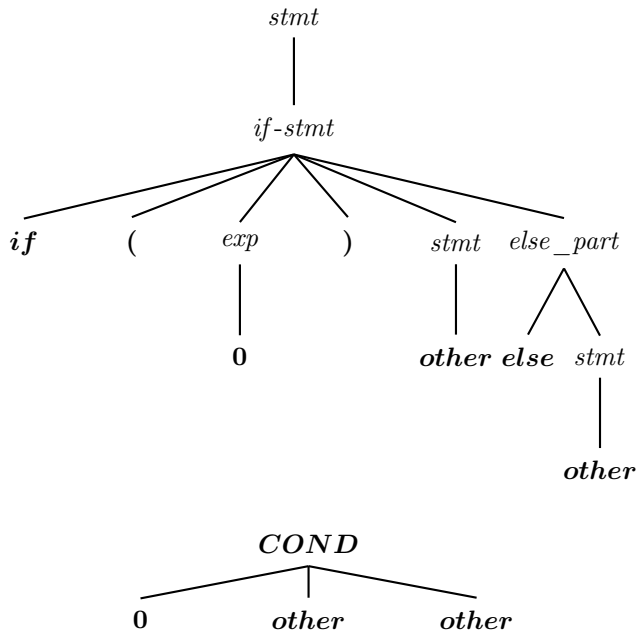
18. Another grammar for conditionals

(a) Conditionals G_2

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

(b) Abbreviation ϵ = empty word

19. A further parse tree + an AST



(a) Note A missing else part may be represented by null-pointers in languages like Java.

1.3 Ambiguity

1. Ambiguous grammar

Definition 2 (Ambiguous grammar)**.** A grammar is *ambiguous* if there exists a word with *two different* parse trees.

Remember grammar from equation (1):

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid (\ \mathit{exp}\ ) \mid \boldsymbol{number} \\ \mathit{op} & \to & + \mid - \mid * \end{array}$$

Consider:

$$\boldsymbol{number} - \boldsymbol{number} * \boldsymbol{number}$$

2. 2 resulting ASTs

different parse trees $\Rightarrow$ different[5] ASTs $\Rightarrow$ different[5] meaning

(a) 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`?

3. Precendence & associativity

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

[5] At least in most cases.

- 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)

4. 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)

5. Expressions, revisited

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

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

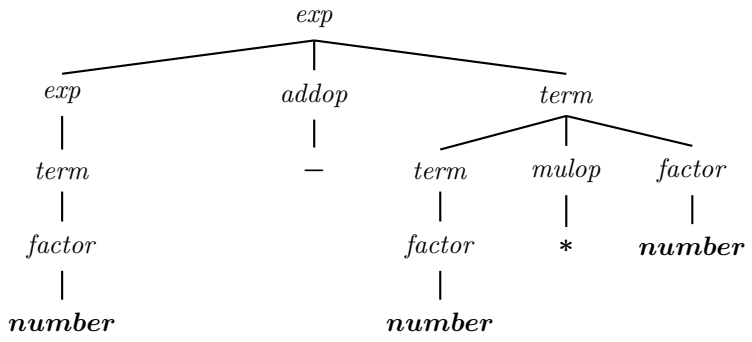
     - *right*-assoc: analogous, but right-recursive
     - *non*-assoc:

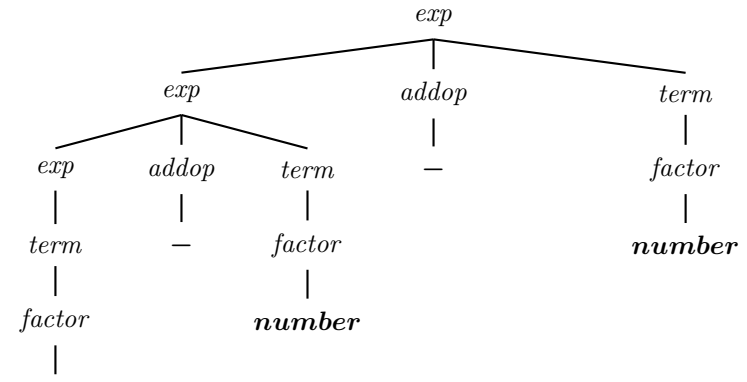
$$exp \rightarrow term\ addop\ term \ \mid \ term$$

   (a) factors and terms

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

6. $34 - 3 * 42$





7. $34 - 3 - 42$

   (a) Ambiguity

   The question whether a given CFG is ambiguous or not is *undecidable*. Note also: if one uses a parser generator, such as yacc or bison (which cover a subset of CFGs), the resulting recognizer is *always* determinstic. In case the construction encounter ambiguous situations, they are "resolved" by making a specific choice. Nonetheless, such ambiguities indicate often that the formulation of the grammar (or even the language it defines) has problematic aspects. Most programmars as "users" of a programming language may not read the full BNF definition, most will try to grasp the language looking at sample code pieces mentioned in the manual etc. And even if they bother studying the exact specification of the system, i.e., the full grammar, ambiguities are *not* obvious (after all, it's undecidable). Hidden ambiguities, "resolved" by the generated parser, may lead misconceptions as to what a program actually means. It's similar to the situation, when one tries to study a book with arithmetic being unaware that multiplication binds stronger than addition. A parser implementing such grammars may make consistent choices, but the programmer using the compiler may not be aware of them. At least the compiler writer, responsible for designing the language, will be informed about "/conflicts/" in the grammar and a careful designer will try to get rid of them. This may be done by adding associativities and precedences (when appropriate) or reformulating the grammar, or even reconsider the syntax of the language. While ambiguities and conflicts are generally a bad sign, arbitrarily adding a complicated "precedence order" and "associativities" on all kinds of symbols or complicate the grammar adding ever more separate classes of nonterminals just to make the conflicts go away is not a real solution either. Chances are, that those parser-internal "tricks" will be lost on the programmer as user of the language, as well. Sometimes, makeing the *language* simpler (as opposed to complicate the grammar for the same language) might be the better choice. That typically be done by making the language more verbose and reducing "overloading" of syntax. Of course, going overboard by making groupings etc. of all constructs crystal clear to the parser, may also lead to non-elegant designs. Lisp is a standard example, notoriously known for it's extensive use of parentheses. Basically, the programmer directly writes down *syntax trees*, which certainly removes all ambiguities, but still, mountains of parentheses are also not the easiest syntax for human consumption. So it's a tricky balance.

   But in general: if it's enormously complex to come up with a reasonably unambigous grammar for an intended language, chances are, that reading programs in that language and intutively grasping what is intended will be hard for humans, too.

   Note also: since already the question, whether a given CFG is ambigiguous or not is undecidable, it should be clear, that the following question is undecidable as well: given a grammar, can I reformulate it, still accepting the same language, that it becomes unambiguous?

8. Real life example

left associative

Operator Precedence

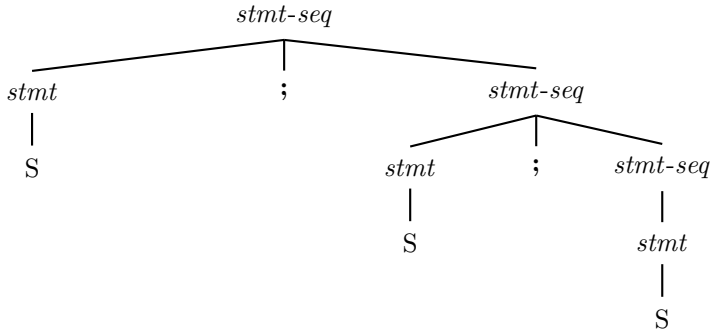
Java performs operations assuming the following ordering (or ***precedence***) rules if parentheses are not used to determine the order of evaluation (operators on the same line are evaluated in left-to-right order subject to the conditional evaluation rule for && and ||). The operations are listed below 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	^
bitwise-or	|
and	&&
or	||
conditional	⟨bool_exp⟩? ⟨true_val⟩: ⟨false_val⟩
assignment	=
op assignment	+= -= *= /= %=
bitwise assign.	>>= <<= >>>=
boolean assign.	&= ^= |=

9. Non-essential ambiguity

   (a) left-assoc

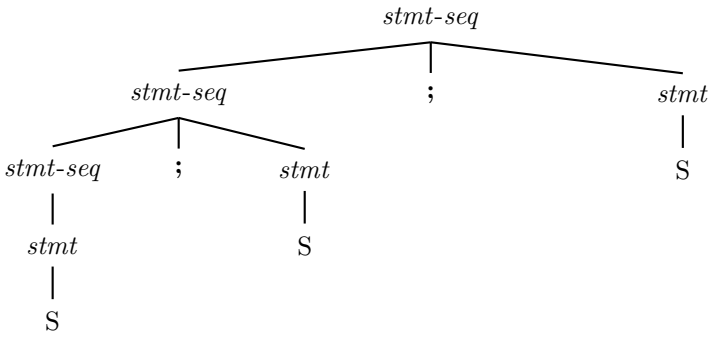
$$\begin{aligned} \textit{stmt-seq} &\rightarrow \textit{stmt-seq}\,;\,\textit{stmt} \mid \textit{stmt} \\ \textit{stmt} &\rightarrow S \end{aligned}$$



10. Non-essential ambiguity (2)

    (a) right-assoc representation instead

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



11. Possible AST representations

12. Dangling else

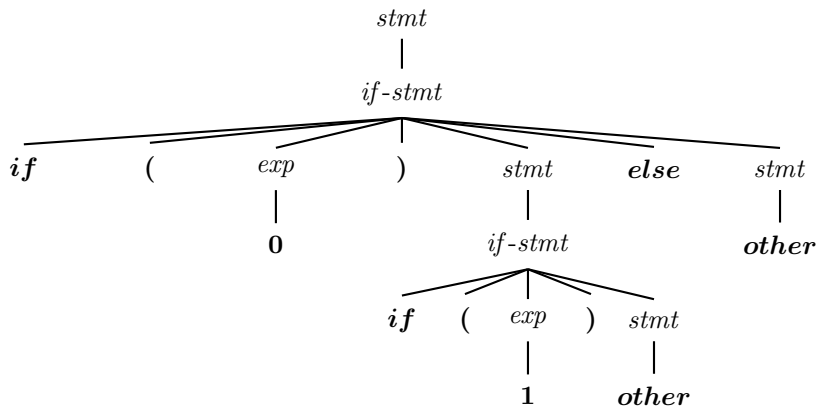
    (a) Nested if's

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

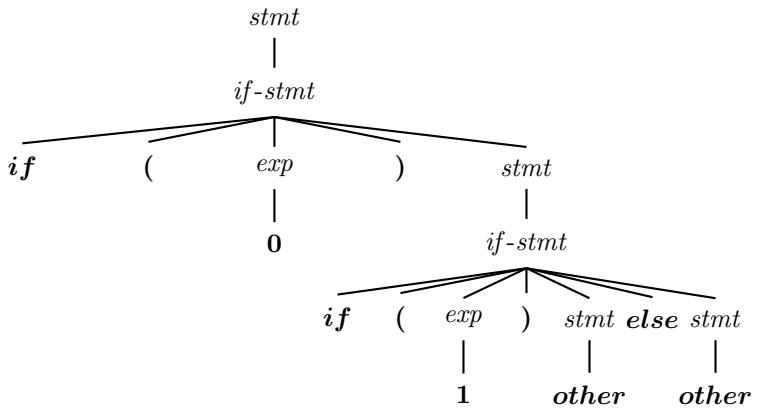
    (b) :$B_{\text{ignoreheading}}$: Remember grammar from equation (5):

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

13. Should it be like this



14. ... or like this



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

15. Unambiguous grammar

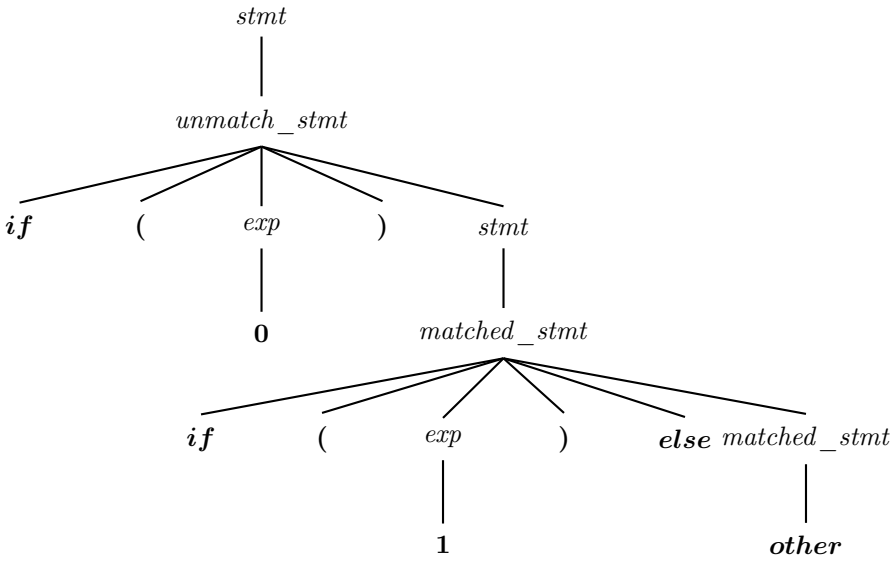
    (a) Grammar

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

    (b) :$B_{\text{ignoreheading}}$:

    - 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***

16. CST



17. Adding sugar: extended BNF

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

    (a) EBNF Main additional notational freedom: use regular expressions on the rhs of productions. The can contain terminals and non-terminals

    (b) Rest

    - 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)

18. EBNF examples

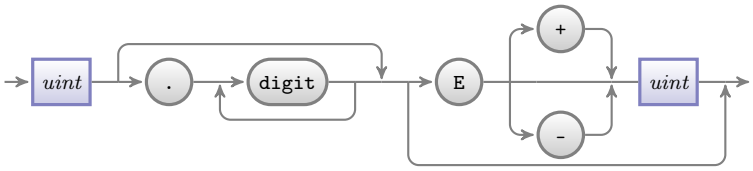
$$
\begin{array}{rcll}
A & \to & \beta\{\alpha\} & \text{for } A \to A\alpha \mid \beta \\
A & \to & \{\alpha\}\beta & \text{for } A \to \alpha A \mid \beta \\
\textit{stmt-seq} & \to & \textit{stmt}\ \{\textbf{;}\ \textit{stmt}\} & \\
\textit{stmt-seq} & \to & \{\textit{stmt}\ \textbf{;}\}\ \ \textit{stmt}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

greek letters: for non-terminals or terminals.

1.4 Syntax diagrams

1. 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 unsigned integer (taken from the TikZ manual):



1.5 Chomsky hierarchy

1. The Chomsky hierarchy

   - linguist Noam Chomsky [Chomsky, 1956]
   - **important** classification of (formal) languages (sometimes Chomsky-Schtzenberger)
   - 4 levels: type 0 languages – type 3 languages
   - levels related to machine models that generate/recognize them
   - so far: regular languages and CF languages

2. 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

   (a) Conventions

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

(b) Remark: Chomsky hierarchy
The rule format for type 3 languages (= regular languages) is also called right-linear. Alternatively, one can use *right-linear* rules. If one mixes right- and left-linear rules, one leaves the class of regular languages. The rule-format above allows only *one* terminal symbol. In principle, if one had sequences of terminal symbols in a right-linear (or else left-linear) rule, that would be ok too.

3. Phases of a compiler & hierarchy

(a) "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?

(b) Rest 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

1.6 Syntax of Tiny

1. BNF-grammar for *TINY*

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

2. 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 */
```

3. 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

4. Sample Tiny program

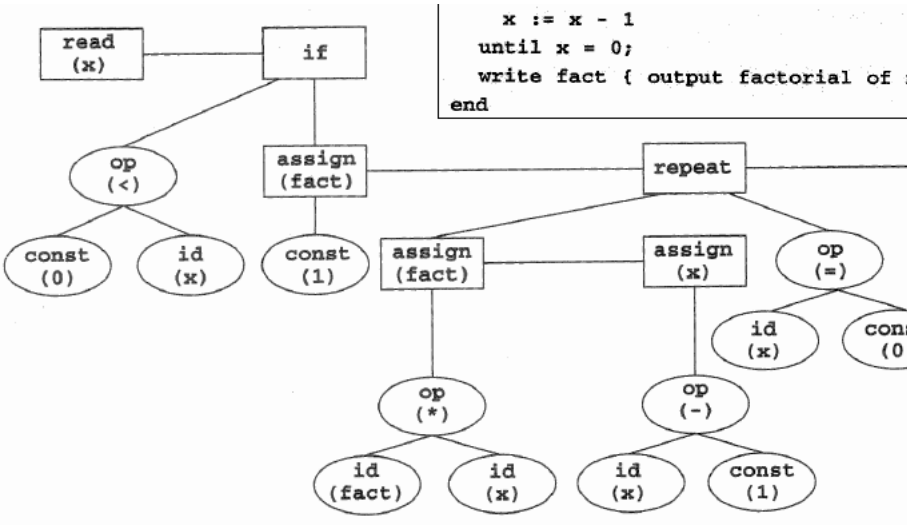
```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0
  write fact   { output factorial of x }
end
```

5. 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

6. Abstract syntax tree for a tiny program



7. Some questions about the Tiny grammy 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

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

Index