

# Languages and Grammars

Jan van Eijck

CWI, Amsterdam and Uil-OTS, Utrecht

[jve@cwi.nl](mailto:jve@cwi.nl)

May 21, 2008

## **Abstract**

We give formal definitions of languages and grammars, and look at examples.

## Module Declaration

```
module LAG  
  
where  
import List  
import Char
```

## Alphabets

An alphabet  $\Sigma$  is a finite set of symbols.

Examples:

- $\Sigma_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ . The ten-element set of the decimal symbols.
- $\Sigma_2 = \{a, b, c, \dots, x, y, z\}$ . The 26-element set of all lowercase letters of English (or Dutch).

A non-example:

- $\mathbb{N} = \{0, 1, 2, \dots\}$ . The set of all natural numbers is not an alphabet, for this set is infinite.

## All strings over an alphabet

If  $\Sigma$  is an alphabet, we use  $\Sigma^*$  of the set of all finite strings over  $\Sigma$ .

Let  $\Sigma^n$  be the set of all  $n$ -tuples of elements of  $\Sigma$ . Then  $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$ .

```
listsOfLength :: Int -> [Char] -> [String]
listsOfLength 0 alphabet = [[]]
listsOfLength n alphabet =
  [ x:xs | x <- alphabet,
          xs <- listsOfLength (n-1) alphabet ]

star :: [Char] -> [String]
star alphabet =
  concat [ listsOfLength n alphabet | n <- [0..] ]
```

A more general version:

```
listsOfLength :: Int -> [a] -> [[a]]
listsOfLength 0 alphabet = [[]]
listsOfLength n alphabet =
  [ x:xs | x <- alphabet,
          xs <- listsOfLength (n-1) alphabet ]

star :: [a] -> [[a]]
star alphabet =
  concat [ listsOfLength n alphabet | n <- [0..] ]
```

This gives:

```
LAG> listsOfLength 3 "ab"
```

```
["aaa","aab","aba","abb","baa","bab","bba","bbb"]
```

```
LAG> take 10 (star "ab")
```

```
["","a","b","aa","ab","ba","bb","aaa","aab","aba"]
```

## Notation

If  $x$  is a symbol, then use  $x^n$  for a string of  $n$   $x$ 's. Haskell implementation:

```
copies :: a -> Int -> [a]
copies x 0 = []
copies x n = x : copies x (n-1)
```

## Strings, Empty String, String Reversal

The empty string is often denoted  $\epsilon$ . Note that  $\emptyset$  and  $\{\epsilon\}$  are different.

If  $w$  is a string,  $w^R$  is the reversal of string  $w$ .

Recursive definition of this operation:

- $\epsilon^R = \epsilon$
- $(xw)^R = w^R x$  (where  $x$  is a single element, and  $w$  a string).

Haskell:

```
reversal :: [a] -> [a]
reversal [] = []
reversal (x:xs) = reversal xs ++ [x]
```

## Languages over an Alphabet

A **language** over an alphabet  $\Sigma$  is a subset of  $\Sigma^*$ .

Examples:

- The set  $\{\text{arno}, \text{jan}\}$ . This language has only two elements.
- The set  $\{a, \dots, z\}^*$  of all strings over the lower-case alphabet  $\{a, \dots, z\}$ .
- The set consisting of the union of  $\{0\}$  and the set of all non-empty strings over  $\{0, \dots, 9\}$  that do not start with 0.
- If  $\Sigma = \{0, 1\}$ , then  $L = \{0^m 1^n \mid m, n \in \mathbb{N}\}$  is a language over  $\Sigma$ .  $L$  is the set of all strings consisting of a number of 0's followed by a (possibly different) number of 1's.

A non-example:

- The singleton set containing the sequence  $0.14285714285714\dots$  (the decimal expansion of  $\frac{1}{7}$ ). This sequence is infinite.

## Important facts

- There are uncountably many languages over  $\Sigma$ , even if  $\Sigma$  is finite.
- $\emptyset$  is a language over  $\Sigma$ .
- $\{\epsilon\} = \Sigma^0$  is a language over  $\Sigma$ .
- $\Sigma = \Sigma^1$  is a language over  $\Sigma$ .
- For every  $n$ ,  $\Sigma^n$  is a language over  $\Sigma$ .
- $\Sigma^*$  is a language over  $\Sigma$ .

## More Example Languages

If  $\Sigma = \{0, 1\}$ , then  $L = \{0^n 1^n \mid n \in \mathcal{N}\}$  is a language over  $\Sigma$ .  $L$  is the set of all strings consisting of a number of 0's followed by an equal number of 1's.

If  $\Sigma = \{0, 1\}$ , then  $L = \{ww^R \mid w \in \Sigma^*\}$  is a language over  $\Sigma$ .  $L$  is the set of all even palindromes over  $\Sigma$ .

If  $\Sigma = \{0, 1\}$ , then  $L = \{wxw^R \mid x \in \Sigma, w \in \Sigma^*\}$  is a language over  $\Sigma$ .  $L$  is the set of all odd palindromes over  $\Sigma$ .

## Operations on Languages

Because languages are sets, the usual set operations are defined on them: we can take unions, intersections and differences.

If  $L_1$  is the set of even palindromes over  $\{0, 1\}$  and  $L_2$  is the set of odd palindromes over  $\{0, '\}$ , then  $L_1 \cup L_2$  is the language of all palindromes over  $\{0, '\}$ .

We use  $\bar{L}$  for the complement of  $L$  with respect to  $\Sigma^*$ , that is to say  $\bar{L} = \Sigma^* - L$ .

The **concatenation** of the languages  $L_1$  and  $L_2$ , notation  $L_1L_2$ , is the set of strings  $\{uw \mid u \in L_1 \text{ and } w \in L_2\}$ .

Note the following:

- $\{\epsilon\}L = L\{\epsilon\} = L$ .
- $\emptyset L = L\emptyset = \emptyset$ .

## *n*-fold product of a language

The *n*-fold product of a language is defined in terms of the concatenation operation on languages.

If  $L$  is a language over  $\Sigma$ , then the ***n*-fold product** of  $L$ , notation  $L^n$ , is given by the following clauses:

- $L^0 = \{\epsilon\}$ .
- $L^n = LL^{n-1} \quad (n \geq 1)$ .

Thus,  $L^n$  consists of all strings  $w_1 \cdots w_n$  with  $w_1 \in L, \dots, w_n \in L$ .

## Closure of a language

The **closure** or **Kleene star** of a language  $L$ , notation  $L^*$ , is the following set:

$$\bigcup_{n \geq 0} L^n.$$

Note the following:

- $\emptyset^* = \{\epsilon\}$
- If  $L = \Sigma$ , then  $L^* = \Sigma^*$ .

## Examples

The language  $\{1\}\{0,1\}^*$  is the set of all strings over  $\{0,1\}$  that start with 1. The language  $\{0\} \cup \{1\}\{0,1\}^*$  can be taken as a representation of the natural numbers in binary notation: the representation of every number except 0 starts with the symbol 1.

The language  $\{1\}\{00\}^*$  is the set of all strings over  $\{0,1\}$  that consist of 1 followed by an even number of 0's.

## Positive Closure

The **positive closure** of a language  $L$ , notation  $L^+$ , is the following set:

$$\bigcup_{n \geq 1} L^n.$$

Note the following:

- $L^+ = LL^* = L^*L$ .
- $L^* = L^+ \cup \{\epsilon\}$ .
- If  $\epsilon \in L$ , then  $L^+ = L^*$ .

## Reversal

The **reversal** of a language  $L$ , notation  $L^R$ , is the set

$$\{w^R \mid w \in L\}.$$

Note the following:

- $\Sigma^R = \Sigma$ , for any alphabet  $\Sigma$ .
- $(L^R)^R = L$ .
- $\Sigma^{*R} = \Sigma^{R*} = \Sigma^*$ .

## Generators and Recognizers for Languages

The notion of a language over an alphabet is very general and coarse-grained. It does make sense to look at ways of **finitely representing** members of the set  $\mathcal{P}(\Sigma^*)$ . There are basically two main ways of doing this:

1. Specify a **generator** for  $L$ , that is to say an effective procedure for enumerating the members of  $L$  (in some arbitrary order).
2. Specify a **recognizer** for  $L$ , that is to say a device that takes strings over  $\Sigma$  as its input and that accepts every member of  $L$  and rejects every member of  $\bar{L}$ .

## 'Most' languages cannot be recognized or generated

If  $\Sigma$  is finite,  $\Sigma^*$  is denumerable. We have already seen a recipe, in Haskell: `star sigma`.

By Cantor's theorem,  $\mathcal{P}(\Sigma^*)$ , the set of all languages over  $\Sigma$ , is **not** denumerable. There are uncountably many languages over  $\Sigma$ .

Generators and recognizers are finite objects, which means that they themselves can be described by means of finite strings of symbols over some alphabet  $\Delta$ . It follows that the number of generators and recognizers is countable. Because the set  $\mathcal{P}(\Sigma^*)$  is uncountable, there are languages over  $\Sigma$  for which there is neither a generator nor a recognizer.

We will concentrate on languages that can be finitely represented.

## Grammars

A **grammar**  $G$  is a quadruple  $(V, \Sigma, R, S)$ , where  $V$  is a finite set of symbols,  $\Sigma \subseteq V$ ,  $R \subseteq V^*(V - \Sigma)V^* \times V^*$ ,  $R$  finite and  $S \in V - \Sigma$ .

Note that  $V^*(V - \Sigma)V^*$  is the set of strings over  $V$  that contain at least one member of  $V - \Sigma$ .

If  $G = (V, \Sigma, R, S)$  is a grammar, then the members of  $\Sigma$  are called the **terminal symbols** of  $G$ , the members of  $V - \Sigma$  are the **nonterminal symbols** of  $G$ , the members of  $R$  are the **rules** or **productions** of  $G$  and the symbol  $S$  is the **start symbol** of  $G$ .

It is usual to write  $x \longrightarrow y$  or  $x ::= y$  for  $(x, y) \in R$ . If  $x \longrightarrow y$  is a rule of a grammar, then  $x$  is called the **lefthand side** of the rule and  $y$  the **righthand side**. A convention is that (strings of) capital letters are used for nonterminal symbols and (strings of) lower-case letters for terminal symbols. A grammar can be presented as a finite set of rules.

## Example Grammar

The following set of grammar rules has  $\{S, N, P, D, C, V\}$  as its set of nonterminal symbols and

$\{\text{john, mary, talked, walked, every,}$   
 $\text{the, man, woman, admired, despised}\}$

as its set of terminal symbols.

1.  $S \longrightarrow N P$
2.  $N \longrightarrow D C$
3.  $P \longrightarrow V N$
4.  $N \longrightarrow \mathbf{john}$
5.  $N \longrightarrow \mathbf{mary}$
6.  $P \longrightarrow \mathbf{talked}$
7.  $P \longrightarrow \mathbf{walked}$
8.  $D \longrightarrow \mathbf{every}$
9.  $D \longrightarrow \mathbf{the}$
10.  $C \longrightarrow \mathbf{man}$
11.  $C \longrightarrow \mathbf{woman}$
12.  $V \longrightarrow \mathbf{admired}$
13.  $V \longrightarrow \mathbf{despised}$

Another convention: use | for choice between righthand sides:

$S \longrightarrow N P$

$N \longrightarrow D C \mid \mathbf{john} \mid \mathbf{mary}$

$P \longrightarrow V N \mid \mathbf{talked} \mid \mathbf{walked}$

$D \longrightarrow \mathbf{every} \mid \mathbf{the}$

$C \longrightarrow \mathbf{man} \mid \mathbf{woman}$

$V \longrightarrow \mathbf{admired} \mid \mathbf{despised}$

## Immediate yield, immediate derivation

If  $G = (V, \Sigma, R, S)$  is a grammar, then  $x \in V^*$  **immediately yields**  $y \in V^*$  **in**  $G$ , notation

$$x \Rightarrow_G y,$$

if there are  $w, z \in V^*$  such that

$$(u, v) \in R, x = wuz \text{ and } y = wvz.$$

If  $x \Rightarrow_G y$ , then  $y$  is called an **immediate derivation from**  $x$ .

For example:

- $S \Rightarrow_G N P$
- $N P \Rightarrow_G N$  **talked**
- $D \text{ man} \Rightarrow_G$  **the man**

We can string immediate derivations together, as follows:

$$\begin{aligned} S &\Rightarrow_G N P \\ &\Rightarrow_G \mathbf{john} P \\ &\Rightarrow_G \mathbf{john} V N \\ &\Rightarrow_G \mathbf{john} V D C \\ &\Rightarrow_G \mathbf{john} V \mathbf{every} C \\ &\Rightarrow_G \mathbf{john} V \mathbf{every} \mathbf{woman} \\ &\Rightarrow_G \mathbf{john} \mathbf{admired} \mathbf{every} \mathbf{woman} \end{aligned}$$

## Yield, derivation

If  $G = (V, \Sigma, R, S)$  is a grammar, then

$x$  **yields**  $y$  **in**  $G$

if

$$x \Rightarrow_G^* y.$$

The string  $y$  is **derived in**  $G$  **from**  $x$ .

In other words: the derivation relation is the reflexive transitive closure of the relation of immediate derivation. The names of the relations  $\Rightarrow$  and  $\Rightarrow^*$  emphasize the close link between the theory of grammars and the theory of deductive systems.

## Examples

- $S \Rightarrow_G^* S$
- $S \Rightarrow_G^* \text{john admired every woman}$
- $N \Rightarrow_G^* \text{every } C$
- $N P \Rightarrow_G^* \text{every woman } P$

## Language generated by a grammar

If  $G = (V, \Sigma, R, S)$  is a grammar, then **the language generated by  $G$** , notation  $L(G)$ , is the set

$$\{x \mid x \in \Sigma^* \text{ and } S \Rightarrow_G^* x\}$$

In other words: the language generated by  $G$  is the set of all terminal strings that the start symbol yields in  $G$ .

Note that there may be several different ways of generating a given string:

- $S \Rightarrow N P \Rightarrow \text{john } P \Rightarrow \text{john walked.}$
- $S \Rightarrow N P \Rightarrow N \text{ walked} \Rightarrow \text{john walked.}$

## Recursion in grammars

Grammar  $G = (V, \Sigma, R, S)$  is **recursive** if  $G$  has a rule with lefthand side  $y$  and there are  $x, z \in \Sigma^*$  such that

$$y \Rightarrow_G \cdots \Rightarrow_G xyz.$$

Only recursive grammars can generate infinite languages.

More precisely: grammar  $G$  generates an infinite language iff (if and only if) the following hold:

- $y \Rightarrow_G \cdots \Rightarrow_G xyz$ , with  $x, z \in \Sigma^*$  and  $x \neq \epsilon$  or  $z \neq \epsilon$ ;
- $S \Rightarrow_G^* y$ ;
- $y$  yields a terminal string.

## Examples

The recursive grammar given by the following rules generates the (infinite) language of binary representations of the whole numbers.

1.  $S \longrightarrow 0$
2.  $S \longrightarrow 1A$
3.  $S \longrightarrow -1A$
4.  $A \longrightarrow 0A$
5.  $A \longrightarrow 1A$
6.  $A \longrightarrow \epsilon$

The infinite language of propositional logic:

$$\begin{aligned} S &\longrightarrow P \mid \neg S \mid (S \wedge S) \mid (S \vee S) \mid (S \rightarrow S) \mid (S \leftrightarrow S) \\ P &\longrightarrow p \mid q \mid r \mid P' \end{aligned}$$

## Weak Equivalence

Grammars  $G_1$  and  $G_2$  are **weakly equivalent** if  $L(G_1) = L(G_2)$ .

The following grammars are weakly equivalent:

$$\begin{aligned} S &\longrightarrow 0 \mid 1A \mid -1A \\ A &\longrightarrow 0A \mid 1A \mid \epsilon \end{aligned}$$

$$\begin{aligned} S &\longrightarrow 0 \mid 1 \mid -1 \mid 1A \mid -1A \\ A &\longrightarrow 0 \mid 1 \mid 0A \mid 1A \end{aligned}$$

## Right-Linear Grammars

Grammar  $G = (V, \Sigma, R, S)$  is a **right-linear grammar** if every member of  $\text{dom}(R)$  is an element of  $V - \Sigma$  and every member of  $\text{rng}(R)$  is either an element of  $\Sigma^*$  or an element of  $\Sigma^*(V - \Sigma)$ .

Example:

$$S \longrightarrow 0S \mid 1S \mid \epsilon$$

Problem: give a right-linear grammar for the set of strings over  $\{0, 1\}$  that have equal numbers of zeros and ones.

## Recognizers for Right-linear Grammars

Haskell example:

```
binaryWhole :: [Char] -> Bool
binaryWhole ['0'] = True
binaryWhole ('1':xs) = binaryWholeA xs
binaryWhole ('-': '1':xs) = binaryWholeA xs
binaryWhole _ = False

binaryWholeA :: [Char] -> Bool
binaryWholeA [] = True
binaryWholeA ('0':xs) = binaryWholeA xs
binaryWholeA ('1':xs) = binaryWholeA xs
binaryWholeA _ = False
```

## Generating Right Linear Languages

Use a recognizer as a filter:

```
genWholes :: [String]
genWholes = filter binaryWhole (star "-01")
```

This gives:

```
LAG> take 10 genWholes
```

```
["0", "1", "-1", "10", "11", "-10", "-11", "100", "101", "110"]
```

## Context Free Grammars

Grammar  $G = (V, \Sigma, R, S)$  is a **context-free grammar** or **CF grammar** if every member of  $\text{dom}(R)$  is an element of  $V - \Sigma$ .

A language that can be generated by a context-free grammar is called a context-free language.

## Recognizing with Context Free grammars

Many techniques, but here is a simple approach in Haskell:

```
split2 :: [a] -> [[a],[a]]
split2 [] = ([],[a])
split2 (x:xs) = ([],[a]):
    (map ( \ (ys,zs) -> ((x:ys),zs)) (split2 xs))
```

```
split3 :: [a] -> [[a],[a],[a]]
split3 xs = [(ys,zs,us) | (ys,ws) <- split2 xs,
                        (zs,us) <- split2 ws ]
```

```
split4 :: [a] -> [[a],[a],[a],[a]]
split4 xs =
    [(ys,zs,us,vs) | (ys,ws) <- split2 xs,
                    (zs,us,vs) <- split3 ws ]
```

Etc, depending on the maximum length of rhs in the rules.



```
genA :: [String]
genA = filter recognizeA (star "ab")
```

This gives:

```
LAG> take 6 genA
["", "ab", "aabb", "aaabbb", "aaaabbbb", "aaaaabbbbb"]
```

## Another example

$$S \longrightarrow AC$$

$$A \longrightarrow aAb \mid \epsilon$$

$$C \longrightarrow cC \mid \epsilon.$$

It is convenient to define:

```
recliteral :: Eq a => a -> [a] -> Bool
recliteral x xs = xs == [x]
```

```
recS :: [Char] -> Bool
recS xs = or [ recA ys && recC zs |
               (ys,zs) <- split2 xs ]

recA [] = True
recA xs = or [ recliteral 'a' ys
               && recA zs
               && recliteral 'b' ws |
               (ys,zs,ws) <- split3 xs ]

recC [] = True
recC xs = or [ recliteral 'c' ys
               && recC zs | (ys,zs) <- split2 xs ]
```

```
genS :: [String]
genS = filter recS (star "abc")
```

This gives:

```
LAG> take 10 genS
```

```
["", "c", "ab", "cc", "abc", "ccc", "aabb", "abcc", "cccc", "aabbc"]
```

## $\epsilon$ free-ness

A grammar  $G$  is  $\epsilon$ -free if either

1.  $G$  does not have  $\epsilon$ -productions, or
2. the only  $\epsilon$ -production of  $G$  is  $S \longrightarrow \epsilon$ , and no production of  $G$  has  $S$  in its righthand side.

Proposition: For every CF grammar  $G$  there is a weakly equivalent  $\epsilon$ -free CF grammar  $G'$ .

## Context-sensitive grammars

Grammar  $G = (V, \Sigma, R, S)$  is a **context-sensitive grammar** or **CS grammar** if the following holds:

- $G$  is  $\epsilon$ -free;
- every member of  $R$  that is not equal to  $S \longrightarrow \epsilon$  has the form

$$xAz \longrightarrow xyz,$$

with  $A \in V - \Sigma$ ,  $x \in V^*$ ,  $z \in V^*$  and  $y \in V^+$ .

Note that all productions of a context-sensitive grammar except the production  $S \longrightarrow \epsilon$  (if this production is present) have their righthand side length  $\geq$  their lefthand side length. It is clear that not every CF grammar is a CS grammar, because CF grammars need not be  $\epsilon$ -free. On the other hand, every  $\epsilon$ -free CF grammar is a CS grammar.

## Example

The following CS grammar is not context-free.

1.  $S \longrightarrow 0SAB$
2.  $S \longrightarrow 01B$
3.  $BA \longrightarrow AB$
4.  $1A \longrightarrow 11$
5.  $1B \longrightarrow 12$
6.  $2B \longrightarrow 22$

This grammar generates the following language:

$$\{0^n 1^n 2^n \mid n \geq 1\}.$$

Can you see why?

## The Chomsky Hierarchy

The following definitions give the so-called **Chomsky-hierarchy** of languages:

- Language  $L$  is **unrestricted** or **type-0** if  $L = L(G)$  for some grammar  $G$ , but  $L$  is not generated by any context-sensitive grammar.
- Language  $L$  is **context-sensitive** or **type-1** if  $L = L(G)$  for some context-sensitive grammar  $G$ , but  $L$  is not generated by any context-free grammar.
- Language  $L$  is **context-free** or **type-2** if  $L = L(G)$  for some context-free grammar  $G$ , but  $L$  is not generated by any right-linear grammar.
- Language  $L$  is **right-linear** or **regular** or **type-3** if  $L = L(G)$  for some right-linear grammar  $G$ .

Chomsky suggested that natural languages are context-sensitive.

## Relations between grammar types, and between languages

The following relations hold between grammar-types:

- every right-linear grammar is a CF grammar;
- every CF grammar has a weakly equivalent  $\epsilon$ -free CF grammar;
- every  $\epsilon$ -free CF grammar is a CS grammar;
- every CS grammar is a grammar.

These facts allow us to make the following statement about the Chomsky-hierarchy (we use  $\mathcal{L}_i$  for the class of all type- $i$  languages):

$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0.$$

## Strictness of the Chomsky hierarchy

In order to show that the  $\subseteq$  relations can be replaced by  $\subsetneq$  relations, we need to establish negative results:

“ $L \notin \mathcal{L}_i$  because no type- $i$  grammar generates  $L$ ”.

This is possible, and it yields:

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0.$$