Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

Chapter 3 Grammars

ms Error: pics

ms Error: Check if the pics are included (and/or the codes or the definitions). In the org-export.

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 [3, Section 3.1–3.2] (or [4, 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 current one covers the underlying concepts, namely context-free grammars. The following chapter will deal with the parsing process.

Context-free grammars resp. notations for context-free grammars play the same role for parsing as 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 (see Section 3.5). 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 aspects* 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, for practical reasons. the chapter about parsing. 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 in Section 3.4.

Figure 3.1: Bird's eye view of a parser

The input-output interface of a parser is shown in Figure 3.1.

The overall *task* of a parser is to *check* that the token sequence correspond to a *syntactically correct* program. if yes: yield a **tree** as intermediate representation for subsequent phases, if not, give *understandable* **error** message(s). The output of the parser is what is known as **abstract syntax tree.**

In the parser phase, we will encounter various kinds of trees, basically two. Since parsing and context free grammars are concerned with syntax, they are generically called syntax trees.

But there are **abstract syntax trees** and **concrete syntax trees** or **parse trees**. An abstract syntax tree is what is given back by the parser, as shown in Figure 3.1. Parse trees is a concept internal to the parser. A parse tree reflects the inner working of the parser, they way the parser constructs or derives at the conclusion that a program is syntactically correct. A program is syntactically correct, if there exists a parse tree for it. Since the parse tree reflects the derivation process in the context-free grammar that specifies the syntax, those trees are also called **derivation trees**

The abstract and the concrete syntax trees are not independent. Depending on design decisions of the designer of the language and of the parser, both kinds of trees may be very close to each other or slightly less so. As the name indicates, typically the concrete syntax tree contains more details than the abstract syntax tree. Since the abstract syntax tree is what the parser hands down to the subsequent phases of the compiler, it contains all necessary information for that, but ideally not more. In the same way, that the lexer filters out white space as irrelevant for the subsequent phases, the parser typically excludes some information in the parse tree, i.e., in the concrete syntax, and gives back a more cleaned-up tree version as output. Typically would be, to leave out parentheses or other grouping syntax, as the grouping information is already represented by the tree itself. You may remember the disucssion about the *syntax* of regular expression and whether parentheses are part of that syntax. The answer is, if we think of it as concrete syntax, parentheses are part if it, if we think of it as abstract syntax (basically thinking a regular expression is a tree), it's not needed.

The task of **parsing** is, 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*. 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, though, to be used for parsing, but good enough for lexing.

Derivations derive "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**. A successful derivation leads to tree-like representations.

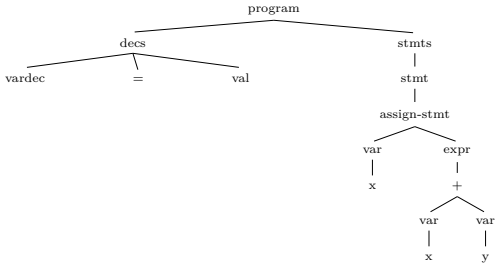


Figure 3.2: Sample syntax tree

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. Those notions are closely related and corresponding trees might all look similar to the tree shown.

Side remark 3.1.1 (Grammars for natural languages)**.** 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. See Figure 3.3. What the tree is exactly supposed to mean is not too important (*VP* and *NP* stand for verb-phrase and noun-phrase etc.). □

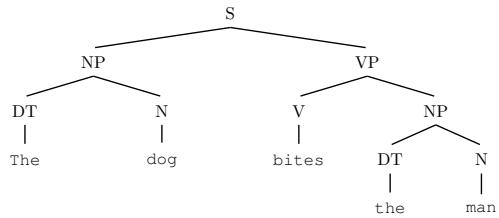


Figure 3.3: Natural language syntax tree

Let's *recap* also the **interface** between lexer and parser. Remember the task of the scanner is chopping up the input character stream, throw away white space, etc. and *classify* the pieces. One piece is called *lexeme*, a classified lexeme is a **token**, consisting of token name and a token value; some token classes may not have values. Sometimes we will use notations like $\langle$`integer`, "42"$\rangle$ for tokens, when referring to both the token name and the token value. We also often call the token name just token.

Remark 3.1.2 (Token (names) and terminals)**.** We said, that sometimes we use "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 identifiers and silently ignore the presence of the values.

> tokens = terminals of the grammar

That is in line with the *rule of thumb* mentioned in the introductory chapter about the classification task for the scanner. It stated *things that are treated identically should be classified into the same token.*

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. □

Remark 3.1.3 (On the use of notations)**.** Writing a compiler, especially a compiler front-end comprising a scanner and a parser, but also for later phases, is about implementing representation of syntactic structures. The script here and the slides 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 describe that language. We already faced the same when talking about 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" on a type writer, or even a manuscript, or a mixture, a typoscript with some extra manual scribbles here and there for symbols not supported by the typewriter. □

3.2 Context-free grammars and BNF notation

In this chapter and the one for parsing, we focus **context-free grammars**. So when mentioning grammars, we mean context-free ones. For lexing, we discussed regular expressions and regular languages. Languages, as defined earlier is a (typically infinite) set of words, and languages that can be described by regular expression are regular languages. Grammars, context-free or otherwise, like regular expression, is a formalism to unambiguously specify a language. Context-free grammars consequently described context-free languages. What is descibed by such grammar is a (programming's) language **syntax**. So such a grammar describes all **syntactically correct** programs of a given progamming language. In the context of a compiler, the alphabet of the context-free language is the set of *tokens* that the lexer procudes.

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

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

Note: a compiler might reject some syntactically correct programs, whose violations *cannot* be captured by CFGs. That is done by *subsequent* phases. 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.

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.

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

Productions are also called **rules**.

As mentioned, in the context of a parser, the terminal symbols correspond to tokens and are the basic building blocks of the syntax. Non-terminals, (e.g. representing an "expression", a "while-loop", a "conditional statement" are **compound** syntactic constructions. For syntax trees (in particular parse trees), the leaves consists of terminals, the inner nodes of non-terminals. A grammar is a notation for *generating languages* —context-free language, of course— by doing **derivations**. This generative aspect, specifying a language, is of course not the same as parsing. **Parsing** is the *inverse* problem: given a grammar and some sequence of terminals (or token in a parser): determine whether that sequence belongs to the grammar's language or not, i.e. whether the input is **syntactically correct** or not. The CFG is thus the specification of what the parser is expected to parse.

3.2.1 BNF notation

BNF is a popular & common format to write context-free languages. It's named after *pioneering* (seriously) work on Algol 60. It's a notation to write productions/rules + some extra meta-symbols for convenience and grouping

> **Backus-Naur form:** What regular expressions are for regular languages, is BNF for context-free languages.

Side remark 3.2.2. It seems like Peter Naur does not like to be associated with BNF ... He prefers the acronym to stand for Backus normal form. He is a Turing award winner and may be to Danemark what OJD is to Norway. John W. Backus, another Turing award winner, is also known for his involvement in Fortran, which is the *first* high-level programming language in actual widespread use. The BNF notation has been used in describing Algol 60 [1] (and ever since). □

Let's have a look at some example for illustration. There will be more than enough BNF-grammars in this and the following chapter, including different variations on capturing expressions

Example 3.2.3 (Expressions in BNF).

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

The notation here uses "$\to$" for productions or rules, i.e., pairs of the grammar, and " | " indicating alternatives for one non-terminal. For convention, we write terminals **boldface**, and non-terminals *italic*. Also simple math symbols like "+" and "(" are meant above as terminals. The start symbol here is *exp*. Unless stated otherwise, the start symbol is the first non-terminal mentioned in the grammar. This expression grammar consists of 6 productions/rules, 3 for *expr* and 3 for *op*. Instead of $\to$, one often finds the notation ::=. □

Remark 3.2.4 (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. □

There are different **variations** notation, BNF is not really "standardized" across books (and especially tools). The "classic" way two write it (as in Algol 60) may look as follows:

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

There exist also extended versions of BNF (EBNF), where one could write:

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

This variation also marks the terminals by "quoting" them. Note that the last grammar has to kinds of parenthes: the quoted ones which represent terminals, as well as *meta-symbols* used for grouping the grammar notation.

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.

As said, 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). BTW: Is it possible to use regular expressions as meta-language to describe regular expressions?

Example 3.2.5 (Different ways of writing the same grammar). Let's look at the following grammar consisting of written as 6 pairs from $\Sigma_N \times (\Sigma_N \cup \Sigma_T)^*$ (6 rules, 6 productions) , with "$\rightarrow$" as nice looking "separator":

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

The choice of non-terminals is of course irrelevant except for human readability. So we can write equivalently

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

Still, we count 6 productions. □

3.2.2 Grammars as language generators

A grammar represents the language it generates, i.e., the set of words that can be derived using it. **Deriving** a word is simple enough: start from start symbol. Pick a matching rule to rewrite the current word to a new one and repeat until the result contains *terminal* symbols, only.

To ease the presentation, let's fix some conventions

Convention 3.2.6. We will use $\alpha_1, \alpha_2, \beta \ldots$ for word of terminals and nonterminals, $w, v_1 \ldots$ for words of *terminals*, and $A, B' \ldots$ for a single non-terminal. We use also X for arbitrary grammar symbols, a terminal, or non-terminal, or $\boldsymbol{\epsilon}$.

We will use those conventions when speaking conceptually about grammars and their properties and will continue to use the conventions also in the chapter about parsing. In concrete examples, we will often deviate from the conventions, for instance, we continue to use *expr* for a non-terminal representing expression, not an upper-case letter, say E.

With these conventions, a **derivation** is of the following form:

$$S \Rightarrow^* \alpha = \quad \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2 \quad \Rightarrow^* w$$

The single derivation step in the middle applies a production $A \rightarrow \beta$ to the non-terminal A occurring in A. The end-result w is a word of *terminals*. The words in the middle contain non-terminals and possibly terminals. Words from $\Sigma^* = (\Sigma_N \cup \Sigma_T)^*$ are also called **sentential forms.**

We write $\alpha_1 \Rightarrow_G \alpha_2$ for one (rewrite) step in a given grammar G, resp. $\alpha_1 \Rightarrow \alpha_2$, when G is clear from the context. If we want to be specific, we can write for one step using rule n explicitly $\alpha_1 \Rightarrow_n \alpha_2$. The many-step derivation relation is written $\Rightarrow^*$, etc.

Definition 3.2.7 (Language of grammar G)**.**

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

Derivation is a **non-deterministic** process, i.e., a derivation makes choices what the next step is. One can distinguish two aspects of non-determinism here: 1) a sentential form contains (most often) more than one non-terminal, including the case that one particular non-terminal occurs more than once. 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.

Example 3.2.8 (Derivation). Take the expression grammar from Example 3.2.3 and assume the word

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

The following shows a derivation for that word:

$$\begin{array}{rcl}
\underline{\mathit{exp}} & \Rightarrow & \underline{\mathit{exp}}\ \mathit{op}\ \mathit{exp} \\
& \Rightarrow & (\underline{\mathit{exp}})\ \mathit{op}\ \mathit{exp} \\
& \Rightarrow & (\underline{\mathit{exp}}\ \mathit{op}\ \mathit{exp})\ \mathit{op}\ \mathit{exp} \\
& \Rightarrow & (\mathbf{n}\ \underline{\mathit{op}}\ \mathit{exp})\ \mathit{op}\ \mathit{exp} \\
& \Rightarrow & (\mathbf{n}-\underline{\mathit{exp}})\ \mathit{op}\ \mathit{exp} \\
& \Rightarrow & (\mathbf{n}-\mathbf{n})\underline{\mathit{op}}\ \mathit{exp} \\
& \Rightarrow & (\mathbf{n}-\mathbf{n})*\underline{\mathit{exp}} \\
& \Rightarrow & (\mathbf{n}-\mathbf{n})*\mathbf{n}
\end{array} \tag{3.5}$$

In the derivation, we use underlining to indicate the "place" where a rule is used, i.e., an *occurrence* of the non-terminal symbol which is being rewritten/expanded. The particular derivation from equation (3.5) is a so-called *leftmost* derivation, the following one from equation (3.6) is a *rightmost* derivation:[1]

$$\begin{array}{rcl}
\underline{\mathit{exp}} & \Rightarrow & \mathit{exp}\ \mathit{op}\ \underline{\mathit{exp}} \\
& \Rightarrow & \mathit{exp}\ \underline{\mathit{op}}\ \mathbf{n} \\
& \Rightarrow & \underline{\mathit{exp}}*\mathbf{n} \\
& \Rightarrow & (\mathit{exp}\ \mathit{op}\ \underline{\mathit{exp}})*\mathbf{n} \\
& \Rightarrow & (\mathit{exp}\ \underline{\mathit{op}}\ \mathbf{n})*\mathbf{n} \\
& \Rightarrow & (\underline{\mathit{exp}}-\mathbf{n})*\mathbf{n} \\
& \Rightarrow & (\mathbf{n}-\mathbf{n})*\mathbf{n}
\end{array} \tag{3.6}$$

There are also other ("mixed") derivations for the same word possible. □

[1] We'll come back to those concepts later, it will be important.

3.2.3 Some common-sense requirements for reasonable grammars

The format of grammars is pretty simple and allows to write all kinds of grammars, some obviously defective. People studied what is a good form for grammars and there are various so-called *normal forms* for grammars and methods how to transform a grammar into a decent form. We don't look into that and keep it informal. We just mention some obvious things one should avoid when writing a grammar, on the level of common sense.

For instance, all symbols, terminals and non-terminals, should occur in a some sentential derivable from the start symbol. Besides avoiding useless symbols, one might not have rules in the grammar that don't contribute to the language being derived or avoid redundancy in the productions. Also at least one word containing only non-terminals should be derivable. In other words, a grammar that specifies the empty language is not really useful ....

Here is an example of a silly grammar G (start-symbol A)

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

where $\mathcal{L}(G) = \emptyset$

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** or particular parsing techniques. 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 3.2.9 ("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'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 describes 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 to 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. □

3.3 Syntax trees, concrete and abstract

As mentioned variously, grammars and parsing is about the syntax of a language and it deals with trees. Consequently it's about syntax trees. We also mentioned that in a parser one distinguishes between parse-tree, also called concrete syntax trees on the one hand, and abstract syntax trees on the other. One also speaks about the contrete syntax of a language and its abstract syntax.

Let's start discussing **parse tree**. A derivation is a sequence of derivation steps, is it's a **linear** representation of the process generating a word or sentential form. The ultimate order of individual steps is irrelevant, however. In particular, the order of the derivation steps is *not needed* for subsequent phases. What matters is the **parse tree**, a tree structure that captures the *essence* of a derivation.

Let's look at some examples.

Example 3.3.1 (Parse tree). Consider the derivations from Example 3.2.8, for the expression grammar from Example 3.2.3. The parse tree for both derivations is shown in Figure 3.3.1.

Note: the numbers in the tree are *not* part of the parse tree. They are just shown here to indicate order of derivation. The concrete numbers are those for the right-most derivation from equation (3.6). □

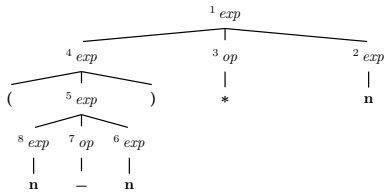
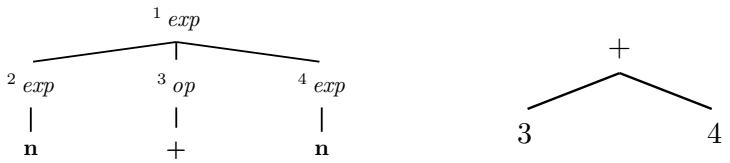


Figure 3.4: Parse tree (numbers for right-most derivation),

The parse trees often contains unnecessary details, for instance *parentheses*, used for grouping, or other unnessessary nodes. Parentheses for grouping are not necessary, the tree structure represents the intended grouping already. Also the tree may be represented more compactly. That is the case in particular, if the grammar is written "weirdly".

Why would one do that, writing up a grammar not in the clearest possible manner? If the grammar is used for parsing, not just for describing the language, then one has to take care of possible restrictions of the chosen parser technology. For instance, the expression grammar from equation (3.1) looked clean enough, but the grammar is *ambigous.* Ambiguous grammars are unwanted, and realistic parsers cannot handle such grammars properly. So, as clean as the grammar is, to be used in a parser, it needs to be reformulated so that it becomes at least unambigous. Depending on what parsing technique one uses, other aspects of a grammar are not acceptable for parsing, and sometimes one can massage the grammar or reformulate it to make it acceptable; sometimes the language is too complicated for a certain class of parsers. Such reformulations, if possible, typically don't increase the clarity of a grammar. We will see examples of that in the chapter about parsing, but also in this chapter when talking about ambiguity (see Section 3.4)

Example 3.3.2 (Parse tree and abstract syntax tree). Let's look at a very simple example, trees for the expression $3 + 4$, again using the grammar from Example 3.2.3. The tree from Figure 3.5a is a parse tree for the expression, actually the only parse tree for it. The numbers indicate that the tree has been derived by a left-most derivation (though again, the numbers are not officially part of the parse tree, as the order is irrelevant.) Figure



(a) Parse tree

(b) AST

Figure 3.5: Trees for $3 + 4$

3.5b sketches a possible abstract syntax tree, containing only three notes. Abstract syntax trees in a compiler are a *data structure*, an important intermediate representation. In the parse tree, we indicate the syntax structure omitting token values, for instance the token class **n** may contain lexeme or values like 3 and 4 in the example, and those of course need to be stored in the abstract syntax tree data structure.

Figure 3.5b shows an AST for the earlier parse tree for the expression $34 - 3 * 42$.[2]

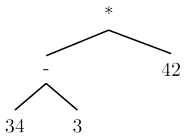


Figure 3.6: Plausible schematic AST (for the parse tree from Figure 3.4)

□

Comparing ASTs and CSTs, they are closely related. It's not surprising, the grammar describes the concrete syntax of a program, in terms of tokens. I.e., it's already no longer the source code as the whitespace and comments have typically been filtered out already and the lexems have been tokenized. But apart from that, the concrete syntax represents the code as is. And abstract syntax trees likewise represent the code, only abstracting away to some degree unneccessary details.

Parse trees are an important *conceptual* structure, to talk about grammars and derivations and later parsing. most likely not explicitly **implemented** in a parser as data structure, it's more that the parser, given a grammar, when doing a parse, follows it's run, the structure of a parse tree. When doing so, it builds up the abstract syntax tree as data structure in the host language. In the extreme case, when the designer choose to keep all the details of the parsing procedure, when the abstract syntax trees are equivalent to parse tree, the parsing process simply records the parse or derivation tree and hands that over as (not so) abstract syntax tree.

However, typically, the designer abstracts a bit from that. If the grammar is given (and if its unambiguos), the parse trees for a given input is fixed. However, for the abstract syntax tree, one has a little bit of freedom how to design then (and how to contretely implement them as data structure in the host-languate or meta-language). Thus, nodes as in ASTs like the one from Figure 3.5b are concrete data stuctures in the meta-language, for instance C-structs, instances of Java classes, or what ever is best suitable in the chosen host language. So the AST figure and similar ones are meant rather schematic, only.

Note also: we use 3 in the AST, where lexeme was `"3"`. At some point, the lexeme, which is a *string*, here `"3"`, is converted to a *number* in the meta-language. As already mentioned in the introductory chapter, that's typically already done by the lexer.

Now, what can one typically abstract away from a parse tree? Parentheses and similar **grouping** constructs are not needed in an AST. Then one could have constructs like conditionals. In concrete syntax it's one could write `if b then S1 else S2 endif`. Here a conditional that indicate the end by some special keyword `endif`. This end-marker is also some form of grouping. The fact that the syntax uses tokens like **if**, **then**, **else** and **endif** (corresponding to the lexemes `if`, `then` `else`, and `endif`, is not really relevant. It's just how a conditional is written down in **concrete** syntax. Abstractly, a

[2] For this expression and in the given grammar, it's not the only parse tree and consequently not the only abstract syntax tree. We come to that later.

conditional can be represented by a node in a tree with **three children nodes**, that's all that matters.

One can even invest a bit more in abstracting away from the parse tree. For instance, if one had a language that allows in concrete syntax two or more forms of loops, say a while- and a repeat-until loop. One form can easily express the other, so one would need not both. Still, the programmer may enjoy having both forms at disposal. Both therefore must be parseable and covered by the concrete syntax, but one could arrange for an abstract syntax tghat support only one, but transforms the other form away. One would call then the extra form **syntactic sugar**. Why would one get rid of such syntactic sugar in the AST? A motivation is, if one can already get rid of superflous constructs early on, one does not have to deal with them afterwards. So, one would not have to type check all variants, one would not have to generate code for all of them, etc.

Since we have just mentioned conditionals, let's have another look at those. 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.

Example 3.3.3 (Conditionals G_1). The following grammar in BNF, let's call it G_1, is a plausible formulation for conditional for a language that supports both two-armed and one armed conditional. The expression syntax is of course simplified, as irrelevant for us right now.

$$\begin{array}{rcl} \mathit{stmt} & \to & \mathit{if\text{-}stmt} \mid \textbf{other} \\ \mathit{if\text{-}stmt} & \to & \textbf{if (}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt} \\ & \mid & \textbf{if (}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ \mathit{exp} & \to & \textbf{0} \mid \textbf{1} \end{array} \tag{3.7}$$

The sequence of terminals

$$\textbf{if (}\ \textbf{0}\ \textbf{)}\ \textbf{other else other} \tag{3.8}$$

is syntactically correct, as evidenced by the parse tree from Figure 3.7. □

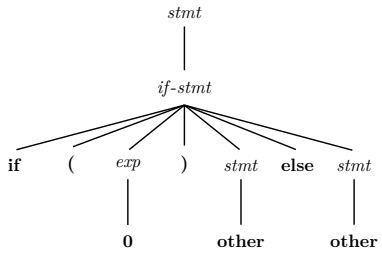


Figure 3.7: Parse tree

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*. As formulation of a grammar may be in a form unhelpful for parsers. That may also depend

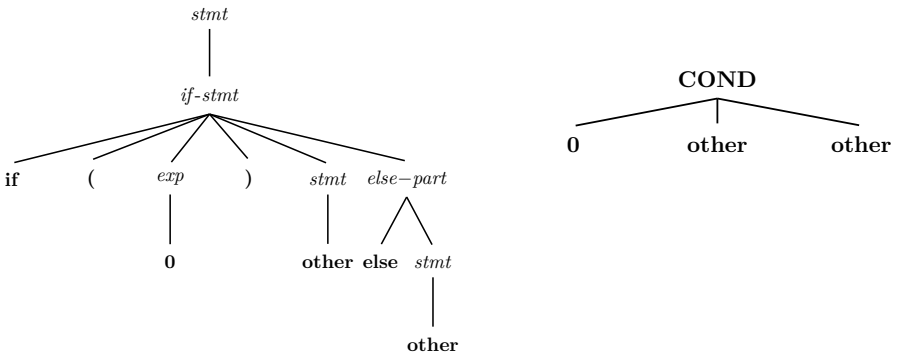
of the chosen style of parsers: some formulations pose problems for top-down parsers resp. for bottom-up parsers. Issues like that will be discussed in the chapter of parsing, here we are still covering grammars. In particular in connection with conditionals (which is a classic example): the chosen syntax 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, 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.

Example 3.3.4 (Another grammar for conditionals). Conditionals G_2

$$\begin{array}{rcl} \mathit{stmt} & \to & \mathit{if\text{-}stmt} \mid \textbf{other} \\ \mathit{if\text{-}stmt} & \to & \textbf{if (}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt}\ \mathit{else{-}part} \\ \mathit{else{-}part} & \to & \textbf{else}\ \mathit{stmt} \mid \epsilon \\ \mathit{exp} & \to & \textbf{0} \mid \textbf{1} \end{array} \tag{3.9}$$

As abbreviation, ϵ representes the **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.

See Figure 3.8 for a parse tree and an AST for that grammar. □



(a) Parse tree

(b) AST

Figure 3.8: Trees for G_2

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.

3.4 Ambiguity

Before we mentioned some "easy" conditions to avoid "silly" grammars, without going into detail. *Ambiguity* is more important and complex. A *grammar* is **ambiguous**, if there

exist *words 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 unambiguous. 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 can be captured by ambiguous grammars, but not by an unambigous one. In that case, one speaks of an context-free ambiguous **language** or says the language is *inherently* ambiguous. We concentrate on ambiguity of *grammars*, not languages.

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 associativity 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.

Figure 3.9: Tempus fugit ... (picture source: wikipedia)

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

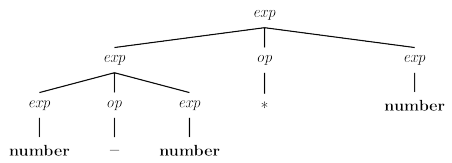
Definition 3.4.1 (Ambiguous grammar)**.** A grammar is *ambiguous* if there exists a word with *two different* parse trees.

We have seen examples of ambiguity before.

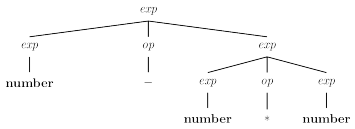
Example 3.4.2. Remember the expression grammar from Example 3.2.3 and equation (3.1) and consider

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

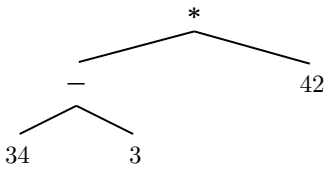
Obviously, there are different parse trees for that, and consequently different ASTs. And of course, the different AST **mean** different things, i.e., represent different numbers. Resp. code ulimately generated from the two trees will result in different outcomes.



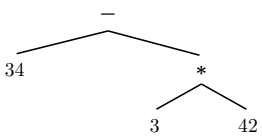
(a) Parse tree (1)



(b) Parse tree (2)T



(c) AST (1)



(d) AST (2)

Figure 3.10: Different parsing of $31 - 3 * 42$

□

Remark 3.4.3 (Different meaning)**.** The issue of different meanings 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 diffffernet 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. □

3.4.1 Precendence & associativity

One way to make a grammar unambiguous (or less ambiguous) is to use the concepts of **precedence** and **associativity.** See Table 3.1 for conventions for standard binary operators in infix-notation. Exponentiation $a \uparrow b$ is written in standard math texts as a^b.

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

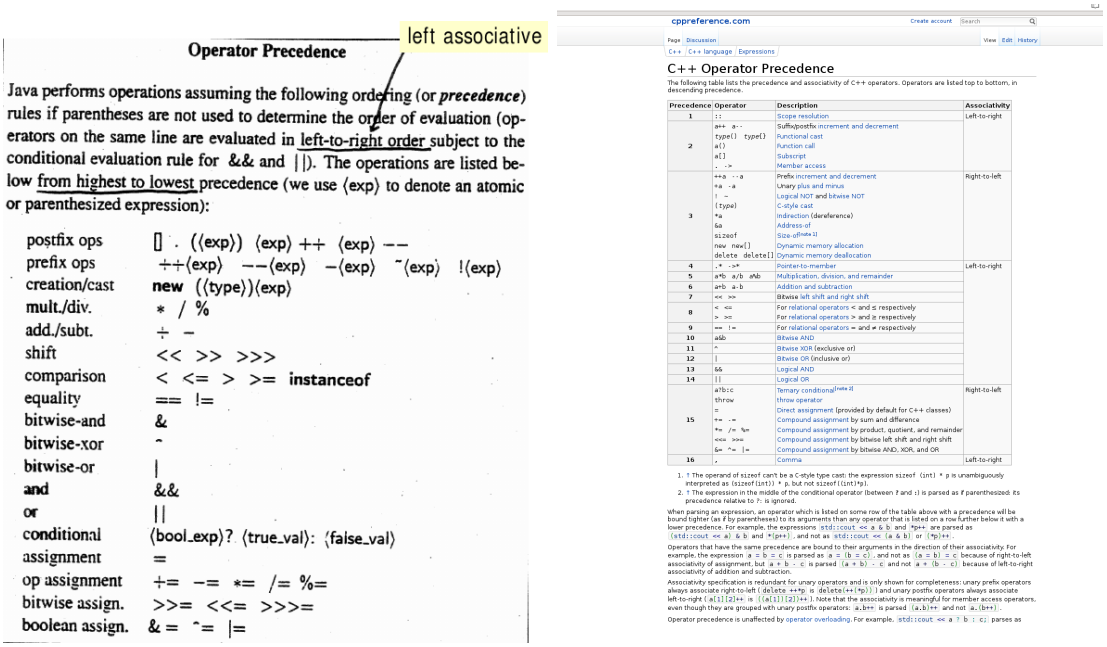
Table 3.1: Standard precedences and associativities of mathematical binary operators

With these conversions understood, one can transform the following expression into another one using paretheses, to make the grouping explicit.

$$\begin{aligned} 5 + 3/5 \times 2 + 4 \uparrow 2 \uparrow 3 \quad &= \\ 5 + 3/5 \times 2 + 4^{2^3} \quad &= \\ (5 + ((3/5 \times 2)) + (4^{(2^3)})) \ . & \end{aligned} \tag{3.10}$$

Those concepts work mostly fine for *binary* operators in infix, but usually also for unary ones (postfix or prefix).

Here are two examples in actual programming languages. The scan from Figure 3.11a is taken from an edition of the book "Java in a nutshell", the one from Figure 3.11b covering C++ is clipped from the net



(a) Java

(b) C++

Figure 3.11: Associativity and precendences

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? At least in general, reformulating a given grammar into an unambigous one is often possible, resp. figuring out that there's not chance for it, as the language is inherently ambiguous.

3.4.2 Unambiguity without imposing explicit associativity and precedence

There exists also a recipe to (try to) remove ambiguity by reformulating the grammar. The method sketched here, known as **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, as well). 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. Let's start with a simple example, illustrating associativity of expressions.

Example 3.4.4 (Assiciativity of expressions). Let's take expressions built with $+$ and $-$ as binary operators (represented by *addop*). To have them **left-associative**, we can introduce a separate non-terminal, here called *term*, and write the grammar in a **left-recursive** manner:

$$exp \to exp\ addop\ term \mid term \tag{3.11}$$

More precisely, the first production is left-recursive over the non-terminal *exp*. Analogously, right-associativity is achieved by right-recursion (see equation 3.12), and for a non-associative (and ambiguous) representation, one would not need the additional non-terminal *term*, if one had expressions only using plus and minus.

$$exp \to term\ addop\ exp \mid term \tag{3.12}$$

If, besides addition and substruction, one also supports multiplication, for instance, it's no longer a question of associativity alone, then also precendence enters the picture. In this case, one will introduce a non-terminal for each precedence level. The two levels could be called expressions and terms, and if the *addop* is intended to be non-associative, the corresponding part of the grammar would look as follows:

$$exp \to term\ addop\ term \mid term \tag{3.13}$$

Of course, with minus as possible operator, a non-associative formulation is bad idea. □

Let's continue the example by adding multiplication and covering precedence.

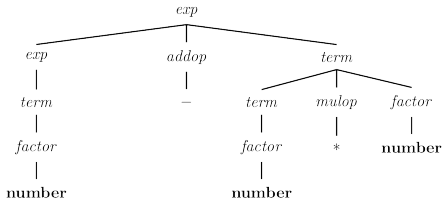
Example 3.4.5 (Factors and terms). As hinted at in the previous example, we need to introduce separate extra *non-terminals* to cover the precedence levels. There, besides expression, there are terms and factors (which are traditional names for those). Concerning associativity, both operator levels are defined as left-associative, as the corresponding productions for *exp* and for *term* are left-recursive.

Figure 3.12 shows two parse-trees, Figure 3.12a illustrates precedence (of multiplication over $-$), Figure 3.12b illustrating left-associativity (of minus).

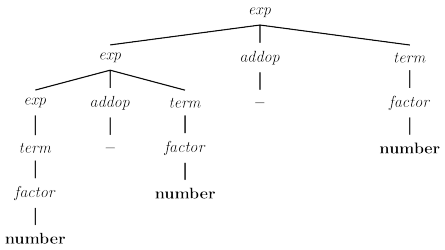
$$\begin{aligned} exp &\to exp\ addop\ term \mid term \\ addop &\to + \mid - \\ term &\to term\ mulop\ factor \mid factor \\ mulop &\to * \\ factor &\to (\ exp\ ) \mid \textbf{number} \end{aligned} \tag{3.14}$$

□

Sometimes, whether operators associates to the left or the right is irrelevant. One speaks of **non-essential ambiguity**.



(a) $34 - 3 * 42$



(b) $34 - 3 - 42$

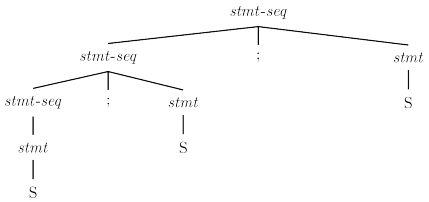
Figure 3.12: Factors and terms

Example 3.4.6 (Non-essential ambiguity). Consider the following two possible ways to specify sequential composition of statements, one with the corresponding operator associating to the left, in the other formulation associating to the right.

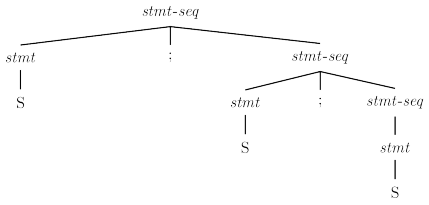
$$\begin{array}{rcl} \mathit{stmts} & \to & \mathit{stmts}\,;\,\mathit{stmt} \\ & \mid & \mathit{stmt} \\ \mathit{stmt} & \to & S \end{array}$$

$$\begin{array}{rcl} \mathit{stmts} & \to & \mathit{stmt}\,;\,\mathit{stmts} \\ & \mid & \mathit{stmt} \\ \mathit{stmt} & \to & S \end{array}$$

For



(a) left-assoc.

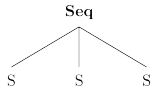


(b) right-assoc.

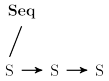
Figure 3.13: Sequential composition

sequential composition, the associativity does not semantically matter. Indeed, the parser could turn both parse trees or concrete syntax trees of Figure 3.13 into the same abstract syntax tree. Figure 3.14 sketches two possible representations; the second one is meant as represing a linked list, where each statement points to its successor.

When using a "non-associative" representation in the AST, later phases would not even see the difference in programs parsed under a left- or under right-associative regime, or between $S;\{S;S\}$ and $\{S;S\};S$, if the syntax support grouping.



(a)



(b)

Figure 3.14: Possible abstract syntax trees

□

So in a situation of inessential ambiguitiy, it does not matter which parse-tree is actually chosen by a parse. Does it mean, one should use an ambiguous grammar. It may be easier to formulate, if one cannot rely on a mechanism that allows to specify associativity without need to come up with some precedence cascade oneself.

Actually, it may not be a good idea. Anyway, we are currently talking about properties of grammars and their parse-tree, not about *parsing* and the parser. Practical parsers cannot handle ambiguous grammars and they are not aware if an ambiguous situation is essential or not. Being responsible for checking syntactic well-formedness and nothing else, it's not the business of the parser of having an opinion about semantical issues (like if the ambiguity is essential or not).

What a parser would do instead is the following. It would stumble upon the situation but would not reject the input. After all, if ambiguite caused the trouble, it is *not* so much a defect on the input, but rather dubious as far as the grammar is concerned. So it would issue a *warning,* but based on its internal working, make a decision concerning the precedence or the associativity, if those cause the ambiguous situation.

One could of course say, who cares. If the ambiguity is inessential, it does not matter what decision the parser does. That's true, of course. On the other hand, a grammar that causes the parser generator to issues warnings is not a good sign.

Actually, warnings are not only caused by ambiguitity or situations like the one we discussed, that could be solved by specifying precedences and associativities. It's guaranteed that an ambiguous grammar will produce at least one warning, resp. will exhibit **conflicts** as they will be called for parsers. But a parser generating tool will exhibit conflicts also for *unambiguous grammara*. Indeed, the parse generator won't try to determine whether a grammar is ambiguous or not, that's undecidable anyway. The warnings reflect more the limitations of a given form of parser technology: situations where it cannot decide which way do to proceed, and then proceeding one way, but issuing a warning that there is an alternative not pursued.

As said, which grammar is accepted without warnings and which not depends depends on the chosen form of parsers (bottom-up vs. top-down) and the amount of so-called look-ahead. So it's common when developing a grammar as basis for parsing, one has to massage the grammar for a while, and during that fine-tuning, one has to struggle which quite an amount of warnings in different parts of the syntax. It also implies that the grammar specifying the syntax of a language for humans is *not* the one used for parsing. That for instance will be the case in the **oblig:** we spefify the syntax by a grammar, but no effort is made to make it unambiguous or that it can be accepted by parser generator without warnings

The grammatical massage may make the grammar not more beautiful in the same sense that putting in more non-terminals into the simple expression grammar to realize a precedence cascade did not make the expression grammar more beautiful.

One could try to find an excuse for a grammar with parse conflict. One could explain away some conflicts as symptom of inessential ambiguity. In other cases an excuse could be, that based on the knowledge how the parser technology works, the desginer use sure which decision will be taken by the tool, and that decision is the one designer can live with.

Both arguments are not too convincing, a grammar without conflicts is a sign of a well-desiged parser. It's more convincing to explicitly specify in the grammar "times is left-associative and of higher priority than plus " than to assure

> "you will notice a conflict, but the parser generator, as far as I know, works in such a way that times is left-associative and of higher-priority than plus, don't worry about the warning. Also the other warnings are features, not bugs.".

The following is actually an example of essential ambiguity. It's a well-known example, and it's an example not involving binary operators. Binary operators are well understood, and they can be tamed in a clear manner by specifying their associativity. It's also an example we will revisit in the parsing chapter, to see which decisions bottom-up parsers do in an ambiguous situation involving *dangling elses*, as the phenomenon is called.

Example 3.4.7 (Dangling else). Let's consider the grammar from Example 3.3.3, in particular the conditionals from equation (3.7).

The grammar is ambiguous, as can be seen on the **nested-if**-expression **Nested if's** in the following expression:

$$\textbf{if (0) if (1) other else other} \tag{3.15}$$

which can be parse with the parse trees from Figure 3.15 The grammar is The common

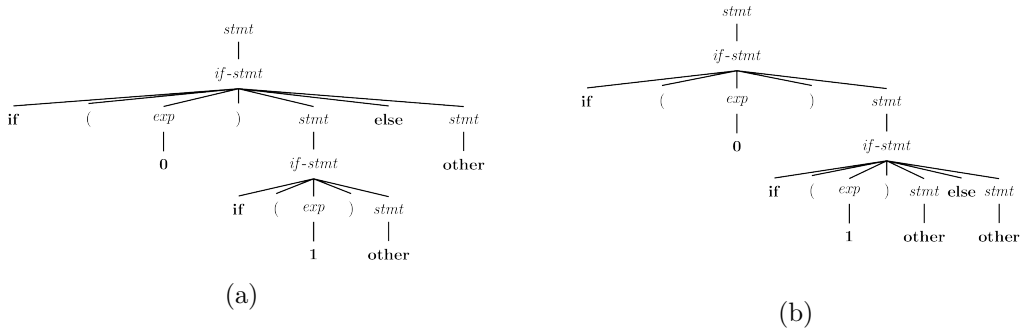


Figure 3.15: Two possible parse-trees

convention in languages that support such forms of conditionals is to connect **else** to closest "free" (= dangling) occurrence of a conditional. It means, the parse tree from Figure 3.15b is the preferred one. □

We mentioned earlier, that often, when facing ambiguity or conflict, one can massage the grammar into an alternative formulation, without that defects. The precendence cascade is an example of such a massage. For the dangling-else situation, the next grammar leads to a unambigous representation, and one with the intended "closest-dangling-conditional" preference.

Example 3.4.8 (Conditionals: unambiguous grammar). An ambiguous grammar for the conditionals-grammar is the following.

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

It enforces to never have an unmatched statement inside a matched one. As we mentioned, massaging a grammar to be acceptable for a parse often does not make the grammar simpler or more beatiful, and this one is no exception ...

So actually, one would seldomly use that formulation, at least not for human consumption. Instead one would use the ambiguous one, with extra instructions to connect each **else** to closest free **if**.

And there are other alternatives: To side-step the whole issue, one may use a **different syntax**, for example, a *mandatory* **else**-branch, or require **endif**. □

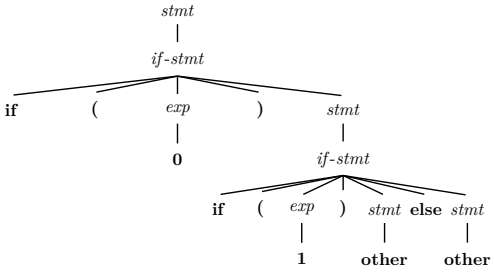


Figure 3.16: CST (dangling else)

3.4.3 Adding sugar: extended BNF

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

syntactic sugar

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*.

3.4.4 EBNF examples

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

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

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

greek letters: for non-terminals or terminals.

3.4.5 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;         }
;
%%
```

3.5 Chomsky hierarchy

The chapter here is concerned with context-free languages. It's not the only form of languages, and context-free grammars not the only form of grammars. For completeness sake, let's shortly place context-free languages and regular language, another class of languages we encountered, inside the so-called **Chomsky hierarchy**[2]. It is an important classification of (formal) languages, due to the linguist Noam Chomsky. The classification is also known as Chomsky-Schützenberger-hierarchy.

The hierarchy contains 4 levels of languages,[3], from type 0 languages (the most expressive

[3]Of course, with a famous concept like that, people came up with refinements but the 4 mentioned levels

ones) to type 3 languages, the most restricted ones (see Table 3.2). The hierarchy is based on the form of grammars used to describe the languages at a level. So there is a most general form of grammars, the one for the type-0 languages, and the other language classes are given by imposing more restrictions on the rule format.

Languages are defined as sets of words over a given alphabet, as we did already for regular languages, and as for the context-free grammars, one splits the symbols into terminal and non-terminal symbols, Σ_T and Σ_N.

As convention, we use letters $a, b, \ldots \in \Sigma_T$ for terminals, $A, B, \ldots \in \Sigma_N$ non-terminals, and for general words $\alpha, \beta \ldots \in (\Sigma_T \cup \Sigma_N)^*$.

The most general of grammars has basically no restrictions on the two words.

Being a hierarchy mean, each type n language is ahierachy is inclusive and strict

	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

Table 3.2: Chomsky hierarchy

The levels of the hierarchy are related to machine models that generate resp. recognize them. For regular languages, we discussed the connection to finite-state automata at some length. For context-free languages, there is a automata model called *push-down* automata. That are basically finite-state automata equipped with an additional memory in the form of a *stack*. The stack is unbounded, so the machine model is no longer finite-state. In the same way that finite-state machines play a role in implementing scanners, PDA's can be as the foundation of *parsers*. However, programming languages typically don't use context-free grammars in their full generality, for instance, ambiguous grammars are a no-no. Instead one restructs to more restricted forms of grammars and we will also not introduce push-down automata as concept (though the parsing process will included working with a stack, either explicitly or by doing a recursion-based procedure for parsing.

Also for context-sensitive languages at level 1, there exists a machine or automaton model, but we won't deal with it ...

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.

remain central.

Bibliography

[1] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samuelson, K., Wegstein, B. V. J. H., van Wijngaarden, A., and Woodger, M. (1963). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:1–17.

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

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

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

Index