Wstęp do informatyki

Elementy teorii jezyków formalnych

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

January 20, 2016

Table of contents

Formal language

Basic concepts

In mathematics, computer science, and linguistics,

Definition

a **formal language** is a set of **strings** of symbols that may be constrained by **rules** that are specific to it.

In short

- The **alphabet** of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed.
- The strings formed from this alphabet are called **words**. The words that belong to a particular formal language are sometimes called well-formed words or well-formed formulas.
- A formal language is often defined by means of a **formal grammar** such as a regular grammar or context-free grammar, also called its formation rule.

In mathematics, computer science, and linguistics,

Definition

a **formal language** is a set of **strings** of symbols that may be constrained by **rules** that are specific to it.

In short

- The **alphabet** of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed.
- The strings formed from this alphabet are called **words**. The words that belong to a particular formal language are sometimes called well-formed words or well-formed formulas.
- A formal language is often defined by means of a **formal grammar** such as a regular grammar or context-free grammar, also called its formation rule.

Formal language

Basic concepts

In mathematics, computer science, and linguistics,

Definition

a **formal language** is a set of **strings** of symbols that may be constrained by **rules** that are specific to it.

In short

- The **alphabet** of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed.
- The strings formed from this alphabet are called **words**. The words that belong to a particular formal language are sometimes called well-formed words or well-formed formulas.
- A formal language is often defined by means of a **formal grammar** such as a regular grammar or context-free grammar, also called its formation rule.

Formal language

Basic concepts

In mathematics, computer science, and linguistics,

Definition

a **formal language** is a set of **strings** of symbols that may be constrained by **rules** that are specific to it.

In short

- The **alphabet** of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed.
- The strings formed from this alphabet are called **words**. The words that belong to a particular formal language are sometimes called well-formed words or well-formed formulas.
- A formal language is often defined by means of a **formal grammar** such as a regular grammar or context-free grammar, also called its formation rule.

Formal language

Why we use them?

The field of formal language theory studies primarily the purely syntactical aspects of such languages—that is, their internal structural patterns. Formal language theory sprang out of linguistics, as a way of understanding the syntactic regularities of natural languages. In computer science, formal languages are used among others as the basis for defining the grammar of programming languages and formalized versions of subsets of natural languages in which the words of the language represent concepts that are associated with particular meanings or semantics. In computational complexity theory, decision problems are typically defined as formal languages, and complexity classes are defined as the sets of the formal languages that can be parsed by machines with limited computational power. In logic and the foundations of mathematics, formal languages are used to represent the syntax of axiomatic systems, and mathematical formalism is the philosophy that all of mathematics can be reduced to the syntactic manipulation of formal languages in this way.

Formal language

Why we use them?

An alphabet, in the context of formal languages, can be **any set**, although it often makes sense to use an alphabet in the usual sense of the word, or more generally a character set such as ASCII or Unicode. The elements of an alphabet are called its **letters**.

Formal language

Why we use them?

A **word** over an alphabet can be any (finite) sequence, or string, of characters or letters, which sometimes may include spaces, and are separated by specified word separation characters. The set of all words over an alphabet Σ is usually denoted by Σ^* (using the Kleene star). The length of a word is the number of characters or letters it is composed of. For any alphabet there is only one word of length 0, the empty word, which is often denoted by e, ε or λ. By concatenation one can combine two words to form a new word, whose length is the sum of the lengths of the original words. The result of concatenating a word with the empty word is the original word.

Formal language

More details about alphabet, words and language

A formal language L over an alphabet Σ is a subset of Σ^*, that is, a set of words over that alphabet. Sometimes the sets of words are grouped into expressions, whereas **rules and constraints** may be formulated for the creation of ’well-formed expressions’.
Formal language theory rarely concerns itself with particular languages, but is mainly concerned with the study of various types of formalisms to describe languages. For instance, a language can be given as

- those strings generated by some formal grammar;
- those strings described or matched by a particular regular expression;
- those strings accepted by some automaton, such as a Turing machine or finite state automaton.

Formal language

Example 1: alphabet and the set of all words over it

Jeśli $\Sigma = \{a, b\}$, to $\Sigma^* = \{e, a, b, aa, ab, bb, aaa, aab, \ldots\}$.

Formal language

Example 2: simple language with unformal rules

The following rules describe a formal language L over the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =\}$

- Every nonempty string that does not contain + or = and does not start with 0 is in L.
- The string 0 is in L.
- A string containing = is in L if and only if there is exactly one =, and it separates two valid strings of L.
- A string containing + but not = is in L if and only if every + in the string separates two valid strings of L.
- No string is in L other than those implied by the previous rules.

Formal language

Example 2: simple language with unformal rules

Under these rules, the string "23+4=555" is in L, but the string "=234=+" is not.

Formal language

Example 2: simple language with unformal rules

This formal language expresses how some strings look like (their syntax), not what they mean (semantics).

For instance, nowhere in these rules is there any indication that "0" means the number zero, or that "+" means addition.

Formal grammar

Basic concepts

Definition

In formal language theory, a **grammar** (formal grammar) **is a set of production rules** for strings in a formal language.

The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them. A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer" – a function in computing that determines whether a given string belongs to the language or is grammatically incorrect.

Formal grammar

More details

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s, a grammar G formally defined as the tuple

$$(N, \Sigma, P, S)$$

where

- N is a finite set of nonterminal symbols, that is disjoint with the strings formed from G.
- Σ is a finite set of terminal symbols that is disjoint from N.
- P is a finite set of production rules, each rule of the form

  $$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*.$$

  That is, each production rule maps from one string of symbols to another, where the first string (the "head") contains an arbitrary number of symbols provided at least one of them is a nonterminal.
- $S \in N$ is a distinguished symbol that is the start symbol.

Formal grammar

Example 1

A language L in which well-formed expressions are of the form $\{1, 11, 111, \ldots\}$ could be defined as

- $N = S$
- $\Sigma = \{1\}$
- P consists of the following production rules:

  ```
  rule 1: S --> 1
  rule 2: S --> S1
  ```
- $S = S$

Now it can be prooved thet 1 and 111 belongs to the language L

```
Proof 1: S --(rule 1)--> 1
Proof 2: S --(rule 2)--> S1 --(rule 2)--> S11 --(rule 1)--> 111
```

Formal grammar

Example 2

Consider the grammar G where

- $N = \{S, B\}$
- $\Sigma = \{a, b, c\}$
- P consists of the following production rules:

```
rule 1: S --> aBSc
rule 2: S --> abc
rule 3: Ba --> aB
rule 4: Bb --> bb
```

- $S = S$

This grammar defines the language $L(G) = \{a^n b^n c^n | n \geq 1\}$ where a^n denotes a string of n consecutive a's. Thus, the language is the set of strings that consist of 1 or more a's, followed by the same number of b's, followed by the same number of c's. Some examples of the derivation of strings in $L(G)$ are:

```
S --(2)--> abc
S --(1)--> aBSc --(2)--> aBabcc --(3)--> aaBbcc --(4)--> aabbcc
```

Backus–Naur Form Notation

Basic concepts

In computer science, BNF (Backus Normal Form or Backus–Naur Form) is a notation techniques for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols.
A BNF specification is a set of derivation rules, written as

```
<symbol> ::= __expression__
```

where

- `<symbol>` is a nonterminal symbol, and the expression `__expression__` consists of one or more sequences of symbols;
- symbols that never appear on a left side are terminals;
- symbols that appear on a left side are non-terminals and are always enclosed between the pair <>;
- symboll `::=` means that the symbol on the left must be replaced with the expression on the right.

Backus–Naur Form Notation

Example 1

Consider BNF rules for some context-free grammar

```
<binary number> ::=<binary number><digit>
<binary number> ::=<digit>
<digit> ::= 0
<digit> ::= 1
```

This is equivalent to the grammar G where

- $N = \{\texttt{<binary number>}, \texttt{<digit>}\}$
- $\Sigma = \{0, 1\}$
- P consists of the following production rules:

```
rule 1: <binary number> --> <binary number><digit>
rule 2: <binary number> --> <digit>
rule 3: <digit> --> 0
rule 4: <digit> --> 1
```

- $S = \texttt{<binary number>}$

Backus–Naur Form Notation

Example 2

Consider BNF rules for some context-free grammar (natural numbers grammar)

```
<zero> ::= 0
<nonzero digit> ::= 1|2|3|4|5|6|7|8|9
<digit> ::= <zero> | <nonzero digit>
<sequence of digits> ::= <digit> | <digit><sequence of digits>
<natural number> ::= <digit> | <nonzero digit><sequence of digits>
```

Backus–Naur Form Notation

Variants and extensions of BNF

Many BNF specifications found online today **are intended to be human readable and are non-formal**. These often include many of the following syntax rules and extensions:

- Optional items enclosed in square brackets: `[<item>]`.
- Items repeating 0 or more times are enclosed in curly brackets or suffixed with an asterisk ('*'), such as

  ```
  <word> ::= <letter> {<letter>}
  <word> ::= <letter> <letter>*
  ```
- Items repeating 1 or more times are suffixed with an addition (plus) symbol (+).
- Terminals may appear in bold rather than italics, and nonterminals in plain text rather than angle brackets.
- Alternative choices in a production are separated by the vertical bar, |, indicating a choice.
- Where items are grouped, they are enclosed in simple parentheses.

Backus–Naur Form Notation

Why extend it?

The main problem with BNF (beeing far from human readable) is that repetitions and optional parts can not be expressed directly. Instead, we have indirect rule and recursive way of defining repetitions and options.

- Repetition

```
Extended BNF <number> ::= { digit }+
BNF          <number> ::= <digit>
             <number> ::= <number> <digit>
```

- Option

```
EBNF <signed number> ::= [ <sign> ] <number>
BNF  <signed number> ::= <sign> <number>
     <signed number> ::= <number>
```

Backus–Naur Form Notation

Which Extended BNF is the right one?

The earliest EBNF was originally developed by Niklaus Wirth incorporating some of the concepts (with a different syntax and notation) from Wirth syntax notation. However, as we have seen, many variants of EBNF are in use. The International Organization for Standardization has adopted an EBNF standard (ISO/IEC 14977). Other EBNF variants use somewhat different syntactic conventions. The most important thing about (E)BNF is not to follow precise and strictly formal rules about it but to keep in mind why we use it and how to make it human readable and easy to use.

Regular expression

Basic concepts

A **regular expression**, often called a **pattern**, is an expression used to specify a set of strings required for a particular purpose. A simple way to specify a set of strings is simply to list its elements or members. However, there are often more concise ways to specify the desired set of strings. For example, the set containing the three strings `Handel`, `Händel`, and `Haendel` can be specified by the pattern `H(ä|ae?)ndel`; we say that this pattern **matches** each of the three strings.

Regular expression

Basic concepts

Most formalisms provide the following operations to construct regular expressions.

- Boolean "or". A vertical bar separates alternatives. For example, `gray|grey` can match `gray` or `grey`.
- Grouping. Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of `gray` or `grey`.

- Quantification. A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` (derived from the Kleene star), and the plus sign + (Kleene cross).
  - The question mark indicates there is zero or one of the preceding element. For example, `colou?r` matches both `color` and `colour`.
  - The asterisk indicates there is zero or more of the preceding element. For example, `ab*c` matches `ac`, `abc`, `abbc`, `abbbc`, and so on.
  - The plus sign indicates there is one or more of the preceding element. For example, `ab+c` matches `abc`, `abbc`, `abbbc`, and so on, but not `ac`.

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations $+$, $-$, etc. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`. Note!!! The precise syntax for regular expressions varies among tools and with context.

Regular expression

Examples

```
Visit http://regexpal.com/ to verify

.at     matches any three-character string ending with "at",
          including "hat", "cat", and "bat".
[hc]at  matches "hat" and "cat".
[^b]at  matches all strings matched by .at except "bat".
[^hc]at matches all strings matched by .at other than "hat" and "cat".
^[hc]at matches "hat" and "cat", but only at the beginning
          of the string or line.
[hc]at$ matches "hat" and "cat", but only at the end
           of the string or line.
\[.\]   matches any single character surrounded by "[" and "]" since
          the brackets are escaped, for example: "[a]" and "[b]".
```