FORMAL METHODS II:
FORMAL LANGUAGES

September 27, 2013
Rolf Pfeifer
Rudolf M. Füchslin
**Grammars and Languages**

Languages

- **Natural Languages**
  - Natural language
    - + High expressiveness
    - + No extra learning
    - - Ambiguity
    - - Vagueness
    - - Longish style
    - - Consistency hard to check

- **Formal Languages**
  - Formal language
    - + Well defined syntax
    - + Unambiguous semantics
    - + Can be processed by computer
    - + Large problems can be solved
    - - High learning effort
    - - Limited expressiveness
    - - Low acceptance
Natural and Formal Languages

• Natural languages are evolved.
• Formal languages are constructed.

• Humans tend to design in a modular manner:
  • The resulting structures are comprehensible.
  • This comprehensibility supports rational planning, and extendibility.
• Evolution has no rational:
  • Solution only need to be effective not necessarily comprehensible.
  • Evolution can only perform optimizations which immediately yield a benefit, but not e.g. "platform strategy" which deliberately facilitates future extensions. The evolutionary approach yields efficient and yet robust solutions.
Evolution of Natural Languages
Words can be categorized.
Natural Languages Have Structure

There are higher order structures.
Natural Languages Have Structure

Sentences are represented as tree-like structures.
Syntax and Syntax Trees

- Tree-like structures can be constructed by replacement rules.

```
I  →  Clause Punc
Clause  →  Subject Verb Object
Subject  →  Determ Noun
Object  →  Determ Noun
Verb  →  chews
Determ  →  the | a
Noun  →  dog | bone
Punc  →  .
```

| indicates a choice. Example: A Noun can be replaced either by dog or by bone.
Syntax and Syntax Trees

```
I → Clause Punc
Clause → Subject Verb Object
Subject → Determ Noun
Object → Determ Noun
Verb → chews
Determ → the | a
Noun → dog | bone
Punc → .
```

“The dog chews a bone.”
“A dog chews the bone.”
“A bone chews a dog.”

1. | 1
2. Clause Punc
3. Clause.
4. Subject Verb Object .
5. Determ Noun Verb Object .
6. the Noun Verb Object.
7. the bone Verb Object.
8. the bone Verb Determ Noun.
9. the bone Verb a Noun.
10. the bone Verb a dog.
11. the bone chews a dog.

....
Syntax Trees – Informal Description

- We have a set of symbols, some red, some green.
- We have a start symbol “I”.
- Replacement rules give substitutions for red symbols either by other red symbols or green symbols.
- Green symbols cannot be replaced.
- One proceeds, until no red symbols are left.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Clause Punc</td>
<td>Clause Punc</td>
<td>Clause</td>
<td>Subject Verb Object</td>
<td>the Noun Verb Object</td>
<td>the bone Verb Object</td>
<td>the bone Verb Determ Noun</td>
<td>the bone Verb a Noun</td>
<td>the bone Verb a dog</td>
<td>the bone chews a dog</td>
</tr>
<tr>
<td>Clause</td>
<td>Subject Verb Object</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subject</td>
<td>Determ Noun</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Determ Noun</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verb</td>
<td>chews</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Determ</td>
<td>the</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Noun</td>
<td>dog</td>
<td>bone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Punc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Syntax Trees – Informal Description

1. I
2. Clause Punc
3. Clause.
4. Subject Verb Object .
5. Determ Noun Verb Object .
6. the Noun Verb Object.
7. the bone Verb Object.
8. the bone Verb Determ Noun.
9. the bone Verb a Noun.
10. the bone Verb a dog.
11. the bone chews a dog.

1. I
2. Clause Punc
3. Clause.
4. Subject Verb Object .
5. Subject Verb Determ Noun .
6. Subject Verb Determ dog.
7. Determ Noun Verb Determ dog.
8. the Noun Verb Determ dog.
9. the Noun Verb a dog.
10. the bone Verb a dog.
11. the bone chews a dog.

Several sequences of applications of replacement rules lead to the same sentence / syntax tree.
Recursive Rules

- Subjects/Objects may consist many adjectives: „The little young white dog ....“
- Possible rules to handle such constructs:

<table>
<thead>
<tr>
<th>Subject</th>
<th>Determ ANoun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Determ ANoun</td>
</tr>
<tr>
<td>ANoun</td>
<td>Noun</td>
</tr>
<tr>
<td>AC</td>
<td>little</td>
</tr>
<tr>
<td>Noun</td>
<td>dog</td>
</tr>
</tbody>
</table>

The more adjectives, the more cumbersome rules!
Recursive Rules

- To keep rule tables small, recursive rules can be defined:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>→ Determ Noun</td>
</tr>
<tr>
<td>Object</td>
<td>→ Determ Noun</td>
</tr>
<tr>
<td>Noun</td>
<td>→ Adjective Noun</td>
</tr>
<tr>
<td>Adjective</td>
<td>→ little</td>
</tr>
</tbody>
</table>
Recursive Rules

To keep rule tables small, recursive rules can be defined:

<table>
<thead>
<tr>
<th>Subject</th>
<th>→</th>
<th>Determ Noun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>→</td>
<td>Determ Noun</td>
</tr>
<tr>
<td>Noun</td>
<td>→</td>
<td>Adjective Noun</td>
</tr>
<tr>
<td>Adjective</td>
<td>→</td>
<td>little</td>
</tr>
</tbody>
</table>

Problem: These rules allow constructs such as “the white white little white white white white white dog.”
The theory of formal languages investigates sets of structured sequences of characters (P. Rechenberg).

“Structure” will be precisely defined.

The “structure” in the theory of formal languages is deterministic \(\Rightarrow\) no stochastic element.
Strings

There are strings and strings:
• “dkjfhd Asdf Nyuh IkjuGty ^45 dfd @EcYTG”, probably a random string.
• “ABABABABABABABABABABABABABABABABABABABABABABABABABABAB” a neatly ordered string with local structure.
• “ABAABAAAAABAAAAABAAAAAAABAAAAAAAAB” a string with simple but non-local structure.
• “It’s Friday morning.” a string with semantic meaning.
• “Str prst zkrz krk” a Czech proverb.
Structure and Meaning

- Using increasingly complex formal means, increasingly complex notions of “Structure” can be defined.
- “Meaning” is a more elusive concept.
- Open debate: Can “Meaning” be explained by “structure”?
Definition: Alphabet

An alphabet $\Sigma$ is a finite set; its elements are called characters.
Characters can be letters, but also symbols or even words.

$$\Sigma_1 = \{a, b, c\}, \quad \Sigma_2 = \{0, 1\}, \quad \Sigma_3 = \{\uparrow, \downarrow\}$$

$$\Sigma_4 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

$$\Sigma_5 = \{\text{'is'}, \text{'sunny'}, \text{'rainy'}, \text{'the'}, \text{'today'}, \text{'tomorrow'}, \text{'weather'}, \text{'yesterday'}\}$$
Definition: Strings

- A string is an ordered sequence of characters.
- Some usual abbreviations are:

\[ \varepsilon : \]
\[ a^0 = \varepsilon, a^n = aa^{n-1} \quad (n > 0) \]
\[ a^+ = \{a^n, n > 0\} \]
\[ a^* = \{a^n, n \geq 0\} \]
\[ \alpha = c_1c_2...c_n \leftrightarrow \alpha^R = c_nc_{n-1}...c_1 \quad \text{Reflection of a string}\ \alpha \]
\[ |\alpha| = \text{length of } \alpha, \quad |\varepsilon| = 0 \]

the empty string

Exponentiation of a character in V

Reflection of a string
Definition: Kleene-Star

- Given an alphabet $\Sigma$. The **Kleene-star of $\Sigma$, $\Sigma^*$**, is the set of all finite concatenations of elements of $\Sigma$ plus the empty string $\varepsilon$ (which is not in $\Sigma$).
- $\Sigma^*$ can be defined recursively:
  1. **Basis**: $\varepsilon \in \Sigma^*$
  2. **Recursive step**: If $\alpha \in \Sigma^*$ and $c \in \Sigma$, then $c\alpha \in \Sigma^*$.
  3. **Closure**: $\beta \in \Sigma^*$ if it can be produced by a **finite application** of the recursive step.
Definition: Formal Language

- A formal language $L$ over an alphabet $\Sigma$ is a subset of $\Sigma^*$: $L \subseteq \Sigma^*$.

- Some trivial languages:
  - $L = \emptyset$: the empty language
  - $L = \{\varepsilon\}$: the language consisting of the empty string.
  - $L = \Sigma^*$: The „Allsprache“.

- Elements of a language are often called
  - „Sentences“ in theoretical computer science
  - „Words“ in mathematics
Definition: Operations on Languages

Languages are sets. Consequently, they can be subject to set operations (L, M are both languages over V):

- The union of two languages:
  \[ L \cup M = \{ \alpha \in \Sigma^* \mid (\alpha \in L) \lor (\alpha \in M) \} \]

- The intersection of two languages:
  \[ L \cap M = \{ \alpha \in \Sigma^* \mid (\alpha \in L) \land (\alpha \in M) \} \]

- The concatenation of two languages:
  \[ LM = \{ \alpha\beta \mid (\alpha \in L) \land (\beta \in M) \} \]
How To Define Languages?

The sets have to be described somehow:

• One can simply enumerate all sentences.
• Languages can be generated by **grammars**.
• A language can be defined by giving an **automaton** that recognizes its elements.
• The elements of a language can be given by a specification of properties: \( L = \{ \alpha : \alpha \in \Sigma^* \land P(\alpha) \} \). \( P(\alpha) \) is a proposition about \( \alpha \) (The difference to the automaton is that specifying properties and specifying how they are checked is not the same thing).
Comment

- Languages can be generated by **grammars**.
- A language can be defined by giving an **automaton** that recognizes its elements.

Native speakers, when checking the correctness of a sentence, usually just check „whether they would it say the same way“, means they try out, whether they can reconstruct a sentence (verification by reproduction).

Only when one starts to learn a language, one analyzes a sentence and checks its compatibility with abstract rules (whether a memorized grammar automaton accepts it).
Definition: A grammar $G$ is defined as a quadruple

$$G = (\Sigma, V, P, S)$$

with

- $\Sigma$: a finite set of terminal symbols (alphabet)
- $V$: a finite set of non-terminal symbols (variables) usually with the condition $(\Sigma \cap V) = \emptyset$.
- $P$: a finite set of production rules.
- $S \in V$: the start symbol.
Production Rules

- Production rules are basically rules for substituting substrings of a given string.
- The most general form of production rules is structured like this:

\[ \alpha \rightarrow \beta \]

\( \alpha \) has the form \( \omega_L \gamma \omega_R \)

\( \omega_L, \omega_R \in (\Sigma \cup V)^* \)

\( \gamma \in V \)

\( \beta \in (\Sigma \cup V)^* \)

- Further requirements on the structure of production rules define types of languages.
- Note: the \( \gamma \) guarantees that there is at least one non-terminal symbol on the left hand side of a production rule.
- Note: The Kleene- star contains by definition the empty string \( \Rightarrow \omega_{R,L} \) may be empty.
A grammar is a finite set of production rules.

A grammar $G$ generates a language $L(G)$.

$L$ can have infinitely many sequences.

The rules of $G$ have to be applied until no non-terminal symbol is present anymore.

Restrictions on production rules define classes of grammars.

A sequence of rule applications is called a derivation.
Example 2.2. Let us consider the following grammar:

- Alphabet $\Sigma$: \{0, 1, +\}
- Variables $V$: \{S, N\}
- Production rules $P$:
  - $S \rightarrow N | N + S$
  - $N \rightarrow 0 | 1 | NN$
- Start symbol: $S$

Example of derivation:

- $S \Rightarrow N + S$
- $\Rightarrow NN + S$
- $\Rightarrow 1N + S$
- $\Rightarrow 10 + S$
- $\Rightarrow 10 + N + S$
- $\Rightarrow 10 + N + N$
- $\Rightarrow 10 + N + 1$
- $\Rightarrow 10 + 0 + 1$
Definition: A grammar tree is a tree where each link corresponds to the application of one particular production rule, and where the leaves represent the elements of the language. The path from the root element to a leaf corresponds to the derivation of that element.
(Note: A grammar tree may be infinite).
Definition: Grammar Tree

\[ \Sigma : \{0,1,+\} \]

\[ V : \{S,N\} \]

Start symbol: \( S \)

\[ S \rightarrow N \mid N + S \]

\[ N \rightarrow 0 \mid 1 \mid NN \]

A syntax tree has characters as leaves, a grammar tree whole sentences.
Grammars and Automata

We analyze specific languages as formal languages partly because there are automata recognizing their elements ➔ file globbing, regular expressions, parsing programs …
Types of Languages

- Languages can be categorized according to the structure of their production rules.
- The American philosopher and linguist Noam Chomsky introduced a categorification which turned out to be easy to use and represents fundamental differences between specific languages.
Definition: Regular Grammars

The production rules of a right-regular grammar have the form:

\[
A \rightarrow \alpha \\
A \rightarrow \beta B \\
A, B \in V \quad \alpha, \beta \in \Sigma^*
\]

Of course, there can be many rules of these types, depending on the size of \( V \) and \( \Sigma \).
Regular Grammars: Comments

- Informal description: Regular grammars produce strings by appending.
- From a physical point of view, they produce discrete time series, where the future is, up to well-defined choices, determined by the past. Once made, a choice cannot be taken back.
- A **regular language** is a language produced by a regular grammar.
**Example 2.4.** An archetypical regular language is:

\[ L = \{ a^m b^n \mid m, n \geq 0 \} \]

\( L \) is the language of all strings over the alphabet \( \Sigma = \{a, b\} \) where all the \( a \)'s precede the \( b \)'s: \( L = \{\epsilon, a, b, aa, ab, bb, aaa, aab, \ldots\} \).

**Example 2.3.** An example of a regular grammar \( G \) with \( \Sigma = \{a, b\} \), \( V = \{S, A\} \), consists of the following set of rules \( P \):

\[
\begin{align*}
S & \rightarrow aS | A \\
A & \rightarrow bA | \epsilon
\end{align*}
\]
Regular Grammars: Examples

- Regular grammars seem to produce sequences based on local rules.
- Is there a regular grammar for binary strings with a number of „1“ being a multiple of three?
Regular Grammars: Examples

- Regular grammars seem to produce sequences based on local rules.
- Is there a regular grammar for binary strings with a number of "1" being a multiple of three?

\[
\begin{align*}
S &\rightarrow 0S \mid 1A \\
A &\rightarrow 0A \mid 1B \\
B &\rightarrow 0B \mid 1C \\
C &\rightarrow 0C \mid 1A \mid \varepsilon
\end{align*}
\]
Application: Regular Expressions

- **Common task:** Search for words with a common structure, move files with names obeying a certain rule, etc.

- ruedi.jpg
- christina.jpg
- kaspar.jpg
- julia.jpg
- finance_jan.dat
- finance_feb.dat
- travel_feb.dat
- travel_oct.dat
- finance_march.dat

Psychoanalyse
psychoanalytisch
Psychoanalytikerin
Psychoanalytisch
Psychotherapie
Application: Regular Expressions

- **Regular expression mechanism** accept a syntax describing patterns and contain a mechanism that filters matching sequences.
- Originally, the underlying mechanism were based on regular languages.
- In the next lecture, we learn why: strings produced by a regular grammar can by recognized by a simple automata model, which is easy to implement.
- **WARNING:** There are several, slightly different “flavors” of regexp: file globbing à la grep is different from POSIX regexp used in vi, emacs, awk, lex.
### Most Common POSIX regexp Operators

- Operators are metacharacters. If they have to be used literally, use a backslash `\`.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Stands for</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>At least one occurrence of preceding symbol</td>
</tr>
<tr>
<td>*</td>
<td>Zero, one or more occurrence of preceding symbol (NOT like in file globbing)</td>
</tr>
<tr>
<td>?</td>
<td>Zero or one occurrence of preceding symbol</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>Grouping</td>
</tr>
<tr>
<td>\</td>
<td>The next symbol literally (escaping)</td>
</tr>
<tr>
<td>.</td>
<td>Any single character (NOT like in file globbing)</td>
</tr>
<tr>
<td>Regular Expression</td>
<td>Elements of the Language</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>abc</td>
<td>abc</td>
</tr>
<tr>
<td>a*bc</td>
<td>bc, abc, aabc, aaabc, aaaaabc, ...</td>
</tr>
<tr>
<td>go+gle</td>
<td>gogle, google, gooogle, ...</td>
</tr>
<tr>
<td>pfeiff?er</td>
<td>pfeifer, pfeiffer</td>
</tr>
<tr>
<td>pf(a</td>
<td>e)ifer</td>
</tr>
</tbody>
</table>

Used in many UNIX/LINUX – like environments. 
NOTE: e.g. grep uses file globbing.
Grammars and Regexps

What is the regular grammar behind \texttt{lec+tu*\texttt{r}(e|a)}

<table>
<thead>
<tr>
<th>Operator</th>
<th>Stands for</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>At least one occurrence of preceding symbol</td>
</tr>
<tr>
<td>*</td>
<td>Zero , one or more occurrence of preceding symbol (NOT like in file globbing)</td>
</tr>
<tr>
<td>?</td>
<td>Zero or one occurrence of preceding symbol</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>Grouping</td>
</tr>
<tr>
<td>\</td>
<td>The next symbol literally (escaping)</td>
</tr>
<tr>
<td>.</td>
<td>Any single character (NOT like in file globbing)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
S & \rightarrow \text{lec}A \\
A & \rightarrow \text{c}A | \text{t}B \\
B & \rightarrow \text{u}B | C \\
C & \rightarrow D | rD \\
D & \rightarrow e | a
\end{align*}
\]
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Matches...</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>any single character (wildcard)</td>
</tr>
<tr>
<td>[ ]</td>
<td>a single character that is contained within the brackets</td>
</tr>
<tr>
<td>[^ ]</td>
<td>a single character that is <em>not</em> contained within the brackets</td>
</tr>
<tr>
<td>\n</td>
<td>a digit from 0 to 9</td>
</tr>
<tr>
<td>^</td>
<td>start of the string</td>
</tr>
<tr>
<td>$</td>
<td>end of the string</td>
</tr>
<tr>
<td>*</td>
<td>zero, one or more copies of the preceding symbol (or expression)</td>
</tr>
<tr>
<td>{x, y}</td>
<td>at least x and not more than y copies of the preceding symbol (or expression)</td>
</tr>
</tbody>
</table>
BTW: Globbing not equal to regexping!

- With UNIX file globbing, "*.jpg" produces a list of files all ending with jpg.
- Regexp uses for the same ".*jpg".
Context Free Languages
Context Free Languages: Motivation

- A program:

```c
main() {
    printf("Hello world!");
    return;
}
```

- A bigger program:

```c
main() {
    printf("Hello world!\n");
    printf("It’s Friday.\n");
    return;
}
```
Context Free Languages: Motivation

• Programming: Replacing SLOT’s by content.
  func0(SLOT_A, SLOT_A)
  → func0(func1(SLOT_B), SLOT_A))
  → func0(func1(x), SLOT_A)
  → func0(func1(x), func2(SLOT_A))
  → ....

• SLOTs can be occupied by other expressions, **independent of the surrounding**.

• There may be several types of SLOTs.
Definition: Context Free Grammar

A context-free language is constructed by a context free grammar, which is defined by having replacement rules of the form

\[ \alpha \rightarrow \beta \]

\[ \alpha \in V \]

\[ \beta \in (\Sigma \cup V)^* \]
Example 2.5. A simple context-free grammar is

\[ S \rightarrow aSb \mid \epsilon \]

It generates the context-free language

\[ L = \{ a^n b^n \mid n \geq 0 \} \]

which consists of all strings that contain some number of \( a \)'s followed by the same number of \( b \)'s: \( L = \{ \epsilon, ab, aabb, aaabbb, \ldots \} \).
Now, we know that context free grammars can construct all parenthesis-balanced expressions. The equivalence between grammars and automata will be exploited for the construction of such a system.
Context Free Languages?

- Regular grammars are also context-free grammars.
- It is not sufficient, to look context-free.

Example 2.6. The following grammar generates strings containing any number of a’s and b’s within a pair of quotes.

\[
\begin{align*}
S & \rightarrow \text{“}A\text{”} \\
A & \rightarrow aA \mid B \\
B & \rightarrow Bb \mid \epsilon
\end{align*}
\]

Looks context-free

But can be replaced by a regular grammar.

\[
\begin{align*}
S & \rightarrow \text{“}A\text{”} \\
A & \rightarrow aA \mid B \\
B & \rightarrow bB \mid \text{”}
\end{align*}
\]
Describing Context Free Languages
Backus-Naur Form

- BNF is metalanguage for the description of context-free languages (almost all programming languages are context free, possible exception: HASKELL)
- The Extended Backus-Naur Form was invented by Niklaus Wirth. EBNF is more handy than BNF, but otherwise of equivalent power of expression.
BNF and EBNF

- An BNF description is an unordered list of BNF rules.
- An BNF-rule consists of a LHS, a “=“ and a RHS. A rule ends with an end-of-line character.
- The “::=“ reads as “is defined as”.
- Non-terminals are enclosed in brackets <>.
- A vertical bar indicates a exclusive choice.
- Sequences of characters between ““ have to be taken literally.
  - A single literal double quote is expressed as ““
  - A single literal quote is expressed as “”.


Postal Address in BNF

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;address&gt;</td>
<td>::= &lt;name&gt; &lt;street&gt; &lt;city&gt;</td>
</tr>
<tr>
<td>&lt;name&gt;</td>
<td>::= &lt;personal-part&gt; &lt;surname&gt; &lt;EOL&gt;</td>
</tr>
<tr>
<td>&lt;personal-part&gt;</td>
<td>::= &lt;first-name&gt;</td>
</tr>
<tr>
<td>&lt;street&gt;</td>
<td>::= &lt;street-name&gt; &lt;number&gt; &lt;EOL&gt;</td>
</tr>
<tr>
<td>&lt;city&gt;</td>
<td>::= “CH-”&lt;zip-code&gt; &lt;city-name&gt; &lt;EOL&gt;</td>
</tr>
</tbody>
</table>

What is lacking?
1. Some non-terminals are not defined (e.g. first-name)
2. <EOL>
3. Treatment of white spaces
4. Titles
BNF in BNF

<syntax> ::= <rule> | <rule> <syntax>
<rule> ::= <opt-whitespace> "<" <rule-name> "">
<opt-whitespace> ::= " " <opt-whitespace> | "" "!-- "" is empty string, i.e. no whitespace -->
<expression> ::= <list> | <list> "|" <expression>
<line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list> ::= <term> | <term> <opt-whitespace> <list>
<term> ::= <literal> | "<" <rule-name> ">
<literal> ::= "" <text> "" | "" <text> "" "!-- actually, the original BNF did not use quotes -->
Extended Backus-Naur Form

- Initially introduced by Wirth, based on WSN (Wirth’s syntax notation).
- Easier to handle than BNF
- No hyphens, no brackets.
- Six special symbols: =, |, [ ], { }, [ ... ] options
- { ... } repetitions
Example: Integers in EBNF

- Integers: 23, -3454, +453223

<table>
<thead>
<tr>
<th>digit</th>
<th>= 0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>=  [+</td>
<td>[-] digit {digit}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>nonzero</th>
<th>= 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit</td>
<td>= 0</td>
<td>nonzero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>= 0</td>
<td>[+</td>
<td>[-] nonzero {digit}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parsing: From ASCII - Strings to Executables
Parsing

• EBNF / BNF gives syntax for textfiles representing programs.
• Computer can’t use text files for executing functions.
• What computers can digest are function calls.

The idea is to use the syntax tree of a sequences for calling internal functions. A parser is a program that transforms a sequence in syntax tree.
Remark: Parsers as Syntax Checkers

- Sometimes, one understands by the term parser a syntax checker.
- Today, CPU-time and memory is cheap.
- Checking a program for syntactical correctness is the main task. Constructing an according calling tree is more or less for free. ➔ Parsers include calling – tree generators.
a = (2 + 3) * (4 - 2)
Programming languages are most often context-free languages.

Their syntax trees can be understood as receipts how to call nested functions.

A parser is a program that takes a string and reconstrucnts its syntax tree. The syntax tree is used for organizing the execution of the program.
Füchslin’s Simple Lisp (FSLISP)

<table>
<thead>
<tr>
<th>Non terminals</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>function</td>
</tr>
<tr>
<td>function</td>
<td>( functionname , expression , expression )</td>
</tr>
<tr>
<td>functionname</td>
<td>prod</td>
</tr>
<tr>
<td>digit</td>
<td>0</td>
</tr>
<tr>
<td>integer</td>
<td>[+-] digit {digit}</td>
</tr>
</tbody>
</table>

FSLISP represents sums and products of integers.
Examples:
• (prod, 4, 5)
• (sum, (prod, 3, 4), (prod, (sum, 2, 3), 7))
FSLISP Interpreter

Assumption: string is of the form integer or (string1, string2, string3)

String1(string) { returns string1 }
String2(string) { returns string2 }
String3(string) { returns string3 }

IsNumber(string) { if string is an integer, return 1 else return 0 }
ToNumber(string) { converts string in integer }
FSLISP ip(string) {
If IsNumber(string) == 1,
    return ToNumber(string);
string1 = String1(string);
string2 = String2(string);
string3 = String3(string);
If string1 == "prod",
    return FSLISP_ip(string2) * 
        FSLISP_ip(string3) ;
If string1 == "sum",
    return FSLISP_ip(string2) + 
        FSLISP_ip(string3);
}

Recursive calls realize parse tree!
Pitfalls in Parsing

\[ S \rightarrow S + S | S - S | S \times S | N \]

\[ N \rightarrow 2 | 3 | 4 \]

“2 + 3”

Harmless, easy to understand.
Pitfalls in Parsing: The String 2-3-4

One string can have several syntax trees. Which one shall one choose?
Ambiguities in Grammar Trees

\[ \Sigma : \{0,1,+\} \]
\[ V : \{S,N\} \]
Start symbol: \( S \)

\[ S \rightarrow N | N + S \]
\[ N \rightarrow 0|1|NN \]
Definition A: A grammar is said to be ambiguous if the language it generates contains some sequence that has more than one possible syntax tree. **Focus is on syntax trees.**

Definition B: A grammar is said to be ambiguous if the language it generates contains some sequence that can be derived by more than one leftmost derivations. **Focus on derivations.**
Focusing on derivations in definition of ambiguity:

• **Disadvantage:** It is a bit more strict than absolutely necessary.

Note: In principle, it wouldn’t be a problem if multiple derivations lead to the same sequence, as long as their according syntax trees are functionally equivalent. But it is hard to say whether two syntax trees are functionally equivalent.
Both derivations lead to the same function!
Definition: **Left-most derivation**: Always apply replacement rules to the left-most non-terminal symbol. Right-most derivation is defined accordingly.
There Are Still Problems

\[ S \rightarrow S - S \rightarrow (S - S) - S \rightarrow (2 - S) - S \rightarrow (2 - 3) - S \rightarrow (2 - 3) - 4 \]

\[ S \rightarrow S - S \rightarrow 2 - S \rightarrow 2 - (S - S) \rightarrow 2 - (3 - S) \rightarrow 2 - (3 - 4) \]

"2–3–4" has two possible parse trees

Both derivation are „left-most“, but nevertheless different.
Removing Ambiguities

• Bad news first: Whether a context free grammar is unambiguous or not is in general an unsolvable problem.
• Amazing fact: There are context-free grammars which have only ambiguous grammars.
• There is a bunch of techniques working in specific cases.
Techniques for Removing Ambiguities

- In usual arithmetic, ambiguities are removed by requiring
  - Left associativity
  - Precedence *, / over +,-
- Solution: Introduce additional symbols that reflect the binding strength of different elements of the string.

\[
\begin{align*}
S & \rightarrow S + S \mid S - S \mid S \times S \mid N \\
N & \rightarrow 2 \mid 3 \mid 4
\end{align*}
\]
Techniques for Removing Ambiguities

- Each string in the language is an expression E.
- Each expression consists either of a term T or the sum or the difference of an expression and a term T.
- Each term consists of one or more factors.

\[
S \rightarrow E \\
E \rightarrow T \mid E + T \mid E - T \\
T \rightarrow F \mid T \times F \\
F \rightarrow N \\
N \rightarrow 2 \mid 3 \mid 4
\]
Context-Sensitive Languages

An in-between of context-free and unrestricted grammars are *context–sensitive grammars*. Definition: A context-sensitive grammar has rules of the form:

\[ \alpha \rightarrow \beta \]
\[ \alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^* \]
\[ \beta \in (\Sigma \cup V)^* \]
\[ |\alpha| \leq |\beta| \]

\( S \rightarrow \varepsilon \) is allowed, if \( S \) doesn't appear on any RHS.

Rarely used, \( |a| \leq |b| \) for proofs.
Example 2.9. A typical example of context-sensitive language that is not context-free is the language

\[ L = \{a^n b^n c^n \mid n > 0\} \]

which can be generated by the following context-sensitive grammar:

\[
\begin{align*}
S & \to aAbc \mid abc \\
A & \to aAbC' \mid abC' \\
C'b & \to bC' \\
Cc & \to cc
\end{align*}
\]

aaaabCbCbCbc \Rightarrow aaaabbbCCbCbc \Rightarrow .... \Rightarrow aaaabbbbCCCcCc \Rightarrow .... \Rightarrow aaaabbbbccccc
Unrestricted Grammars

Definition: In an unrestricted grammar, the replacement rules have the form

\[ \alpha \rightarrow \beta \]
\[ \alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^* \]
\[ \beta \in (\Sigma \cup V)^* \]
\[ \alpha \neq \varepsilon \]
# Chomsky Classification

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rules</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Type 0 Unrestricted | $\alpha \rightarrow \beta$  
$\alpha \in (\Sigma \cup V)^*V(\Sigma \cup V)^*$  
$\beta \in (\Sigma \cup V)^*$ |          |
| Type 1 Context-sensitive | $\alpha \rightarrow \beta$  
$\alpha \in (\Sigma \cup V)^*V(\Sigma \cup V)^*$  
$\beta \in (\Sigma \cup V)^*$  
$|\alpha| \leq |\beta|$  
$S \rightarrow \epsilon$ if $S$ doesn't appear on any RHS | $\{a^nb^nc^n\}$ |
| Type 2 Context-free | $A \rightarrow \beta$  
$A \in V, \beta \in (\Sigma \cup V)^*$ | $\{a^n b^n\}$ |
| Type 3 Regular     | $A \rightarrow \alpha B$  
$A, B \in V$  
$\alpha \in \Sigma^*$ | $\{a^m b^n\}$ |
$L_{\text{regular}} \subseteq L_{\text{contextfree}} \subseteq L_{\text{contextsensitive}} \subseteq L_{\text{unrestricted}}$
Why Working With Simple Grammars?

Depending on the grammar type, important questions can be answered:

- **Recognition problem**: Given a string $w$ and a grammar $G$. Is $w$ in $L(G)$?

- **Emptiness problem**: Given $G$, is $L(G) = \emptyset$? That is not trivial, because we have to determine, whether a given set of rules terminates.

- **Equivalence problem**: Given $G_1$ and $G_2$. Is $L(G_1) = L(G_2)$?

- **Ambiguity problem**: Is $G$ ambiguous?
### Languages and Problems

<table>
<thead>
<tr>
<th>Type</th>
<th>Recognition</th>
<th>Emptiness</th>
<th>Equivalence</th>
<th>Ambiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>Yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

- Though not being solvable in general, a problem may well find a solution in specific cases.
- We see, why we don’t use unrestricted grammars: We could not even check whether a sequence is syntactically correct (recognize as element of the language).
- Again: One has to find the equilibrium between expressiveness and ability to answer important question.
A language proven to be context-sensitive is Swiss German.

Example for a context – sensitive construction: "Mer bliibe deheime, wel mer d'Chind em Hans sis Huus lënd hälfe aaschtriiche."