8 Automata and formal languages

This exposition was developed by Clemens Gröpl and Knut Reinert. It is based on the following references, all of which are recommended reading:

1. Uwe Schöning: Theoretische Informatik - kurz gefasst. 3. Auflage. Spektrum Akademischer Verlag, Heidelberg, 1999. ISBN 3-8274-0250-6
2. http://www.expasy.org/prosite/prosuser.html — PROSITE user manual
3. Sigrist C.J., Cerutti L., Hulo N., Gattiker A., Falquet L., Pagni M., Bairoch A., Bucher P.. PROSITE: a documented database using patterns and profiles as motif descriptors. Brief Bioinform. 3:265-274(2002).

We will present basic facts about:

- formal languages,
- regular and context-free grammars,
- deterministic finite automata,
- nondeterministic finite automata,
- pushdown automata.

8.1 Formal languages

An *alphabet* Σ is a nonempty set of *symbols* (also called *letters*). In the following, Σ will always denote a finite alphabet.

A *word* over an alphabet Σ is a sequence of elements of Σ. This includes the *empty word*, which contains no letters and is denoted by ε.

For any alphabet Σ, the set Σ^* is defined to be the set of all words over the alphabet Σ. The set $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ contains all nonempty words Σ.

E. g., if $\Sigma = \{a, b\}$, then

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$$

and

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, aab, \ldots\}.$$

The *length* of a word x is denoted by $|x|$.

For *words* $x, y \in \Sigma^*$ we denote their *concatenation* by xy: If $x = x_1, \ldots x_m$ and $y = y_1, \ldots, y_n$ (where $x_i, y_j \in \Sigma$) then $xy = x_1, \ldots, x_m, y_1, \ldots, y_n$.

For $x \in \Sigma^*$ let $x^n := \underbrace{xx \ldots x}_{n}$. (That is, n concatenated copies of x).

Thus $|x^n| = n|x|$, $\left|xy\right| = |x| + \left|y\right|$, and $|\varepsilon| = 0$.

A (formal) *language* A over an alphabet Σ is simply a set of words over Σ, i. e., a subset of Σ^*. The *empty language* $\emptyset := \{\}$ contains no words.

(Note: The empty language $\emptyset$ must not to be confused with the language $\{\varepsilon\}$, which contains only the empty word.)

The *complement* of a language A (over an alphabet Σ) is the language $\bar{A} := \Sigma^* \setminus A$.

For languages A, B we define their *product* as

$$AB := \{xy \mid x \in A, y \in B\},$$

using the concatenation operation defined above.

E. g., if $A = \{a, ha\}$, $B = \{\varepsilon, t, ttu\}$ then

$$AB = \{a, ha, at, hat, attu, hattu\}.$$

The *powers* of a language L are defined by

$$\begin{aligned} L^0 &:= \{\varepsilon\} \\ L^n &:= L^{n-1}L, \quad \text{for } n \geq 1. \end{aligned}$$

The "*Kleene star*" (or Kleene hull) of a language L is

$$L^* := \bigcup_{i=0}^{\infty} L^n.$$

Funnily, $\emptyset^n = \emptyset$ for $n \geq 1$, but $\emptyset^* = \emptyset^0 = \{\varepsilon\}$, according to these definitions. But (defined this way) *language exponentiation* works as expected: For *every* language A and $m, n \geq 0$, we have $A^m A^n = A^{m+n}$.

Most formal languages are infinite objects. In order to deal with them algorithmically, we need finite descriptions for them. There are two approaches to this:

- *Grammars* describe rules how to *produce* words from a given language. We can classify languages according to the kinds of rules which are allowed.
- *Automata* describe how to *test* whether a word belongs to a given language. We can classify languages according to the computational power of the automata which are allowed.

Fortunately, the two approaches can be shown to be equivalent in many cases.

8.2 Grammars

Definition.
A *grammar* is a 4-tuple $G = (V, \Sigma, P, S)$ satisfying the following conditions.

- V is a finite set of *variable symbols*. For brevity, the variable symbols are often simply called *variables*, or *nonterminals*.

- Σ is a finite set of *terminal symbols*, also called the *terminal alphabet*. This is the alphabet of the language we want to describe. We require that variable symbols and terminal symbols can be distinguished, i. e., $V \cap \Sigma = \emptyset$.
- P is a finite set of *rules* or *productions*. A rule has the form

  $$\text{(left hand side)} \rightarrow \text{(right hand side)},$$

  where lhs $\in (V \cup \Sigma)^+$ and rhs $\in (V \cup \Sigma)^*$.
- $S \in V$ is the *start variable*.

We can think of the productions of a grammar as ways to "*transform*" words over the alphabet $V \cup \Sigma$ into other words over $V \cup \Sigma$.

Definition.
We can *derive* v from u in G *in one step* if there is

- a production $y \rightarrow y'$ in P and
- $x, z \in (V \cup \Sigma)^*$ such that
- $u = xyz$ and $v = xy'z$.

This is denoted by $u \Rightarrow_G v$. If the grammar is clear from the context, we write just $u \Rightarrow v$.

Definition.
The *reflexive and transitive closure* $\Rightarrow_G^*$ of $\Rightarrow_G$ is defined as follows. We have $u \Rightarrow_G^* v$ if and only if $u = v$ or v can be derived from u in a series $u \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \Rightarrow v$ of steps using the grammar G. Then

$$L(G) := \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$$

is the *language generated by* G.

Note: The crucial point is that we wrote $w \in \Sigma^*$, not $w \in (V \cup \Sigma)^*$. Variables are not allowed in generated words which are "output".

(*Note 2:* This raises another question: Productions can lengthen and shorten the words. How can we tell how long it will take until we have removed all variable symbols? Well, that's another story.)

Example 1.
The following grammar generates all words over $\Sigma = \{a, b\}$ with equally many a's and b's.

$G = (\{S\}, \Sigma, P, S)$, where

$$\begin{aligned} P := \{ & \\ & S \rightarrow \varepsilon, \\ & S \rightarrow Sab \\ & ab \rightarrow ba, \\ & ba \rightarrow ab \\ \} & \end{aligned}$$

(Can you prove this?)

Example 2.
The following grammar generates well-formed arithmetic expressions over the alphabet $\Sigma = \{(,), a, b, c, +, *\}$.

$G = (\{A, M, K\}, \Sigma, P, A)$, where

$$\begin{aligned} P := \{ & \\ & A \to M, \\ & A \to A + M \\ & M \to K, \\ & M \to M * K, \\ & K \to a, \\ & K \to b, \\ & K \to c, \\ & K \to (A), \\ \} & \end{aligned}$$

Chomsky described four sorts of restrictions on a grammar's rewriting rules. The resulting four classes of grammars form a hierarchy known as the *Chomsky* hierachy. In what follows we use capital letters $A, B, W, S, \ldots$ to denote nonterminal symbols, small letters $a, b, c, \ldots$ to denote terminal symbols and greek letters $\alpha, \beta, \gamma, \ldots$ to represent a string of terminal and non-terminal letters.

1. **Regular grammars**. Only production rules of the form $W \to aW$ or $W \to a$ are allowed.
2. **Context-free grammars**. Any production of the form $W \to \alpha$ is allowed.
3. **Context-sensitive grammars**. Productions of the form $\alpha_1 W \alpha_2 \to \alpha_1 \beta \alpha_2$ are allowed.
4. **Unrestricted grammars**. Any production rule of the form $\alpha_1 W \alpha_2 \to \gamma$ is allowed.

8.3 Regular grammars

For Bioinformatics we will be interested in the *regular* and *context-free* grammars. Hence a more detailed definition:

Definition.
A *grammar* is called *regular* if all productions have the form $\ell \to r$, where $\ell \in V$ and $r \in \Sigma \cup \Sigma V$.

That is, we can only replace a variable with:

- a terminal ($r \in \Sigma$), or
- a terminal followed by a variable ($r \in \Sigma V$).

Example.
The following regular grammar generates valid identifier names in many programming languages.

$G = (\{\texttt{[alpha]}, \texttt{[alnum]}\}, \{A, \ldots Z, a, \ldots, z, \_, 0, 1, \ldots, 9\}, P, \texttt{[alpha]}\}$, where

$$\begin{aligned} P := \{ & \\ & \texttt{[alpha]} \to A, \ldots, \_, \\ & \texttt{[alpha]} \to A\texttt{[alnum]}, \ldots, \_\texttt{[alnum]}, \\ & \texttt{[alnum]} \to A, \ldots, \_, 0, \ldots, 9, \\ & \texttt{[alnum]} \to A\texttt{[alnum]}, \ldots, \_\texttt{[alnum]}, 0\texttt{[alnum]}, \ldots, 9\texttt{[alnum]} \\ \} & \end{aligned}$$

Here we used commas to write several productions sharing the same left hand side in one line.

8.4 Context-free grammars

Here the definition of a *context-free grammar*:

Definition 1. A *context free grammar* G is a 4-tuple $G = (V, \Sigma, P, S)$ with V and Σ being alphabets with $V \cap \Sigma = \emptyset$.

- V is the nonterminal alphabet.
- Σ is the terminal alphabet.
- $S \in N$ is the start symbol.
- $P \subseteq V \times (V \cup \Sigma)^*$ is the finite set of all productions.

Consider the context-free grammar

$$G = \Big(\{S\},\ \{a, b\},\ \{S \to aSa \mid bSb \mid aa \mid bb\},\ S \Big).$$

This CFG produces the language of all *palindromes* of the form $\alpha\alpha^R$.

For example the string *aabaabaa* can be generated using the following derivation:

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabaabaa.$$

The "palindrome grammar" can be readily extended to handle RNA hairpin loops. For example, we could model hairpin loops with three base pairs and a *gcaa* or *gaaa* loop using the following productions.

$$\begin{aligned} S &\to aW_1u \mid cW_1g \mid gW_1c \mid uW_1a, \\ W_1 &\to aW_2u \mid cW_2g \mid gW_2c \mid uW_2a, \\ W_2 &\to aW_3u \mid cW_3g \mid gW_3c \mid uW_3a, \\ W_3 &\to gaaa \mid gcaa. \end{aligned}$$

(We don't mention the alphabets V and Σ explicitly if they are clear from the context.)

Grammars *generate* languages. They are a means to quickly specify all (possibly an infinite number) words in a language.

Now we will turn the attention to the automata that can decide whether a word is in the language or not. If the word is in the language the automaton *accepts* the word. We start with finite automata and proove that they are able to accept exactly the words generated by a regular grammar.

8.5 Deterministic finite automata

Definition.
A *deterministic finite automaton (DFA)* is a 5-tuple $M = (Z, \Sigma, \delta, z_0, E)$ satisfying the following conditions.

- Z is a finite set of *states* the automaton can be in.
- Σ is the alphabet. The automaton "moves" along the input from left to right. In each step, it reads a single character from the input.
- $\delta : Z \times \Sigma \to Z$ is the *transition function*. When the character has been read, M changes its state depending on the character and its current state. Then it proceeds to the next input position.

- z_0 is the *initial state* of M before the first character is read.
- E is the set of *end states* or *accepting states*. If M is in a state contained in E after the last letter has been read, the input is accepted.

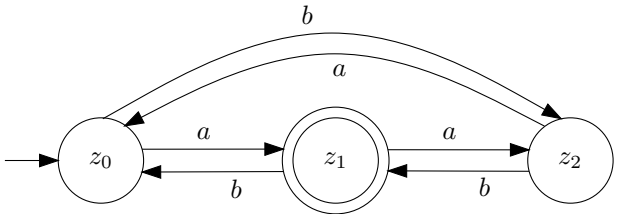
DFAs can be drawn as *digraphs* very intuitively.

- *States* correspond to *vertices*. They are drawn as single circles; accepting states are indicated by double circles.
- *Edges* correspond to *transitions* and are labeled with letters from Σ. There is an arc from u to v labeled a if and only if there is a transition $\delta(u, a) = v$. The initial state is marked by an ingoing arrow.

Example.
The following automaton accepts the language

$$L = \{x \in \{a, b\}^* \mid (\#_a(x) - \#_b(x)) \,\%\, 3 = 1\}\,.$$

(where % denotes modulus, i. e. remainder of division)



Using the definition of DFA, we have $M = (\{z_0, z_1, z_2\}, \{a, b\}, \delta, z_0, \{z_1\})$, where

$$\begin{array}{lll} \delta(z_0, a) = z_1 & \delta(z_1, a) = z_2 & \delta(z_2, a) = z_0 \\ \delta(z_0, b) = z_2 & \delta(z_1, b) = z_0 & \delta(z_2, b) = z_1 \end{array}$$

Definition.
A *language* L is called *regular* if there is a regular grammar that produces $L \setminus \{\varepsilon\}$.

Lengthy remark: The issue with ε is really just a technical complication. We can always modify a grammar G that generates a language L into a grammar G' that generates the language $L \cup \{\varepsilon\}$ by the following trick: Let S be the start variable of G. Let S' be a new variable symbol not used by G. Then G' is obtained by replacing the start variable by S' and adding the following productions: $S' \to S \mid \varepsilon$.

Whether the resulting grammar G' is also called regular (if G was regular) depends on the literature. Schöning uses the following "ε-Sonderregelung": If $\varepsilon \in L(G)$ is desired, then the production $S \to \varepsilon$ is admitted, where S is the start variable. However, in this case S must not appear on the right hand side of a production.

8.6 From DFAs to regular grammars

Again, let $M = (Z, \Sigma, \delta, z_0, E)$ be a deterministic finite automaton.

It is useful to extend the transition function $\delta : Z \times \Sigma \to Z$ to a mapping $\delta^* : Z \times \Sigma^* \to Z$, called the *extended transition function*. We define $\delta^*(z, \varepsilon) := z$ for every state $z \in Z$ and inductively,

$$\delta^*(z, xa) := \delta(\delta^*(z, x), a) \qquad \text{for } x \in \Sigma^*, a \in \Sigma\,.$$

Observe that if $x = x[1 .. n]$ is an input string, then $\delta^*(z_0, x[1.. 0])$, $\delta^*(z_0, x[1 .. 1]), \ldots, \delta^*(z_0, x[1 .. n])$ is the path of states followed by the DFA. Hence, the *language accepted by M* is

$$L(M) := \{x \in \Sigma* \mid \delta^*(z_0, x) \in E\}.$$

We are now ready to prove:

Theorem 2. *Every language which is accepted by a deterministic finite automaton is regular.*

Proof: Let $M = (Z, \Sigma, \delta, z_0, E)$ be a DFA and $A := L(M)$. We will construct a regular grammar $G = (V, \Sigma, P, S)$ that generates A.

We let $V := Z$ and $S := z_0$.

Every arc $\delta(u, a) = v$ becomes a production $u \to av \in P$, and if $v \in E$ we also include a production $u \to a$. That is,

$$P := \{u \to av \mid \delta(u, a) = v\} \cup \{u \to a \mid \delta(u, a) = v \in E\}.$$

Now we have

$x[1 .. n] \in L(M)$

$\Leftrightarrow$ there are *states* $z_1, \ldots, z_n \in Z$ such that
$\delta(z_{i-1}, x[i]) = z_i$ for $i = 1, \ldots, n$,
where z_0 is the start state and $z_n \in E$ is an accepting state

$\Leftrightarrow$ there are *variables* $z_1, \ldots, z_n \in V$ such that
$z_{i-1} \to x[i]z_i$ is a production in P,
where z_0 is the start variable,
and $z_{n-1} \to x[n]$ is also a production in P

$\Leftrightarrow$ we can derive
$S = z_0 \Rightarrow x[1]z_1 \Rightarrow x[1]x[2]z_2 \Rightarrow \ldots \Rightarrow x[1 .. n{-}1]z_{n-1} \Rightarrow x[1 .. n]$
in G, i. e., $S \Rightarrow_G^* x$

$\Leftrightarrow$ $x[1 .. n] \in L(G)$.

If $\varepsilon \in A$, i. e., $z_0 \in E$, then we need to apply the "ε-Sonderregelung" and modify G accordingly. ∎

8.7 Nondeterministic finite automata

In DFAs, the path followed upon a given input was completely determined. Next we will introduce *nondeterministic finite automata (NFAs)*. These are defined similar to DFAs, but each state can have more than one successor state for any given letter, or none at all.

Definition.
A *nondeterministic finite automaton (NFA)* is a 5-tuple $M = (Z, \Sigma, \delta, U_0, E)$ satisfying the following conditions.

- Z is a finite set of *states*.
- Σ is the alphabet.
- $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ is the *transition function*. Here $\mathcal{P}(Z)$ is the *power set* of Z, i. e. the set of all subsets of Z.

- U_0 is the *set of initial states*.
- E is the *set of accepting states*.

When the automaton reads $a \in \Sigma$ and is in state z, it is free to *choose* one of several sucessor states in $\delta(z, a)$, or its *gets stuck* if $\delta(z, a) = \emptyset$.

Definition.
A nondeterministic finite automaton *accepts* and input if there is *at least one* accepting path.

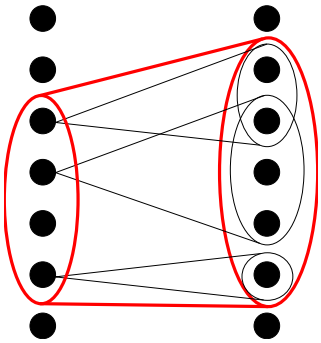
Again, we can define an *extended transition function* $\delta^* : \mathcal{P}(Z) \times \Sigma^* \to \mathcal{P}(Z)$. We let $\delta^*(U, \varepsilon) := U$ for all subsets of states $U \subseteq Z$ and inductively,

$$\delta^*(U, xa) := \bigcup_{v \in \delta^*(U,x)} \delta(v, a) \qquad \text{for } x \in \Sigma^*, a \in \Sigma \ .$$

Then the *language accepted by* M is

$$L(M) := \{x \in \Sigma^* \mid \delta^*(U_0, x) \cap E \neq \emptyset\} .$$

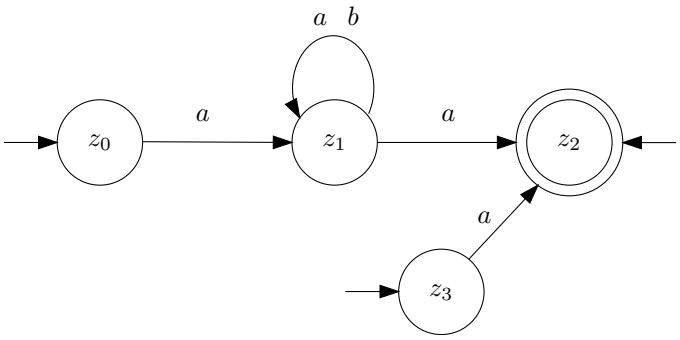
The following illustrates the definition of δ^*. The large bubble on the left side is $\delta^*(U, x)$, the large bubble on the right side is $\delta^*(U, xa)$, where a is some letter. The state space (fat dots) is shown twice for clarity. Time goes from left to right. The smaller cones indicate $\delta(v, a)$ for each $v \in \delta^*(U, x)$.



NFAs can be *drawn* as *digraphs*, similar to DFAs. The resulting digraphs are more general:

- We can have several arrows pointing to start states.
- The number of arcs with a given label leaving a vertex is no longer required to be exactly 1, it can be any number (including 0).

Example.
The following NFA accepts all words over the alphabet $\Sigma = \{a, b\}$ which do not start or end with the letter b.



8.8 DFAs and NFAs are equivalent

Although NFAs are a generalization of DFAs, they accept the same languages:

Theorem 3 (Rabin, Scott)**.** *For every nondeterministic finite automaton M there is a deterministic finite automaton M' such that $L(M) = L(M')$.*

Proof. Let $M = (Z, \Sigma, \delta, U_0, E)$ be an NFA. The basic idea of the proof is *to view the subsets of states of Z as single states of an DFA M' whose state space is $\mathcal{P}(Z)$.* Then the rest of the definition of M' is straightforward.

The "*power set automaton*" is defined as $M' := (\mathcal{P}(Z), \Sigma, \delta', U_0, E')$, where

- $\mathcal{P}(Z)$ is the *state space*
- The *transition function* $\delta' : \mathcal{P}(Z) \times \Sigma \to \mathcal{P}(Z)$ is defined by

$$\delta'(U, a) := \bigcup_{v \in U} \delta(v, a) = \delta^*(U, a) \qquad \text{for } U \in \mathcal{P}(Z)\,.$$

- $U_0 \in \mathcal{P}(Z)$ is the new *start state* (note that in M it was the *set* of start states).
- $E' := \{U \subseteq Z \mid U \cap E \neq \emptyset\}$ is the new set of *end states*.

Using the definitions of M and M', we have:

$x[1 \,..\, n] \in L(M)$

$\Leftrightarrow$ there are accepting paths in M,

$$\delta^*(U_0, x[1 \,..\, n]) \cap E \neq \emptyset$$

$\Leftrightarrow$ there are subsets $U_1, U_2, \ldots, U_n \subseteq Z$ such that

$$\delta'(U_{i-1}, x[i]) = U_i \quad \text{and} \quad U_n \cap E \neq \emptyset$$

$\Leftrightarrow$ there is an accepting path in M',

$$\delta'^*(U_0, x[1 \,..\, n]) = U_n \in E'$$

$\Leftrightarrow$ $x[1 \,..\, n] \in L(M')$. ∎

Remarks:

1. In the power set construction, we can safely *leave out states which cannot be reached* from the start state U_0. That is, we can generate the reached state sets "on the fly".
2. The *exponential blow up* of the number of states (from $|Z|$ to $2^{|Z|}$) cannot be avoided in general. For example, the language
$$L := \{x \in \{a,b\}^* \mid |x| \geq k \text{ and the } k\text{-last letter of } x \text{ is an } a\}$$
has an NFA with $k+1$ states, but it is not hard to show that no DFA for L can have less that 2^k states.

8.9 From regular grammars to NFAs

We have just seen how to transform a nondeterministic finite automaton into a deterministic finite automaton.

We have seen before how to transform a deterministic finite automaton into a regular grammar.

Next we will see how to transform a regular grammar into a nondeterministic finite automaton.

This concludes the proof that the regular languages are precisely those which are accepted by finite automata (of both kinds).

Theorem 4. *Every regular language is accepted by a nondeterministic finite automaton.*

Proof.

- Let $G = (V, \Sigma, P, S)$ be a *regular grammar* and $A := L(G)$. We will construct an *NFA* $M = (Z, \Sigma, \delta, \{z_0\}, E)$ such that $L(M) = A$.
- Note that in every derivation in a regular grammar, the intermediate words contain *exactly one variable*, and the variable must be at the end. This variable will become a *state* of the NFA. We need one more extra state, which M enters when the variable is eliminated in the last step.
- Thus we let the state set be $Z := V \cup \{X\}$. The only possible *initial state* is $z_0 := S$, the start variable of G. The set of *end states* is $E := \{X\}$ if $\varepsilon \notin A$. If $\varepsilon \in A$, then we let $E := \{X, S\}$.
- Next we *translate productions into transitions*. We define $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ by
$$\begin{aligned} \delta(u,a) &\ni v && \text{iff } u \to av \in P \\ \delta(u,a) &\ni X && \text{iff } u \to a \in P \end{aligned}$$
That is,
$$\delta(u,a) = \{v \mid u \to av \in P\} \cup \{X \mid u \to a \in P\}.$$
Note that the end state X has no successor states. And if S is an end state, then by the "ε-Sonderregelung" there is no way to get back to S, as it does not appear on the right side of a production.
- Now we have for $n \geq 1$:

  $x[1 \,..\, n] \in L(G)$
  $\Leftrightarrow$ there are *variables* $z_1, \ldots, z_n \in V$ such that we can derive
  $z_0 = S \Rightarrow_G x[1]z_1 \Rightarrow_G x[1]x[2]z_2 \Rightarrow_G \ldots \Rightarrow_G x[1 \,..\, n-1]z_{n-1} \Rightarrow_G x[1 \,..\, n]$
  in G, that is, $z_{i-1} \to x[i]z_i$ is a production in P,
  where $z_0 = S$ is the start variable,
  and $z_{n-1} \to x[n]$ is also a production in P
  $\Leftrightarrow$ there are *states* $z_1, \ldots, z_n \in Z \cup \{X\}$ such that
  $\delta(z_{i-1}, x[i]) \ni z_i$ for $i = 1, \ldots, n$,
  where z_0 is the start state,
  and $z_n = X$ is only end state which is feasible for a word of length ≥ 1

$\Leftrightarrow\ x[1\,..\,n] \in L(M)$

Moreover, by construction we have $\varepsilon \in L(M) \Leftrightarrow \varepsilon \in A$. ■

Finally we return to context-free grammars and introduce the automaton that accepts a context-free language, the pushdown automaton (PDA).

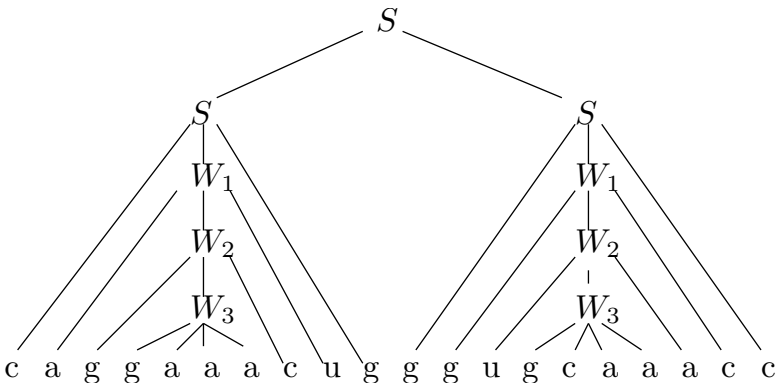
We will first define it and then show at an example how we can decide whether a string is in the language generated by a given CFG.

Recall our small hairpin generating CFG:

$$\begin{aligned} S &\to aW_1u \mid cW_1g \mid gW_1c \mid uW_1a,\\ W_1 &\to aW_2u \mid cW_2g \mid gW_2c \mid uW_2a,\\ W_2 &\to aW_3u \mid cW_3g \mid gW_3c \mid uW_3a,\\ W_3 &\to gaaa \mid gcaa. \end{aligned}$$

There is an elegant representation for derivations of a sequence in a CFG called the *parse tree*. The root of the tree is the nonterminal S. The leaves are the terminal symbols, and the inner nodes are nonterminals.

For example if we extend the above productions with $S \to SS$ we can get the following:



Using a pushdown automaton we can parse a sequence left to right according.

Definition.
A (nondeterministic) PDA is formally defined as a 6-tuple:
$M = (Z,\ \Sigma,\ \Gamma,\ \delta,\ z_0, S)$ where

- Z is a finite set of states, Σ is a finite input alphabet, Γ is a finite stack alphabet
- $\delta : Z \times \Sigma \cup \{\epsilon\} \times \Gamma \longrightarrow \mathcal{P}(Z \times \Gamma^*)$ is the
- z_0 is the start state, S is the initial stack symbol
- $S \in \Gamma$ is the lowest stack symbol.

There is of course also a deterministic version, however the nondeterministic PDA allows for a simple construction when a CFG is given.

If M is in state z and reads the input a and if A is the top stack symbol, then M *can* go to state z' and replace A by other stack symbols.

This implies, that A can be deleted, replaced, or augmented.

After reading the input, the automaton accepts the word if the stack is empty.

Given a CFG $G = (V, \Sigma, P, S)$ we define the corresponding PDA as $M = (\{z\}, \Sigma, V \cup \Sigma, \delta, z, S)$. Using the production set P we can define δ. For each rule $A \to \alpha \in P$ we define δ such that $(z, \alpha) \in \delta(z, \epsilon, A)$ and $(z, \epsilon) \in \delta(z, a, a)$.

Lets look at our example:

$$\begin{aligned} S &\to aW_1u \mid cW_1g \mid gW_1c \mid uW_1a, \\ W_1 &\to aW_2u \mid cW_2g \mid gW_2c \mid uW_2a, \\ W_2 &\to aW_3u \mid cW_3g \mid gW_3c \mid uW_3a, \\ W_3 &\to gaaa \mid gcaa. \end{aligned}$$

Hence $M = (\{z\}, \{a, c, g, u\}, \{a, c, g, u, W_1, W_2, W_3, S\}, \delta, z, S)$ with δ as explained (blackboard).

Lets see how the automaton parses a word in our hairpin language.

Given our CFG, the automaton's stack is initialized with the start symbol S. Then the following steps are iterated until no symbols remain. If the stack is empty when no input symbols remain, then the sequence has been successfully parsed.

1. Pop a symbol off the stack.
2. If the popped symbol is a non-terminal: Peek ahead in the input and choose a valid production for the symbol. (For deterministic PDAs, there is at most one choice. For non-deterministic PDAs, all possible choices need to be evaluated individually.) If there is no valid transision, terminate and reject.

   Push the right side of the production on the stack, rightmost symbols first.
3. If the popped symbol is a terminal: Compare it to the current symbol of the input. If it matches, move the automaton to the right on the input. If not, reject and terminate.

Lets try this with the string `gccgcaaggc`.

Example. (The current symbol is written using a capital letter):

```
Input string    Stack      Operation
Gccgcaaggc      S          Pop S.   Produce S->gW1c
Gccgcaaggc      gW1c       Pop g.   Accept g.  Move right on input.
gCcgcaaggc      W1c        Pop W1.  Produce W1->cW2g
gCcgcaaggc      cW2gc      Pop c.   Accept c.  Move right on input.
gcCgcaaggc      W2gc       Pop W2.  Produce W2->cW3g
gcCgcaaggc      cW3ggc     Pop c.   Accept c.  Move right on input.
gccGcaaggc      W3ggc      Pop W3.  Produce W3->gcaa
gccGcaaggc      gcaaggc    Pop g.   Accept g.  Move right on input.
...             ...        (several acceptances)
gccgcaaggC      c          Pop c.   Accept c.  Move right on input.
gccgcaaggc$     -          Stack empty, input string empty.  Accept!
```

8.10 Summary

- Formal languages are a basic means in computer science to formally describe objects that follow certain rules, that is that can be *generated* using a grammar.
- Two fundamental views on formal languages are i) to view them as generated by a grammar, or ii) to view them as accepted by an automaton.
- The Chomsky hierarchy places different restrictions on the grammars. This limits the possibilities you have, but makes the decision whether a word is in such a language easier.
- The nondeterministic automata (DFA and PDA) are as powerful as te deterministic counterparts in deciding whether a word is in a language or not. However, it is easier to define a nondeterministic automaton.
- The importance for bioinfomatics lies in the ”stochastic versions” of the grammars which are used to train ”acceptors” for biological sequence objects (i.e. Genes using Hidden Markov models, or RNA using stochastic CFGs).