Topic of the day: Grammars
Ch. 3 in K. C. Louden

My slide style:

- Sometimes more text than one can possibly digest at the lecture.
- This slide style is meant for easy revision before the exam.
- I will sometimes skip slides with much text (but you should indeed read them afterwards!)
- The textbook is still the most important source of information for the exam (except when we specifically stress the opposite!)
Where are we now? - Ch. 3, 4 og 5:

Pre-processor
- Macros
- Conditional compilation
- Files

Scanner
- Partition the text into a sequence of lexemes
- Can be described by regular expressions

Parser
- Find the structure of the program
- Can be described by a (BNF) grammar

Checker
- Checks usage against declarations
- Checks types in expressions

Code generator
- Usually some type of optimizer, for efficient execution

Symbol table

**Tools:**
- Grammars.
- Lex
- Flex
- Top-down and bottom-up parsing.
- **Tools:** Antlr, Yacc, Bison, CUP, etc.
- Attributte grammars
- More or less systematic techniques and methods
Simplified sketch of what the parser is doing

Checking that the sequence of tokens makes up a syntactically correct program.

If an error occurs: Give an understandable error message

The program structure is transformed to a tree of objects (e.g., Java objects).

Syntax-tree for a given program:
"Abstract" or "concrete"?
This tree is typically abstract.
Overview – Ch. 3 (basic properties of grammars)

- Context-free grammars and BNF grammars
- How does a grammar define a language
- Parsing trees and abstract syntax trees
- Ambiguous and unambiguous grammars
- Extended BNF (EBNF) and syntax diagrams
- Example:
  - The Tiny language in the textbook

AND:

- Before we move to Ch. 4, we shall look at some general properties of grammars.
- In the textbook this stuff is found in between other discussions in Ch. 4 and 5, but we look at it before we start at Ch. 4.
What do we get from the scanner?

We get a sequence of such packagers:
(where comments, line shifts etc. are removed):

- Often the unit as a whole is called a «token».
- During parsing it is also called a «terminal symbol»
- The classification of tokens into categories is not always obvious:
  Should e.g. «+» and «*» be in the same category

Token (or "token category")
The «type» of the symbol
- name, integer, arith-op, "if",
...

Lexeme: How the symbol
looks in the program text, or
a code showing which
lexeme it is in the given
category
About BNF grammars:

- A grammar is using a number of symbols to describe the sentences («programs»). The set of these forms a «language».
  - **Terminal symbols** – The symbols (tokens) we get from the scanner, or often only their category: name, integer, "if", …
  - **Nonterminal symbols** – Names of constructions in the language: While-statement, assignment, class declaration, expression, …
    - Plus: The **start symbol**: The start for producing a legal sentence
  - **Meta symbols** – Symbols or characters used for forming the grammar rules (or «productions»)

- **Note**: From a grammar it is quite easy to form different legal sentences in the language
  - The parsing problem is the **opposite**: Given a sentence. Can it be «derived» from the grammar? And if so: How?

- A grammar specifies a language by a number of «rules» describing the legal constructions in the language.
  - These rules are also called «productions»
Contextfree grammars: BNF forms, with variations

- BNF = Backus (Fortran) – Naur (Algol) – Form
- The basic version used in the textbook:
  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp op exp} \mid (\text{exp}) \mid \text{number} \\
  \text{op} & \rightarrow + \mid - \mid *
  \end{align*}
  \]
  - Meta symbol: “\(\rightarrow\)” (Read: “Can have the form”), “\(|\)” (Read: “or”)
  - Non-terminal: \(\text{exp}, \text{op}\) (Non-terminal symbol is too long!)
  - Terminal symbol: \(\text{number}, (,), *, +, -\)
  - Start symbol: Ex.: \(\text{expression}, \text{program}\) (Always a non-terminal)

- Another traditional way (The Algol report):
  \[
  \begin{align*}
  <\text{exp}> & ::= <\text{exp}> <\text{op}> <\text{exp}> \mid ( <\text{exp}> ) \mid \text{NUMBER} \\
  <\text{op}> & ::= + \mid - \mid *
  \end{align*}
  \]

- Extended BNF (EBNF), and yet another BNF style:
  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp} \ (“+” \mid “-” \mid “*”) \text{exp} \mid “(“ \text{exp} “)” \mid \text{number}
  \end{align*}
  \]
Different ways of writing a grammar

- In the textbook this form of BNF is used as the most basic one:
  
  $\text{exp} \rightarrow \text{exp}\ \text{op}\ \text{exp}$
  
  $\text{exp} \rightarrow (\ \text{exp}\ )$
  
  $\text{exp} \rightarrow \text{num}ber$
  
  $\text{op} \rightarrow +$
  
  $\text{op} \rightarrow -$  
  
  $\text{op} \rightarrow *$
  
  6 rules or "productiones"

- As short as possible (but we still say that there are 6 rules/productions):
  
  $E \rightarrow E\ O\ E\ |\ (\ E\ )\ |\ n$
  
  $O \rightarrow +\ |\ -\ |\ *$
Derivation (here *leftmost derivation*) of:

\[(\text{number} - \text{number}) \times \text{number}\]

**Intermediate forms**

<table>
<thead>
<tr>
<th>Rule (production) used</th>
<th>Intermediate forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\exp \rightarrow \exp \op \exp]</td>
<td>(1) [\exp \Rightarrow \exp \op \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow (\exp) \op \exp]</td>
<td>(2) [\Rightarrow (\exp) \op \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow \exp \op \exp \op \exp]</td>
<td>(3) [\Rightarrow (\exp \op \exp) \op \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow \text{number} \op \exp \op \exp]</td>
<td>(4) [\Rightarrow (\text{number} \op \exp) \op \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow \text{number} - \exp \op \exp]</td>
<td>(5) [\Rightarrow (\text{number} - \exp) \op \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow \text{number} - \text{number} \op \exp]</td>
<td>(6) [\Rightarrow (\text{number} - \text{number}) \op \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow \text{number} \op \exp]</td>
<td>(7) [\Rightarrow (\text{number} - \text{number}) \times \exp]</td>
</tr>
<tr>
<td>[\exp \rightarrow \text{number}]</td>
<td>(8) [\Rightarrow (\text{number} - \text{number}) \times \text{number}]</td>
</tr>
</tbody>
</table>

*Leftmost derivation:* Always choose the leftmost nonterminal for further derivation. We are finished when we only have terminal symbols.

The language of G:

\[L(G) = \{ s \mid \exp \Rightarrow^* s \} \]
**Derivation** (here rightmost derivation) of:

\[(\text{number} - \text{number}) \times \text{number}\]

<table>
<thead>
<tr>
<th>Intermediate forms</th>
<th>Rule used</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) (\text{exp} \Rightarrow \text{exp op exp})</td>
<td>([\text{exp} \rightarrow \text{exp op exp}])</td>
</tr>
<tr>
<td>(2) (\Rightarrow \text{exp op number})</td>
<td>([\text{exp} \rightarrow \text{number}])</td>
</tr>
<tr>
<td>(3) (\Rightarrow \text{exp} \times \text{number})</td>
<td>([\text{op} \rightarrow \times])</td>
</tr>
<tr>
<td>(4) (\Rightarrow (\text{exp}) \times \text{number})</td>
<td>([\text{exp} \rightarrow (\text{exp})])</td>
</tr>
<tr>
<td>(5) (\Rightarrow (\text{exp op exp}) \times \text{number})</td>
<td>([\text{exp} \rightarrow \text{exp op exp}])</td>
</tr>
<tr>
<td>(6) (\Rightarrow (\text{exp op number}) \times \text{number})</td>
<td>([\text{exp} \rightarrow \text{number}])</td>
</tr>
<tr>
<td>(7) (\Rightarrow (\text{exp} - \text{number}) \times \text{number})</td>
<td>([\text{op} \rightarrow \text{-}])</td>
</tr>
<tr>
<td>(8) (\Rightarrow (\text{number} - \text{number}) \times \text{number})</td>
<td>([\text{exp} \rightarrow \text{number}])</td>
</tr>
</tbody>
</table>

**Rightmost derivation:** Always choose the rightmost nonterminal for further derivation. We are finished when we only have terminal symbols.

There are a lot of other ways to derive the same sentence from the grammar.
Obvious requirements to a reasonable grammar

All terminals and non-terminals:
- Must occur in some sentence derivable from the start symbol

From all non-terminals
- one must be able to derive sentences with only terminal symbols

Eks:
- \( A \rightarrow B \ x \)
- \( B \rightarrow A \ y \)
- \( C \rightarrow z \)

- C or z cannot occur in any string derivable from the start symbol A
- We cannot derive a string from A containing only terminal symbols

So this is a really bad grammar
Parse-trees (often called *concrete syntax-trees*)

- Exp: \( \text{num} + \text{num} \)
- NB: A representation that is independent of the derivation order
- The numbers indicate a leftmost derivation

If we disregard the numbers, all derivations give the same tree

- Exp: \((\text{num} - \text{num}) \times \text{num}\)
- The numbers indicate a rightmost derivation
- If we disregard the numbers, all derivations give the same tree
Abstract syntax trees (AST)

We remove the “unnecessary” nodes from the tree, namely those that stems from “syntactic sugar” in the language. We are then left with the essential meaning of the program. Exactly which elements should be retained in the AST must be made precise for each language. During a compilation the compiler usually builds an AST for the whole program.
A strange rule for boolean expressions

More parse trees (concrete syntax trees)

G1:

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
        | if ( exp ) statement else statement
exp → 0 | 1
```

Sentence:

```
if (0) other else other
```
Another grammar G2 for if-sentences:

G2:

\[
\begin{align*}
\text{statement} & \rightarrow \text{if-stmt} \mid \text{other} \\
\text{if-stmt} & \rightarrow \text{if} \ ( \text{exp} \ ) \text{statement} \ \text{else-part} \\
\text{else-part} & \rightarrow \text{else} \ \text{statement} \mid \varepsilon \\
\text{exp} & \rightarrow 0 \mid 1 
\end{align*}
\]

A possible abstract syntax tree (AST) for the statement to the right:

If there are no else-part, it can be represented by a null-pointer (e.g. in Java).
Ambiguous grammars.
Trees for the sentence: \( n - n * n \)

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp \ op \ exp} \mid (\text{exp}) \mid \text{number} \\
\text{op} & \rightarrow + \mid - \mid *
\end{align*}
\]

G is *ambiguous* if there exists a sentence in \( L(G) \) that can be obtained by two different parse trees.

The different parse trees can often indicate two quite different meanings of the sentence.
Use of *precedence* and *associativity* to make an ambiguous grammar unambiguous

- We look at the following ambiguous grammar (see previous slide)

  \[
  \begin{align*}
  exp & \rightarrow exp \ op \ exp \ | \ ( \ exp \ ) \ | \ number \\
  op & \rightarrow + \ | \ - \ | \ *
  \end{align*}
  \]

- In addition, we give the following rules for precedence and associativity. This will lead to only one legal syntax tree.
  
  +, – low, left associative
  
  *, / higher, left ass.
  
  ↑, highest, right ass.

\[
3 + 5 / 3 * 2 + 4 \uparrow 2 \uparrow 3
\]

Means: \((3 + ((5 / 3) * 2)) + (4 \uparrow (2 \uparrow 3))\)

- This is simple to understand for binary operations, but it «usually» also works fine for *unary* postfix or prefix operators.
How to make ambiguous grammars unambiguous without using precedence and associativity?

- **Presedence for an operator**
  - Some operations «bind» more strongly than others (* stronger than +)
  - Change: Make one extra non-terminal for each «precedence level», and the corresponding set of operations. E.g. (term {+, -}, factor {*}, ...)

- **Associativity for operators in the grammar:**
  - Left associativity: Is fixed by making all grammar rules for these operations «left recursive»:
    \[ exp \rightarrow exp \ op \ exp \mid ( \ exp \ ) \mid \text{number} \]
    \[ op \rightarrow + \mid - \mid * \]
  - Right associativity: Is fixed likewise, but the other way around
  - No associativity is fixed like this:
    \[ exp \rightarrow \text{term} \ op \ \text{term} \mid \text{term} \]

If we only have +, -, and * (which are all left associative) we’ll get the grammar to the right.
The resulting unambiguous grammar

An extra non-terminal for each precedence level

The order here corresponds to the actual associativity.

For students with INF4130:
The question: «given a BNF-grammar, is it unambiguous?» is undecidable
Precendence and associativity in Java

Operator Precedence

Java performs operations assuming the following ordering (or **precedence**) rules if parentheses are not used to determine the order of evaluation (operators on the same line are evaluated in left-to-right order subject to the conditional evaluation rule for `&&` and `||`). The operations are listed below from highest to lowest precedence (we use `(exp)` to denote an atomic or parenthesized expression):

postfix ops
- `[]` (tuple, list, array), etc.
- `(exp)`, `++ (exp)`, `-- (exp)`

prefix ops
- `++(exp)` and `--(exp)`, `- `(exp)`
- `~(exp)`, `!(exp)`

creation/cast
- `new ((type))(exp)`

mult./div.
- `* / %`

add./subt.
- `+ -`

shift
- `<<` (left shift)
- `>>` (right shift)
- `>>>` (unsigned right shift)

comparison
- `< <= > >=` (less than, less than or equal, greater than, greater than or equal)
- `instanceof`

equality
- `== !=`

bitwise-and
- `&`

bitwise-xor
- `~`

bitwise-or
- `|`

and
- `&&`

or
- `||`

conditional
- `(bool_exp)? (true_val): (false_val)`

assignment
- `=`

op assignment
- `+= -= *= /= %=` (assignment operators)

bitwise assign.
- `>>= <<= >>>=`

boolean assign.
- `&= ^= |=`
Non-essential ambiguity

stm-seq → stm-seq; stm | stm
stm-seq → stm; stm-seq | stm
stm → s

Can as well be represented like this:

seq or: seq

s/ s/ s s s s s
The «dangling else» problem

Problem: To which *if-sentence* does the *else* belong to?

Like this: ( )

```plaintext
if (0) if (1) other else other
```

or like this: ( )

The grammar below are unambiguous, see next slide

```plaintext
statement → if-stmt | other
if-stmt → if ( exp ) statement
         | if ( exp ) statement else statement
exp → 0 | 1
```
Two trees for the same sentence

Sentence:

if (0) if (1) other else other

The most used rule:
Connect an else to the closest «free» occurrence of if.
Unambiguous grammar for if-statement, giving the same solution as the rule on the previous slide.

Unambiguous Grammar:

\[
\begin{align*}
\text{statement} & \rightarrow \text{matched-stmt} \mid \text{unmatched-stmt} \\
\text{matched-stmt} & \rightarrow \text{if} (\text{exp}) \text{matched-stmt} \text{else} \text{matched-stmt} \mid \text{other} \\
\text{unmatched-stmt} & \rightarrow \text{if} (\text{exp}) \text{statement} \\
& \quad \mid \text{if} (\text{exp}) \text{matched-stmt} \text{else} \text{unmatched-stmt}
\end{align*}
\]

\[
\text{exp} \rightarrow 0 \mid 1
\]

**Idea:** Never have an unmatched inside a matched

- **matched-stmt:**
  Is itself containing an else, and can therefore not be connected to a later else

- **unmatched-stmt:**
  Have no else, and can therefore be connected to a later else

**Question:** How can we know that this grammar can generate all legal statements according to the previous ambiguous grammar?

This complex grammar is seldom used. Instead, we use the ambiguous one, with the extra rule:

Connect each else to the closest free if
Extended BNF (EBNF)

Idea: One can generally use «regular expressions» on the right hand side of a rule/production, and these can freely contain non-terminals.

Much used: \( \alpha \star \) is written: \{\alpha\} \quad \alpha \) is here a string of terminals and non-terminals
\( \alpha ? \) is written: \[\alpha\]

Example:

\[
exp \rightarrow exp \left( \text{“+” | “-” | “*”} \right) exp \mid \text{“(“ exp “)”)” \mid \text{number}
\]

Meta symbol \quad Teminal symbol

\[
A \rightarrow A \alpha \mid \beta \quad \text{can be written: } A \rightarrow \beta \{\alpha\}
\]

\[
A \rightarrow \alpha \ A \mid \beta \quad \text{can be written: } A \rightarrow \{\alpha\} \beta
\]

\[
\text{stm-seq} \rightarrow \text{stm} \{; \text{stm}\} \quad \text{or} \quad \text{stm-seq} \rightarrow \{\text{stm;}\} \text{stm}
\]

\[
\text{if-stm} \rightarrow \text{if (expr) stm [ else stm ]}
\]

NB: For some grammar tools you have to use basic BNF!
Syntax-diagrams

Much like non-deterministic automata for regular languages. However, this time they can be «recursive».

\[
\text{factor} \rightarrow ( \text{exp} ) | \text{number}
\]

\[
\text{factor} \quad \rightarrow \quad ( \quad \rightarrow \quad \text{exp} \quad \rightarrow \quad )
\]

\[
\text{factor} \quad \rightarrow \quad \text{number}
\]

**NB:** «Ambiguity» and similar concepts cannot easily be defined with this notation.
$A \rightarrow \{ B \}$

$A \rightarrow [ B ]$
BNF-grammar:

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp } \text{addop } \text{term} \mid \text{term} \\
\text{addop} & \rightarrow + \mid - \\
\text{term} & \rightarrow \text{term } \text{mulop } \text{factor} \mid \text{factor} \\
\text{mulop} & \rightarrow * \\
\text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
\end{align*}
\]

EBNF-grammar for the same language. This is more directly connected to the syntax diagrams:

\[
\begin{align*}
\text{exp} & \rightarrow \text{term} \{ \text{addop } \text{term} \} \\
\text{addop} & \rightarrow + \mid - \\
\text{term} & \rightarrow \text{factor} \{ \text{mulop } \text{factor} \} \\
\text{mulop} & \rightarrow * \\
\text{factor} & \rightarrow ( \text{exp} ) \mid \text{number}
\end{align*}
\]

NB: Here *associativity* (but not precedence) must be specified separately
The Chomsky hierarchy

- **Type 0 - language**
  - Unrestricted prods.: 
  \[ \alpha \rightarrow \beta, \quad \alpha \neq \varepsilon \quad (\alpha \text{ is not empty}) \]
  - Related to Turing machines

- **Type 1 - language**
  - Context sensitive prods.:
  \[ \beta A \gamma \rightarrow \beta \alpha \gamma \]
  - Related to “Linearly restricted automata”

- **Type 2 - language** (Used for normal syntax specification)
  - Contextfree prods. (E)BNF:
  \[ A \rightarrow \alpha \]
  - Related to “Automata with stack memory”

- **Type 3 - language** (Used to describe lexemes/tokens)
  - Regular languages:
    - Regular expressions
    - NFA
    - DFA
  - All productions have the form:
    \[ A \rightarrow B a \quad \text{or} \quad A \rightarrow a \]
  - or all productions have the form:
    \[ A \rightarrow a B \quad \text{or} \quad A \rightarrow a \]
  - Related to finite automata
Why not have one big grammar that says everything about the language: lexemes, syntax and semantics?

- Couldn’t we use one big grammar to define the whole language??
  - It is e.g. easy to use BNF to describe the lexemes for the scanner
  - And may be we could also write the BNF grammar so that it also takes care of the semantic rules?

- We are not doing this because:
  - Such a grammar would be unnecessarily large
  - It turns out to be impossible to describe all the semantic rules by a BNF grammar (or by a context-free grammar in general)
    - It is e.g. impossible to ensure that all variables are declared with BNF
  - It is much more convenient to:
    - Use simple regular grammars (or expressions) to describe the lexemes
    - Use BNF to describe the structural form (syntax) of the language
    - Describe the semantic rules in precise English (and check them with a program that looks at the syntax tree).

- About syntax: One often has to rewrite a grammar for making it suitable for a certain tool.
  - Many different grammars can correctly describe the syntax of a language.
BNF-grammar for TINY

program → stmt-sequence
stmt-sequence → stmt-sequence ; statement | statement
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence end
           | if exp then stmt-sequence else stmt-sequence end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp comparison-op simple-exp | simple-exp
comparison-op → < | =
simple-exp → simple-exp addop term | term
addop → + | −
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number | identifier
The structure of the nodes in syntax trees for programs in TINY

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 */
} TreeNode;
Node structure in C for programs in Tiny

NOTE:
Here the same node structure is used to represent all different statement nodes.

The structure of the nodes may become more natural if we use an OO language, where we have classes and subclasses.

Then we’ll have a hierarchy of classes that represent the different statements.

We’ll use this technique in a later mandatory assignments.

If, Repeat, Assign, Read, Write - is drawn:

Op, Const, id - is drawn:

Is only used for nodes with kind «expr.»
Abstract syntax tree for a Tiny-program

Question: Indicate a class hierarchy in an OO-language (e.g. Java?) for better representing the different statement types.
Some questions about the TINY grammar
Is later given as an assignment

- Is the grammar unambiguous?
- How can we change it so that TINY allows empty statements?
- What if we want semicolons in between and not after the statements?
- What is the precedence and associativity of the different operators?

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