Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

Chapter 3 Grammars

What is it about?

Learning Targets of this Chapter

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]).

Contents

3.1 Introduction

The compiler phase after the lexer is the parser. In the lecture, treating that phase is done in two chapters. The first one, i.e., the current one, covers the underlying concepts, namely context-free grammars, and one that deals with the parsing process. Context-free grammars resp. notations for context-free grammars play the same role for parsing that regular expressions played for lexing. There are grammars other than context-free grammars, later we will at least mention the so-called Chomsky hierarchy, the most well-known classification of language description formalisms. Context-free languages correspond to one level there, and actually regular language to another one, actually the simplest level; regular language can be seen as a restricted form of context-free languages.

Context-free grammars are probably the most-well known example of grammars, so when speaking simply about "a grammar", one often just means context-free grammar, though there are other types as well, as said.

Context-free grammars specify the *syntax* of a language, as opposed to regular expressions, which specify the *lexical aspect* of the language. That's basically by convention: the syntax of the language refers to those aspects that can be captured by a context-free grammar.

When it comes to *parsing*, one typcially don't make use of the full power of context-free grammars, one restricts oneself to special, limited forms of them, for practical reasons. We come to that in the parsing chapter. One restriction one wants to impose on parsing will already be discussed in this chapter. That is that one does not want the grammar to be *ambigious*. Ambiguous grammars are not useful in parsing, as we will discuss.

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

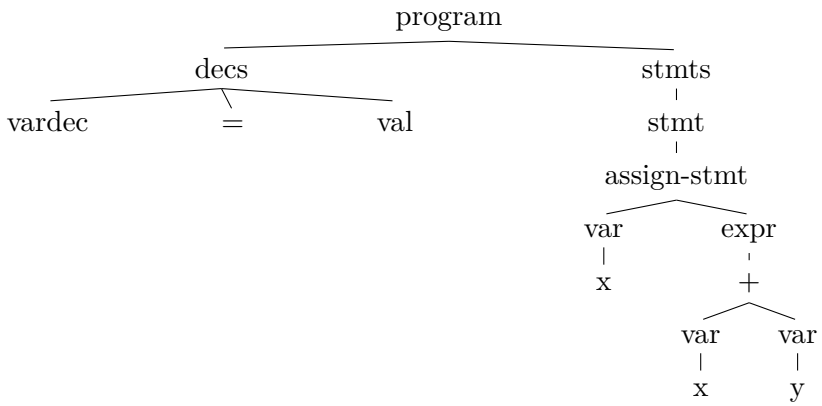
Parsing

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

Parsing is concerned with context-free grammars. As mentioned, one will generally not try to use the full-power of context-free grammars, but make some restrictions. To the very least, one insists on the grammar to be *non-ambiguous*. We come to the important notion of ambiguity of context-free grammars (and of context-free languages) later. More globally, there are different classes of grammars, some more restritive than context-free grammars, some more expressive. Actually, regular languages correspond to a restricted form of context-free languages. They are too restricted, thought, to be used for parsing (but good enough for lexing).

The slide talks about deriving "words". In general, words are finite sequences of symbols from a given alphabet (as was the case for regular languages). In the concrete picture of a parser, the words are sequences of *tokens*, which are the elements that come out of the scanner. A successful derivation leads to tree-like representations. There are various slightly different forms of trees connected with grammars and parsing, which we will later see in more detail; for a start now, we will just informally illustrate such tree-like structures, without distinguishing between (abstract) syntax trees and parse trees.

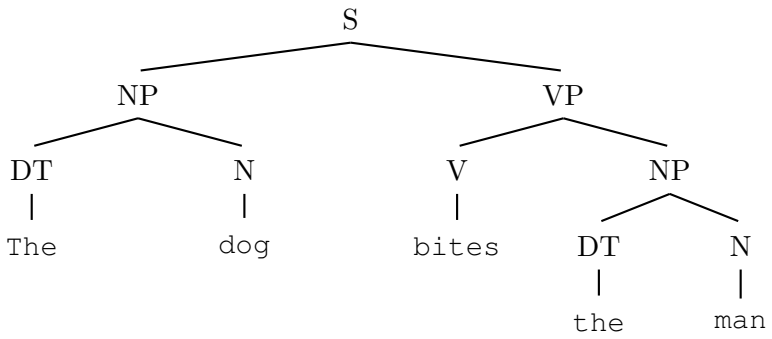
Sample syntax tree



Syntax tree

The displayed syntax tree is meant "impressionistic" rather than 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 look 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, at the beginning of this section.

Natural-language parse tree



The concept of context-free grammars goes back to Chomsky (and Schützenberger). They were (also) used in describing natural languages, not computer languages (Chomsky is, among other things, a linguist). So the tree represents the syntactic structure of a (simple) English sentence. What the tree is exactly supposed to mean is not too important (*VP* and *NP* stand for verb-phrase and noun-phrase etc.).

"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 $\langle$`integer`, "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)

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 values. In an implementation, and in lexer/parser generators, the value* "42" *of an integer-representing token must obviously* not *be forgotten, though . . . The grammar may be 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.*

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 the 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 that is described (or at least whose* syntax *is described) and the language used to descibe that language. We face the same when talking about regular languages, in particular regular expressions. The situation is, of course, the same when writing a book teaching a human language: there is a language being taught, and a language used for teaching (both may be different). More closely, it's analogous when implementing a general purpose programming 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 . . . (that was a bit discussed in the introductory chapter, under the header "bootstrapping"). 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 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 (maybe written as "typoscript", i.e., as "manuscript" on a type writer).*

3.2 Context-free grammars and BNF notation

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]

Note: a compiler might reject some syntactically correct programs, whose violations *cannot* be captured by CFGs. That is done by *subsequent* phases. For instance, the type checker will reject many syntactically correct programs that are ill-typed. The type checker is an important part from the *semantic* phase (or *static analysis* phase). A typing discipline is *not* a syntactic property of a language (in that it cannot captured most commonly by a context-free grammar), it's therefore a "semantic" property.

Remarks on grammars

Sometimes, the word "grammar" is synonymously for context-free grammars, as CFGs are so central. However, the concept of grammars is more general; there exists context-sensitive and Turing-expressive grammars, both more expressive than CFGs. Also a restricted class of CFG correspond 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.

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

Context-free grammar

Definition 3.2.1 (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

Those notions will be explained with the help of examples.

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

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp}\ \ \mathit{op}\ \ \mathit{exp} \mid (\ \mathit{exp}\ ) \mid \textbf{number} \\ \mathit{op} & \to & + \mid - \mid * \end{array} \tag{3.1}$$

- "$\to$" indicating productions and " | " 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.

The grammar on the slide consists of 6 productions/rules, 3 for *expr* and 3 for *op*, the | is just for convenience. Side remark: Often also ::= is used for →.

Terminals

Conventions are not always 100% followed, often bold fonts for symbols such as + or (are unavailable or not easily visible. The alternative using, for instance, boldface "identifiers" like **PLUS** and **LPAREN** looks ugly. Some books would write '+' and '('.

In a concrete parser implementation, in an object-oriented setting, one might choose to implement terminals as classes (resp. concrete terminals as instances of classes). In that case, a class name + is typically not available and the class might be named `Plus`. Later we will have a look at how to systematically implement terminals and non-terminals, and having a class `Plus` for a non-terminal '+' etc. is a systematic way of doing it (maybe not the most efficient one available though).

Most texts don't follow conventions so slavishly and hope for an intuitive understanding by the educated reader, that + is a terminal in a grammar, as it's not a non-terminal, which are written here in *italics*.

Different notations

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

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

- Extended BNF (EBNF) and yet another style

$$\begin{array}{rcl} exp & \rightarrow & exp\ (\ "+"\ \mid\ "-"\ \mid\ "*"\ )\ exp \\ & \mid & "("\ exp\ ")"\ \mid\ "number" \end{array} \tag{3.2}$$

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

"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 likewise important. In that way, bureaucratically fixed notations may distract from the core, which is *understanding* the principles. XML, anyone? Most textbooks (and we) rely on simple typographic conventions (boldface, italics). For "implementations" of BNF specification (as in tools like yacc), the notations, based mostly on ASCII, cannot rely on such typographic conventions.

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?

Different ways of writing the same grammar

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

$$\begin{array}{rcl} \mathit{exp} & \rightarrow & \mathit{exp}\ \mathit{op}\ \mathit{exp} \\ \mathit{exp} & \rightarrow & (\ \mathit{exp}\ ) \\ \mathit{exp} & \rightarrow & \textbf{number} \\ \mathit{op} & \rightarrow & + \\ \mathit{op} & \rightarrow & - \\ \mathit{op} & \rightarrow & * \end{array} \tag{3.3}$$

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

$$\begin{array}{rcl} E & \rightarrow & E\ O\ E \mid (\ E\ ) \mid \textbf{number} \\ O & \rightarrow & + \mid - \mid * \end{array} \tag{3.4}$$

- 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.

Non-determinism means, that the process of derivation allows choices to be made, when applying a production. One can distinguish two forms of non-determinism here: 1) a sentential form contains (most often) more than one non-terminal. In that situation, one has the choice of expanding one non-terminal or the other. 2) Besides that, there may be more than one production or rule for a given non-terminal. Again, one has a choice.

As far as 1) is concerned. whether one expands one symbol or the other leads to different derivations, but won't lead to different *derivation trees* or *parse trees* in the end. Below, we impose a fixed discipline on *where* to expand. That leads to *left-most* or *right-most* derivations.

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]

Right-most derivation

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

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

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

Some easy requirements for reasonable grammars

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

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

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

There can be further conditions one would like to impose on grammars besides the one sketched. A CFG that derives ultimately only 1 word of terminals (or a finite set of those) does not make much sense either. There are further conditions on grammar characterizing their usefulness for *parsing*. So far, we mentioned just some obvious conditions of "useless" grammars or "defects" in a grammer (like superfluous symbols). "Usefulness conditions" may refer to the use of ϵ-productions and other situations. Those conditions will be discussed when the lecture covers *parsing* (not just grammars).

Remark 3 ("Easy" sanitary conditions for CFGs)**.** *We stated a few conditions to avoid grammars which technically qualify as CFGs but don't make much sense, for instance to avoid that the grammar is obviously empty; 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. Whether or not 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 have to deal with the important concept of grammatical ambiguity later). Coming back for the initial question: fortunately, the emptyness problem for CFGs* is *decidable.*

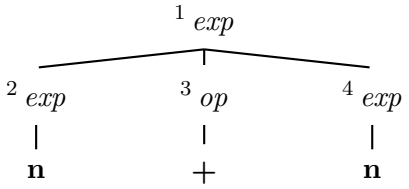
Questions concerning decidability may seem not too relevant at first sight. Even if some grammars can be constructed to demonstrate difficult questions, for instance related to decidability or worst-case complexity, the designer of a language will not intentionally try to achieve an obscure set of rules whose status is unclear, but hopefully strive to capture in a clear manner the syntactic principles of an equally hopefully clearly structured language. Nonetheless: grammars for real languages may become large and complex, and, even if

conceptually clear, may contain unexpected bugs which makes them behave unexpectedly (for instance caused by a simple typo in one of the many rules).

In general, the implementor of a parser will often 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 is seen 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. In the part about parsing, we will cover some such classes.*

Parse tree

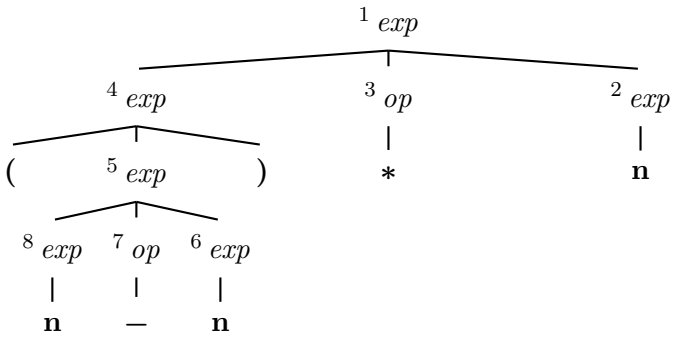
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent phases
- **parse tree**: structure for the *essence* of derivation
- also called **concrete** syntax tree.



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

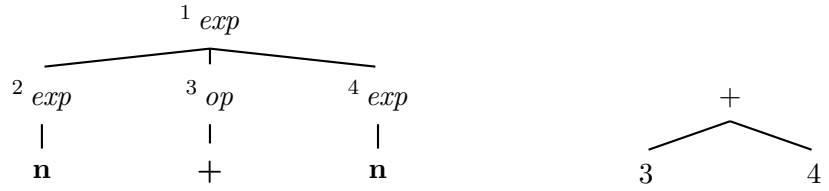
There will be *abstract* syntax trees, as well, in contrast to concrete syntax trees or parse trees covered here.

Another parse tree (numbers for right-most 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`" ...)

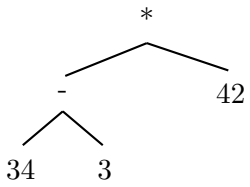


AST vs. CST

- **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

We should repeat: the shown ASTs are "schemantic" and for illustration. It's best to keep in mind, that in a concrete compiler, the AST is a data structure. A specific source file is then represented as a specific tree, i.e., as instance of the AST data structure.

Conditionals

Conditionals G_1

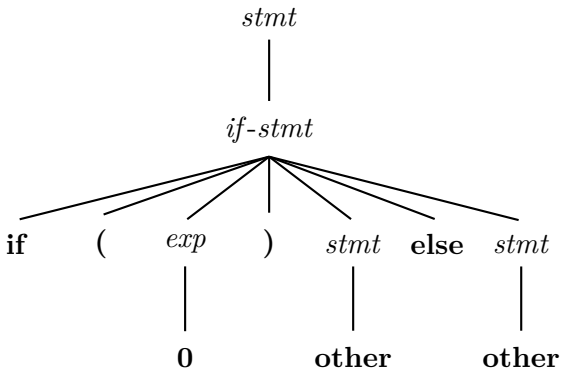
$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & \mathit{if\text{-}stmt} \mid \textbf{other} \\ \mathit{if\text{-}stmt} & \rightarrow & \textbf{if (}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt} \\ & \mid & \textbf{if (}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ \mathit{exp} & \rightarrow & \textbf{0} \mid \textbf{1} \end{array} \tag{3.5}$$

Conditionals in one syntactic form or other occur in basically all programming languages. As of now, we use the conditionals for not much more than pointing out something that should be rather obvious: there is (always) more than one way to describe an intended language by a context-free grammar. The same was the case for regular expressions, as well (and generally for all notational systems): there is always more than one way to describe things.

Of course, with more than one formulation, some may "better" than others. That may refer to "clarity" or readability for humans. But there are also aspects relevant for *parsing*. One formulation of a grammar may be in a form that is unhelpful for parsers. It may also depend of the chosen style of parsers: some formulations pose problems for top-down parsers resp. for bottom-up parsers. Issues like that will be discussed in the chapter of parsing, here we are still covering grammars. In particular on connection with conditionals (which is a classic example): the chosen syntax here will lead to *ambiguity*, which we will discuss later. In this particular examples, both formulations of the grammar are ambiguous (it will be a classical example of ambiguitity). Actually, it's quite straightforward to convince oneself, that one cannot reformulate the grammar even further, to get an equivalent but unambigous grammar. The ambiguity goes deeper (in this case): the *language itself* is ambiguous. We pick up on those issues later.

Parse tree

if (0) other else other



Another grammar for conditionals

Conditionals G_2

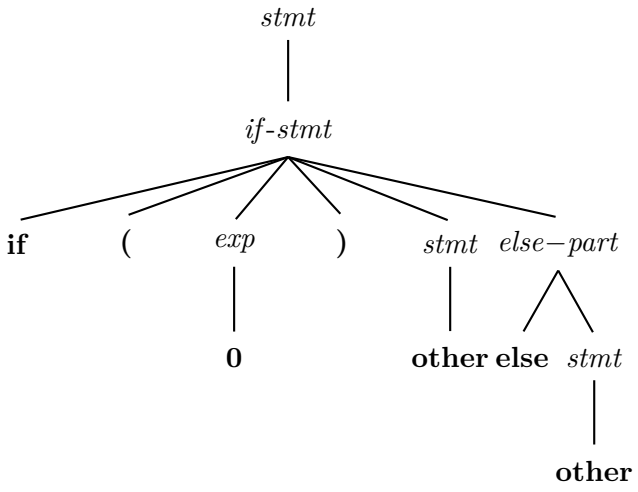
$$
\begin{array}{rcl}
\mathit{stmt} & \to & \mathit{if\text{-}stmt} \mid \textbf{other} \\
\mathit{if\text{-}stmt} & \to & \textbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt}\ \mathit{else{-}part} \\
\mathit{else{-}part} & \to & \textbf{else}\ \mathit{stmt} \mid \epsilon \\
\mathit{exp} & \to & \textbf{0} \mid \textbf{1}
\end{array}
\tag{3.6}
$$

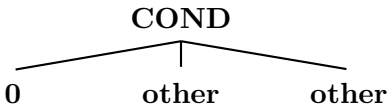
Abbreviation

ϵ = empty word

We have encountered the symbol ϵ before, in the context of regular languages. In regular expressions, the symbol $\boldsymbol{\epsilon}$ represents "the same"' as here: the empty word, the absence of a symbol, the empty sequence, etc.

A further parse tree + an AST





A potentially missing else part may be represented by null-"pointers" in languages like Java.

In functional languages, one could use "option" types to represent in a safer way the fact that the else part is there or may be missing. With null-pointers, there is always the danger that the programmer forgets that the value may not be there and then forgets to check that case properly, and cause some null pointer exception.

3.3 Ambiguity

Before we mentioned some "easy" conditions to avoid "silly" grammars, without going into detail. *Ambiguity* is more important and complex. Roughly speaking, a grammar is ambiguous, if there exist *sentences for which there are two different parse trees.* That's in general highly undesirable, as it means there are sentences with different syntactic interpretations (which therefore may ultimately interpreted differently). That is mostly a no-no, but even *if* one would accept such a language definition, parsing would be problematic, as it would involve *backtracking* trying out different possible interpretations during parsing (which would also be a no-no for reasons of efficiency) In fact, later, when dealing with actual concrete parsing procedures, they cover certain *specific* forms of CFG (with names like LL(1), LR(1), etc.), which are in particular non-ambiguous. To say it differently: the fact that a grammar is parseable by some, say, LL(1) top-down parser (which does not do backtracking) implies directly that the grammar is unambiguous. Similar for the other classes we'll cover.

Note also: given an ambiguous grammar, it is often possible to find a *different* "equivalent" grammar that *is* unambiguous. Even if such reformulations are often possible, it's not guaranteed: there are context-free languages which do have an ambiguous grammar, but no unambigous one. In that case, one speaks of an ambiguous context-free *language.* We concentrate on ambiguity of *grammars.*

Now that we have said that ambiguity in grammars must be avoided, we should however also say, that, in certain situations, one can in some way live with it. One way of living with it is: imposing extra conditions on the way the grammar is used, that removes it (in a way, priorizing some rules over others). In practice, that often takes the form of specifying assiciativity and binding powers of operators, like making clear that $1 + 2 + 3$ is "supposed" to be interpreted as $(1 + 2) + 3$ (addition is left-associative) and $1 + 2 \times 3$ is the same as $1 + (2 \times 3)$ (multiplication binds stronger than addition). The grammar as such is ambigiguous, but that's fine, since one can make it non-ambiguous by imposing such additional constraints. And not only can one do that technically, that form of disambiguation is also *transparent* for the user.

Tempus fugit ...

picture source: wikipedia

One famous sentence often used to illustrate ambiguity in natural languages is *"Time flies like a banana"*. That sentence is often attributed to Groucho Marx, but it's a bit aprocryphal.

Ambiguous grammar

Definition 3.3.1 (Ambiguous grammar)**.** A grammar is *ambiguous* if there exists a word with *two different* parse trees.

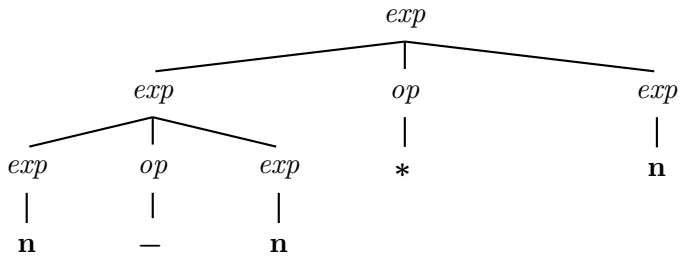
Remember grammar from equation (3.1):

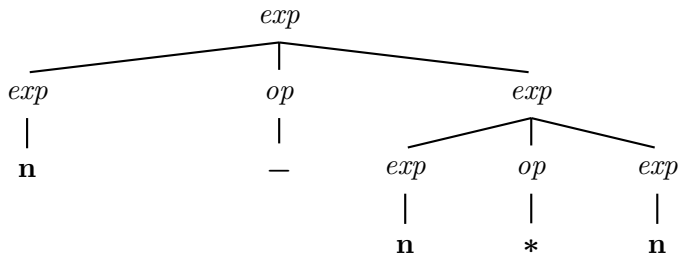
$$\begin{array}{rcl} exp & \to & exp\ \ op\ \ exp \mid (\ exp\ ) \mid \mathbf{number} \\ op & \to & + \mid - \mid * \end{array}$$

Consider:

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

2 CTS's





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 ?

The slides stipulates that diffferent parse trees lead to different ASTs and this in turn into different meanings. That is principle correct, but there may be special circumstances when that's not the case. Different CSTs may actually result in the same AST. Or also: it may lead to different AST which turn out to have the same meaning. The slide gave an example of where it's debatable whether two different ASTs have the same meaning or not.

Precendence & associativity

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

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

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

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

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

Unambiguity without 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)

The method sketched here ("precedence cascade") is a receipe to massage a grammar in such a way that the result captures intended precedences (and at the same time their associativities). It works in that way for syntax using *binary* operators. That receipe is commonly illustrated using numerical expressions. We will encounter analogous tasks also in the exercises.

Expressions, revisited

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

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

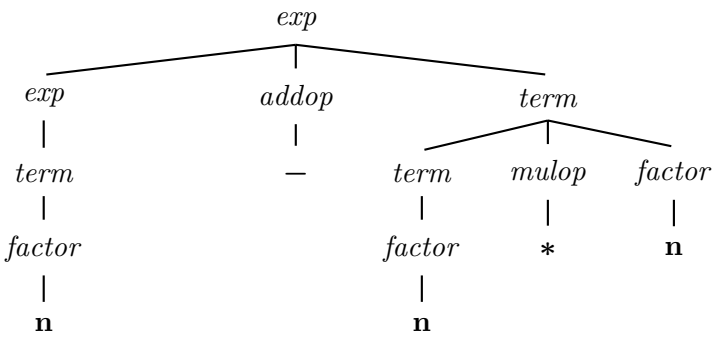
  - *right*-assoc: analogous, but right-recursive
  - *non*-assoc:

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

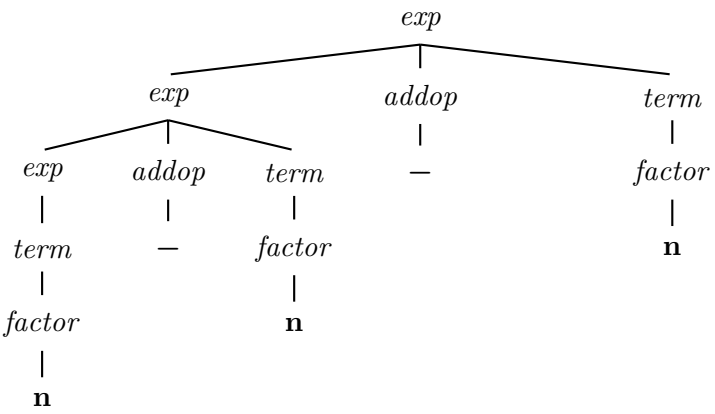
factors and terms

$$\begin{array}{rcl} exp & \rightarrow & exp\ \ addop\ \ term \mid term \\ addop & \rightarrow & + \mid - \\ term & \rightarrow & term\ \ mulop\ \ factor \mid factor \\ mulop & \rightarrow & * \\ factor & \rightarrow & (\ exp\ ) \mid \mathbf{number} \end{array} \tag{3.7}$$

$34 - 3 * 42$



$34 - 3 - 42$



Ambiguity

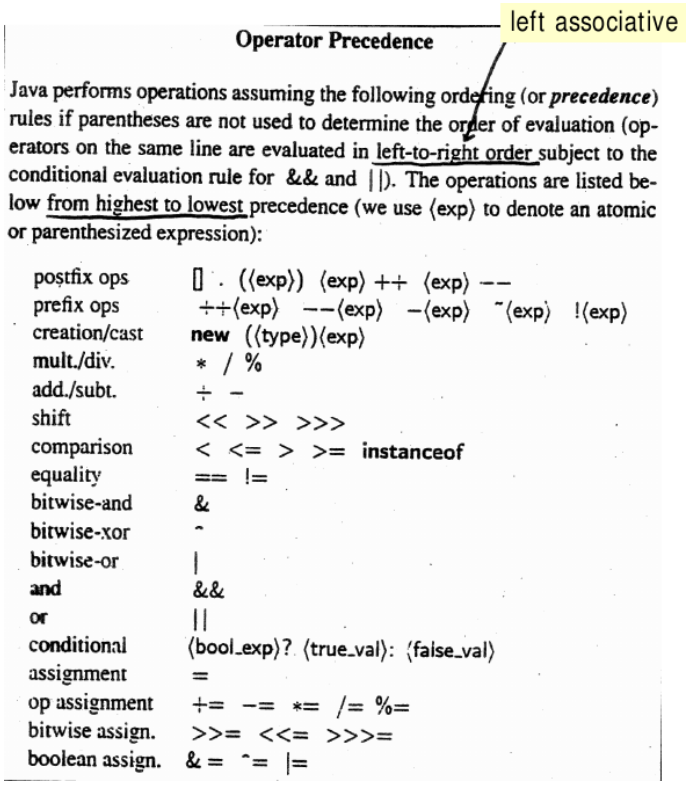
As mentioned, the question whether a given CFG is ambiguous or not is *undecidable.* Note also: if one uses a parser generator, such as yacc or bison (which cover a practically usefull subset of CFGs), the resulting recognizer is *always* deterministic. In case the construction encounters ambiguous situations, 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 programmers 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, at least the problem in general). Hidden ambiguities, "resolved" by the generated parser, may lead to 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. Without being aware of that, some sections won't make much sense. 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, making the *language* simpler (as opposed to complicate the grammar for the same language) might be the better choice. That can 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 its extensive use of parentheses. Basically, the programmer directly writes down *syntax trees*, which certainly removes ambiguities, but

still, mountains of parentheses are also not the easiest syntax for human consumption (for most humans, at least). So it's a balance (and at least partly a matter of taste, as for most design choices and questions of language pragmatics).

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 may be hard for humans, too.

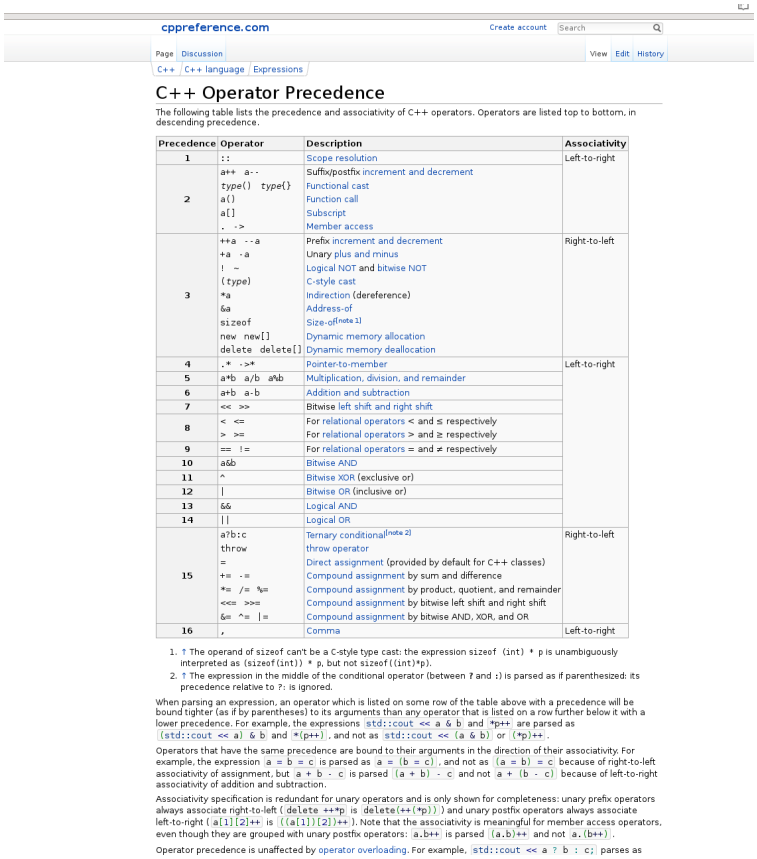
Note also: since already the question, whether a given CFG is ambiguous or not is undecidable, it should be clear, that the following question is undecidable, as well: given a grammar, can I reformulate it, still accepting the same language, that it becomes unambiguous?

Real life example



The scan is taken from an edition of the book "Java in a nutshell". The next example covering C++ is clipped from the net

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.

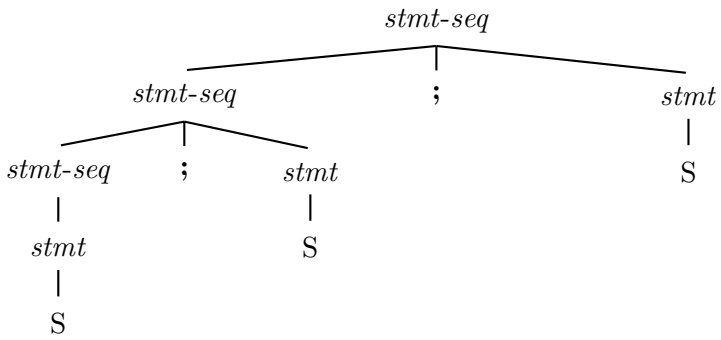
Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity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

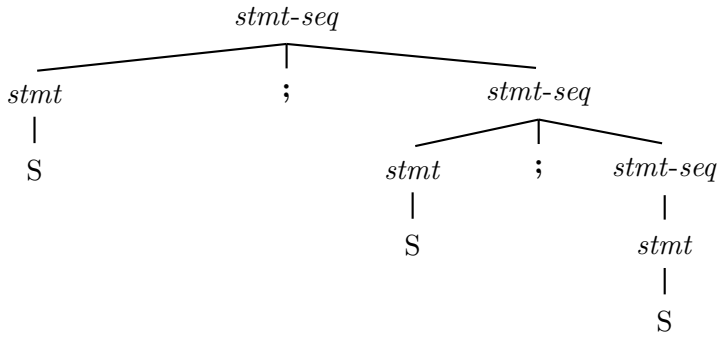
$$\begin{aligned} \textit{stmt-seq} &\rightarrow \textit{stmt-seq}\ \mathbf{;}\ \textit{stmt} \mid \textit{stmt} \\ \textit{stmt} &\rightarrow S \end{aligned}$$



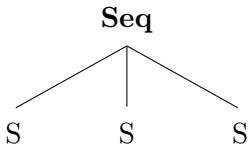
Non-essential ambiguity (2)

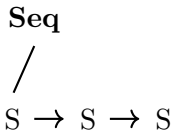
right-assoc representation instead

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



Possible AST representations





Dangling else

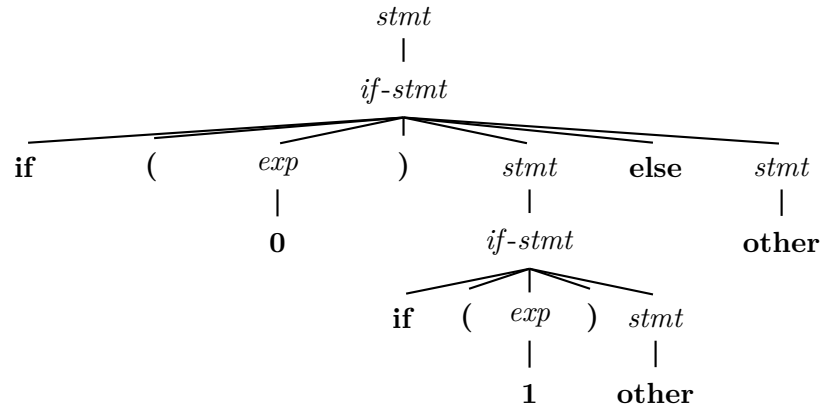
Nested if's

if (0) if (1) other else other

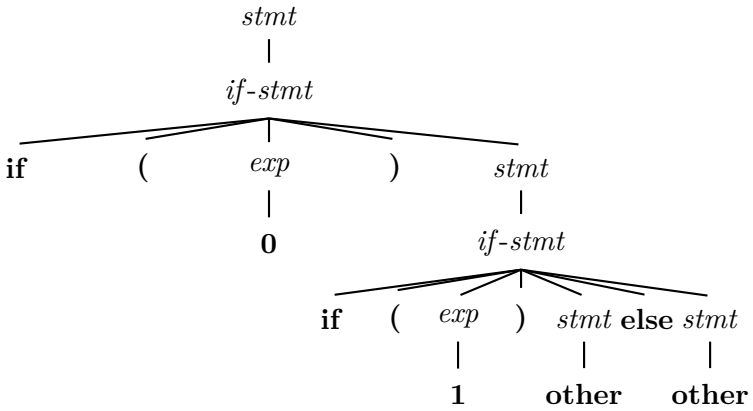
Remember grammar from equation (3.5):

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

Should it be like this . . .



. . . or like this



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

Unambiguous grammar

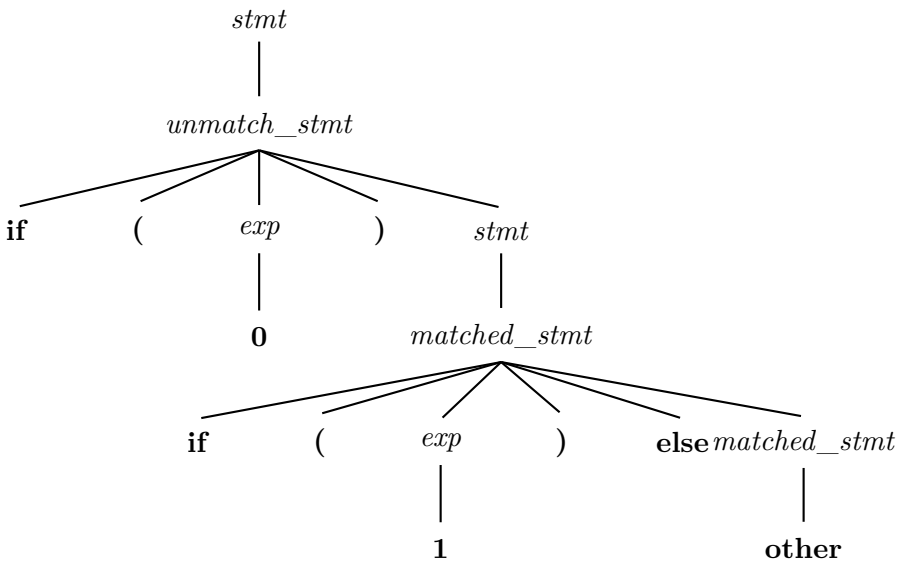
Grammar

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

- 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



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 (3.2)

The notion of *syntactic sugar* was mentioned earlier, when discussing sugared versions of regular expression. They were consequently called *extended* regular expressions. Syntactic sugar is a techical term. The process of removing syntactic sugar (typically by the parser when generating the abstract syntax tree), is called *desugaring*.

EBNF examples

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

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

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

greek letters: for non-terminals or terminals.

Some yacc style grammar

Let's also have a short look at how grammars are writting in parser generators. Here's an example code snippet. It sketches the syntax in yacc-style for an example involving simple arithmetical expressions. That example, in one form or the other, is almost unavoidable, when looking at such tools (and lectures like this one): the always illustrate they syntax and usage with a small expression example as warm-up. It's like the "hello-world" for yacc and friends.

Without going into details, we see additional information beyond the pure grammar. The grammar is on the "lower left corner" of the file. There is additional information before that part. The grammar as such is ambiguous; we have seen similar grammars in the lectures. It's made unabigiguous by specifying appropriate associativities and precedences. So one does not need to massage such grammars using the technique of precendence cascades, we have discussed earlier.

One thing that we don't have discussed yet is the "effect" of the grammar, or the *action part*. That's written, for each production or rule, on the right-hand side, in parentheses. That specifies what the parser should return, when processes a given production of the grammar during parsing.

In a standard setting, the action should give back an *abstract syntax tree*, which then is handed down to subsequent phases of a compiler, for instance, taking the AST and doing a type check on it before continuing even further. The expression example illustrates abstract syntax trees. Instead it uses the action part of the specification to do something simpler: it calculates the numerical value of the corresponding expression. In a way, the parser "executes" the code already during parsing. That's possible, because the grammar is so very simple. In more complex setting, doing computations is beyond the power of the parser resp. cannot be captured by (actions on a) context-free grammar. That's why further phases in a compiler are needed, until the resulting code is handed over to a execution platform. Compilers don't execute code themselves (at least not in general.)

The result of a action as far as productions for the expression non-terminal is concerned is thus a number. In one of the first lines, the corresponding type (in the implementing language) is defined, namely as `double`.

```
/* 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;         }
;
%%
```

3.4 Syntax of a "Tiny" language

The section is meant impressionistic mostly. We won't look in detail at the Tiny language anyway, but the oblig will be concerned with another quite small language (which we call "compila"). In this section, there is some hints of how to implement an AST data structure for TINYin C. Of course, languages like TINYand Compila share quite some commonalities (expressions, assignments, conditionals). That means, the sketchy hints of how AST could be designed for TINYcarry over to Compila (mutatis mutandis). However, the data structures used here are C, and most people won't use C, but Java (or some other language).

Later, we will give a "lecture" talking about the oblig; there we will say a bit more about the design of AST for the oblig in Java. Therefore, this section is not too central, and in the lecture, I will not waste much time on the C-AST implementation here.

BNF-grammar for TINY

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

BNF grammar for Compila20 (parts)

Here is another small language, namely *Compila20*, the language of the oblig, in the version of last year. It's not all of the grammar, just maybe 30%. The version from spring 2021 will be quite similar (we always make only mild syntactic massagings for each new round).

```
PROGRAM            -> "program" NAME "begin" [ DECL {";" DECL}]  "end"

DECL               -> VAR_DECL | PROC_DECL | REC_DECL

VAR_DECL           -> "var" NAME ":" TYPE [ ":=" EXP ] |  "var" NAME ":="  EXP

PROC_DECL          -> "procedure" NAME
                      "(" [ PARAMFIELD_DECL { "," PARAMFIELD_DECL } ] ")"
                      [ ":" TYPE ]
                      "begin" [[DECL{";" DECL}] "in"] STMT_LIST "end"
REC_DECL           -> "struct" NAME "{" [ PARAMFIELD_DECL
                                          {";" PARAMFIELD_DECL }]  "}"
PARAMFIELD_DECL     -> NAME ":" TYPE


STMT_LIST          -> [STMT {";" STMT}]
```

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 */
```

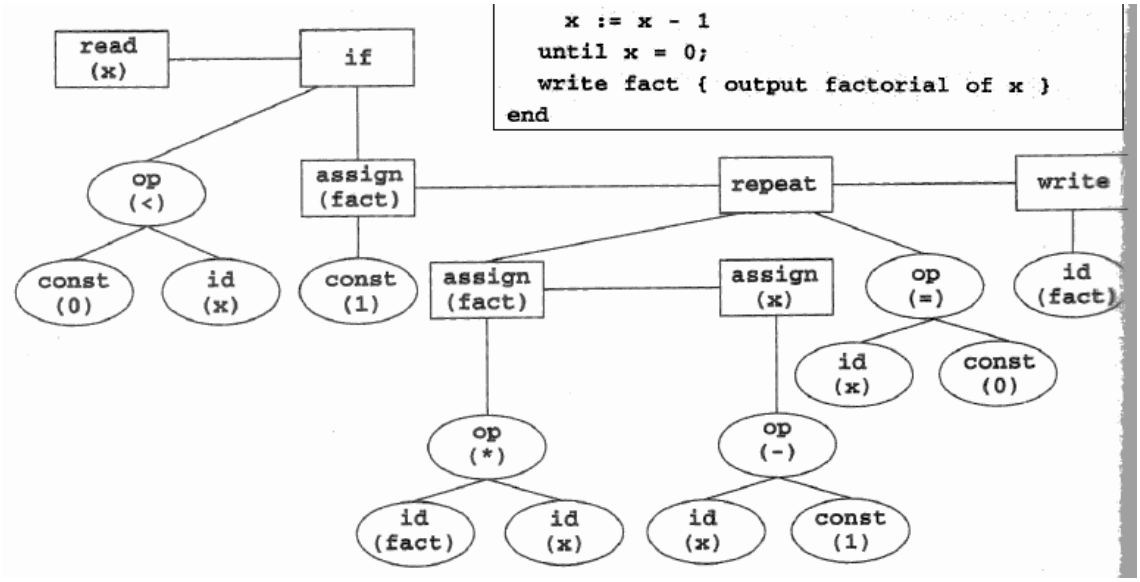
Comments on C-representation

- 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
```

Abstract syntax tree for a tiny program



Some questions about the Tiny grammar

- 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?

3.5 Chomsky hierarchy

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 $\alpha, \beta \ldots \in (\Sigma_T \cup \Sigma_N)^*$

Remark: Chomsky hierarchy

The rule format for type 3 languages (= regular languages) is also called *right-linear*. Alternatively, one can use *left-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.

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

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.

Index