St. Pölten University of Applied Sciences

/informatik & security

/fh/// st. pölten

Formal Languages and Automata 2

Theoretical Computer Science

Dipl.-Ing. Hubert Schölnast, BSc
September 11, 2021

St. Pölten University of Applied Sciences, Campus-Platz 1, 3100 St. Pölten, T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E: csc@fhstp.ac.at, I: www.fhstp.ac.at

Table of Contents

4 Type 2 grammars, languages and automata

4.1 Context-free grammars

Let us recall again the general definition of a grammar:

A grammar is a set with 4 elements:

$$G = \{V, \Sigma, P, S\}$$

- G = a grammar
- V = the alphabet of variables (auxiliary signs = non-terminal signs)
- Σ = the alphabet of final (terminal) signs
- P = the set of production rules
- S = the start sign (actually a start word that consists of exactly 1 sign)

V and Σ are disjoint: $V \cap \Sigma = \emptyset$

S is a word consisting of exactly one sign from V (in most books written as $S \in V$)

Each element of P is a rule, and each rule looks like this:

$$l \to r$$

with

$$l \in (V \cup \Sigma)^+$$

$$r \in (V \cup \Sigma)^*$$

For the production rule of a regular grammar, these restrictions applied:

- For the left hand side:
  - $l = A$ where A had to be exactly one sign from the alphabet V.
- For the right side:
  - Either $r = aB$ where a had to be a sign from the alphabet Σ and B had to be a sign from the alphabet V.
  - Or $r = \varepsilon$

For context-free grammars, only the rule for the left-hand side continues to apply; for the right-hand side, anything is allowed that is allowed in the general definition. Thus:

- $l = A$ where A must be exactly one sign from the alphabet V.
- $r \in (V \cup \Sigma)^*$

4.1.1 What does "context-free" mean?

This term results from the fact that on the left side of each rule only exactly one sign may stand, and this sign must be an auxiliary character. It must come thus from the alphabet V.

As a consequence, no single rule in a context-free grammar can be influenced by any sign from the alphabet Σ in the already existing word. These terminal signs in the neighborhood of that place where the rule acts are called the context. And the characteristic feature of context-free grammars is that they do not care about this context. Therefore, these languages are called "context-free".

Of course, all regular grammars have this property, too, because exactly the same restriction applies to the left-hand side there, too. Therefore, every regular grammar is also a context-free grammar. But there are also context-free grammars that are not regular. An example of this was the grammar at the very end of the previous part, which could generate the language $a^n b^n$.

So the regular grammars are a real subset of the context-free grammars. Therefore, the regular languages are also a subset of the context-free languages, and as has already been shown in the example of the language $a^n b^n$ just mentioned, the subset relation of the languages is also real. (That is: the two sets are not equal).

4.1.2 Normal forms

Chomsky normal form (CNF)

Noam Chomsky has shown that the following constraint for the right-hand sides of the production rules can be introduce without changing the set of languages produced:

$$A \to BC$$

$$A \to a$$

$$S \to \varepsilon$$

Here, as always, A, B, C and S are auxiliary signs from the alphabet V, of which S is the start sign. a is a terminal sign from Σ, and ε is the empty word.

Chomsky also gave an algorithm that can be used to convert any context-free grammar into this normal form. The existence of this algorithm, which can be applied to any context-free grammar, also proves that CNF grammars can produce the same languages as any context-free languages.

The presentation of this algorithm is omitted here.

Greibach Normal Form (GNF)

The American computer scientist Sheila A. Greibach has developed another normal form for context-free grammars, which admits the following rule types:

$$A \to aB_1B_2 \cdots B_k \text{ with } k \geq 0$$

$$S \to \varepsilon$$

Thus, the first rule type allows rules in the form $A \to a,\ A \to aB,\ A \to aBC,\ A \to aBCD, \cdots$. That is, any rule that does not generate the empty word starts on the right-hand side with exactly one terminal sign, which may be followed by any number of non-terminal signs.

The special thing about GNF grammars is that (if the empty word is not generated) they add exactly one terminal character to the word at each step of a rule application. Generating a word with a GNF grammar requires exactly the same number of steps as the finished word contains signs.

This property also holds for all regular grammars, and in fact all regular grammars are a subset of GNF grammars where the number of non-terminal signs in each rule is exactly 1 ($k = 1$).

Ms. Greibach has also given an algorithm that applies to any context-free grammar and makes it a GNF grammar. The existence of this algorithm proves that GNF grammars can produce all languages that can produce all context-free grammars. However, this algorithm is not shown here.

4.1.3 Examples

Example 1

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{0, 1\}$
- $V = \{\mathrm{S}\}$
- $\mathrm{P} = \{\mathrm{S} \to 0\mathrm{S}1 \mid \varepsilon\}$

$\mathcal{L}(G) = \{\varepsilon, 01, 0011, 000111, 00001111, 0000011111, 000000111111, \cdots\}$

$\mathcal{L}(G) = \{w \mid w = 0^n 1^n\}$

Example 2

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{0, 1, +, *, (,), |\}$
- $V = \{\mathrm{R}\}$
- $\mathrm{P} = \{\mathrm{R} \to \mathrm{RR} \mid \mathrm{R}+ \mid \mathrm{R}* \mid \mathrm{R}|\mathrm{R} \mid (\mathrm{R}) \mid 0 \mid 1 \mid \varepsilon\}$
- $S = \mathrm{R}$

$\mathcal{L}(G) = ?$

Example 3

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{0 \cdots 9, a \cdots z, A \cdots Z, +, -, *, /, ., ,, (,), [,], \_\}$
- V = {‹expression›, ‹term›, ‹factor›, ‹unary expr›, ‹func call›, ‹arg list›, ‹var ref›, ‹simple var›, ‹qualifier›, ‹index›, ‹func name›, ‹identifier›, ‹literal›, ‹int lit›, ‹float lit›, ‹letter›, ‹digit›, ‹add op›, ‹mult op›}
- S = {‹expression›}
- P = {

‹expression› → ‹term› | ‹term›‹add op›‹expression› ,
‹term› → ‹factor› | ‹factor›‹mult op›‹term› ,
‹factor› → (‹expression›) | ‹literal› | ‹func call› | ‹var ref› | ‹unary expr› ,
‹unary expr› → ‹add op›‹factor› ,
‹func call› → ‹func name›() | ‹func name›(‹arg list›) ,
‹arg list› → ‹expression› | ‹arg list›, ‹expression› ,
‹var ref› → ‹simple var› | ‹simple var›‹qualifier› ,
‹simple var› → ‹identifier› ,
‹qualifier› → ‹index› | .‹simple var› | .‹func call› ,
‹index› → [‹expression›] ,
‹func name› → ‹identifier› ,
‹identifier› → ‹letter› | _ | ‹identifier›‹letter› | ‹identifier›_ | ‹identifier›‹digit› ,
‹literal› → ‹int lit› | ‹float lit› ,
‹int lit› → ‹digit› | ‹int lit›‹digit› ,
‹float lit› → ‹int lit›.‹int lit› | ‹int lit›E‹int lit› | ‹int lit›E‹add op›‹int lit› ,
‹letter› → a | b | ⋯ | z | A | B | ⋯ | Z ,
‹digit› → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ,
‹add op › → + | − ,
‹mult op› → ∗ | /

}

Step by step insertion to create a word of this language:

‹expression›
‹term›‹add op›‹expression›
‹term›‹add op›‹term›
‹term› + ‹term›
‹factor› + ‹factor›‹mult op›‹term›
‹factor› + ‹factor› ∗ ‹term›
‹factor› + ‹factor› ∗ ‹factor›
(‹expression›) + ‹factor› ∗ ‹factor›
(‹expression›) + ‹literal› ∗ ‹factor›
(‹expression›) + ‹literal› ∗ ‹literal›
(‹term›‹add op›‹expression›) + ‹literal› ∗ ‹literal›
(‹term› − ‹expression›) + ‹literal› ∗ ‹literal›
(‹term› − ‹term›) + ‹literal› ∗ ‹literal›
(‹factor› − ‹term›) + ‹literal› ∗ ‹literal›
(‹factor› − ‹factor›) + ‹literal› ∗ ‹literal›
(‹literal› − ‹factor›) + ‹literal› ∗ ‹literal›

(‹literal› − ‹literal›) + ‹literal› ∗ ‹literal›
(‹int lit› − ‹literal›) + ‹literal› ∗ ‹literal›
(‹int lit› − ‹float lit›) + ‹literal› ∗ ‹literal›
(‹int lit› − ‹float lit›) + ‹float lit› ∗ ‹literal›
(‹int lit› − ‹float lit›) + ‹float lit› ∗ ‹int lit›
(‹int lit›‹digit› − ‹float lit›) + ‹float lit› ∗ ‹int lit›
(‹int lit›‹digit›‹digit› − ‹float lit›) + ‹float lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹float lit›) + ‹float lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹int lit›. ‹int lit›) + ‹float lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹int lit›. ‹int lit›) + ‹int lit›E‹add op›‹int lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹digit›. ‹int lit›) + ‹int lit›E‹add op›‹int lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹digit›. ‹digit›) + ‹int lit›E‹add op›‹int lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹digit›. ‹digit›) + ‹digit›E‹add op›‹int lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹digit›. ‹digit›) + ‹digit›E − ‹int lit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹digit›. ‹digit›) + ‹digit›E − ‹digit› ∗ ‹int lit›
(‹digit›‹digit›‹digit› − ‹digit›. ‹digit›) + ‹digit›E − ‹digit› ∗ ‹digit›
(4‹digit›‹digit› − ‹digit›. ‹digit›) + ‹digit›E − ‹digit› ∗ ‹digit›
(40‹digit› − ‹digit›. ‹digit›) + ‹digit›E − ‹digit› ∗ ‹digit›
(408 − ‹digit›. ‹digit›) + ‹digit›E − ‹digit› ∗ ‹digit›
(408 − 2. ‹digit›) + ‹digit›E − ‹digit› ∗ ‹digit›
(408 − 2.3) + ‹digit›E − ‹digit› ∗ ‹digit›
(408 − 2.3) + 3E − ‹digit› ∗ ‹digit›
(408 − 2.3) + 3E − 3 ∗ ‹digit›
(408 − 2.3) + 3E − 3 ∗ 6

4.2 Pushdown Automata (PDA)

Pushdown automata are an extension of finite state machines, and the element that is added is the stack. The stack is a special kind of memory. In many restaurants there are devices in which plates are stacked. A spring at the bottom ensures that the top plate of the stack is always at the same height, no matter how many plates are already in the stack. When a new plate is placed on the stack, the entire stack is pushed down. If you remove a plate, the whole stack pops back up. And this behavior of the stack when you add something to the stack is where the name "pushdown automaton" comes from. This name just should remind you that there is a stack in each pushdown automaton.

Besides the state machine and the stack, however, pushdown automatons also have two other elements that already existed in finite state automatons, but were not explicitly mentioned there: The tape and the reading head. The individual components are now described separately.

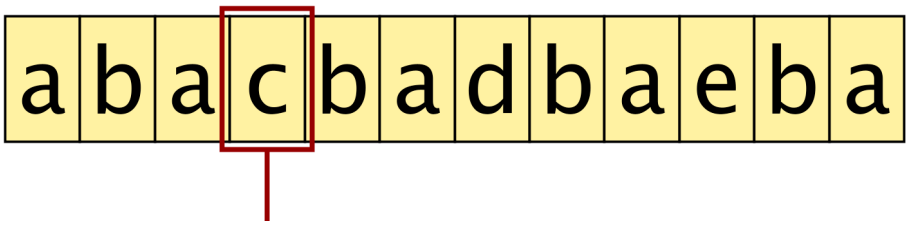
4.2.1 The tape

The tape already existed with finite state machines, but is never explicitly mentioned in the literature for this type of automaton.

In finite state machines, the word to be read in is always displayed somehow detached from the state diagram of the automaton. However, one can think of the word as a text written on a strip of paper. This strip is divided into small boxes so that there is exactly one sign in each box. This strip is called a "tape" and the model for this was magnetic tapes on which the signs were stored as patterns of alternating magnetic polarization, in such a way that each sign took up the same amount of space on the tape.

But the actual technical realization is irrelevant for the definition of the automatons. The tape exists in all automaton types, from finite automata to Turing machines. It is a linear memory, which is divided into memory cells. In each cell there is exactly one sign from the tape alphabet. This tape alphabet is usually called Σ. The characters are adjacent on the tape, each character except the first has exactly one left neighbor, and each character except the last has exactly one right neighbor. The right neighbor of a sign is called "next sign" in finite state machines and in pushdown automatons, and the left neighbor is called "previous sign".

In these two types of automata, the tape is a read-only memory. The automaton reads signs from the tape, but cannot write anything on it.



A tape with a reading head positioned above it. The read head is positioned on the fourth sign of the input word

4.2.2 The reading head

This name also comes from the analogy with magnetic tapes. However, it is actually nothing more than a pointer that marks a certain place on the tape. The phrase "the reading head is on a certain sign" means that this sign is the currently active sign that the automaton is currently taking care of.

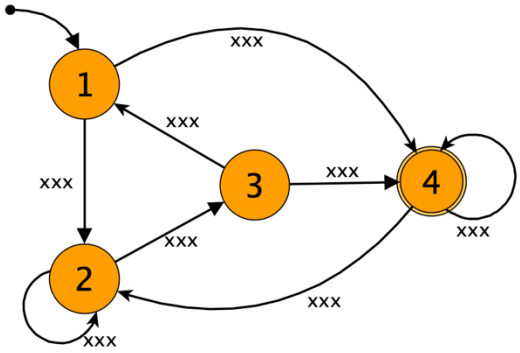
In finite state machines and pushdown automata, reading starts at the first sign, which is the leftmost sign, and then another sign is read in each processing step. The reading head always moves forward by exactly 1 sign, and only to the right.

The read head moves from the previous character to the current character, then the automaton "reads" the character with the help of the head, and then the automaton somehow reacts to that character. While the automaton is reacting, the head stays on the current sign. As soon as the automaton finishes its reaction, the head moves exactly one position further to the right, to the next sign.

4.2.3 The state machine

The state machine was the central component of finite state machines, and this is also the heart of all other automaton types, including pushdown automata, and also the principle operation is always the same:

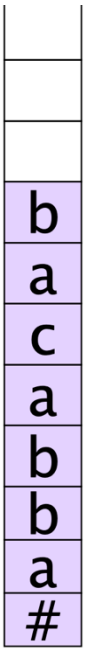
The automaton is in a certain state before reading a sign from the tape. Before reading the very first sign (the leftmost sign of the word) this is the initial state, which is part of the definition of the automaton. For all other signs it is the state the automaton has reached after reacting to the previous sign.



The combination of this input state and the sign read in then determines what new state the automaton should change to.

This is essentially already the description of the state machine in a finite state machine. The state machines of other automaton types can do even more, this will be explained in more detail in the respective section. For pushdown automata this explanation follows already in a few pages. For this purpose, one must know something about the stack:

4.2.4 The Stack



The pushdown automaton can place individual signs in the stack one after the other. Each new sign added to the stack is placed on top of all the existing signs. So, the newest signs is always on top. All other signs are below it. This way the signs form a stack that grows from the bottom to the top.

The signs that the automaton places in the stack are signs from the stack alphabet. The stack alphabet may contain some or all of the characters from the tape alphabet, and it may also contain any number of other characters that do not appear in the tape alphabet.

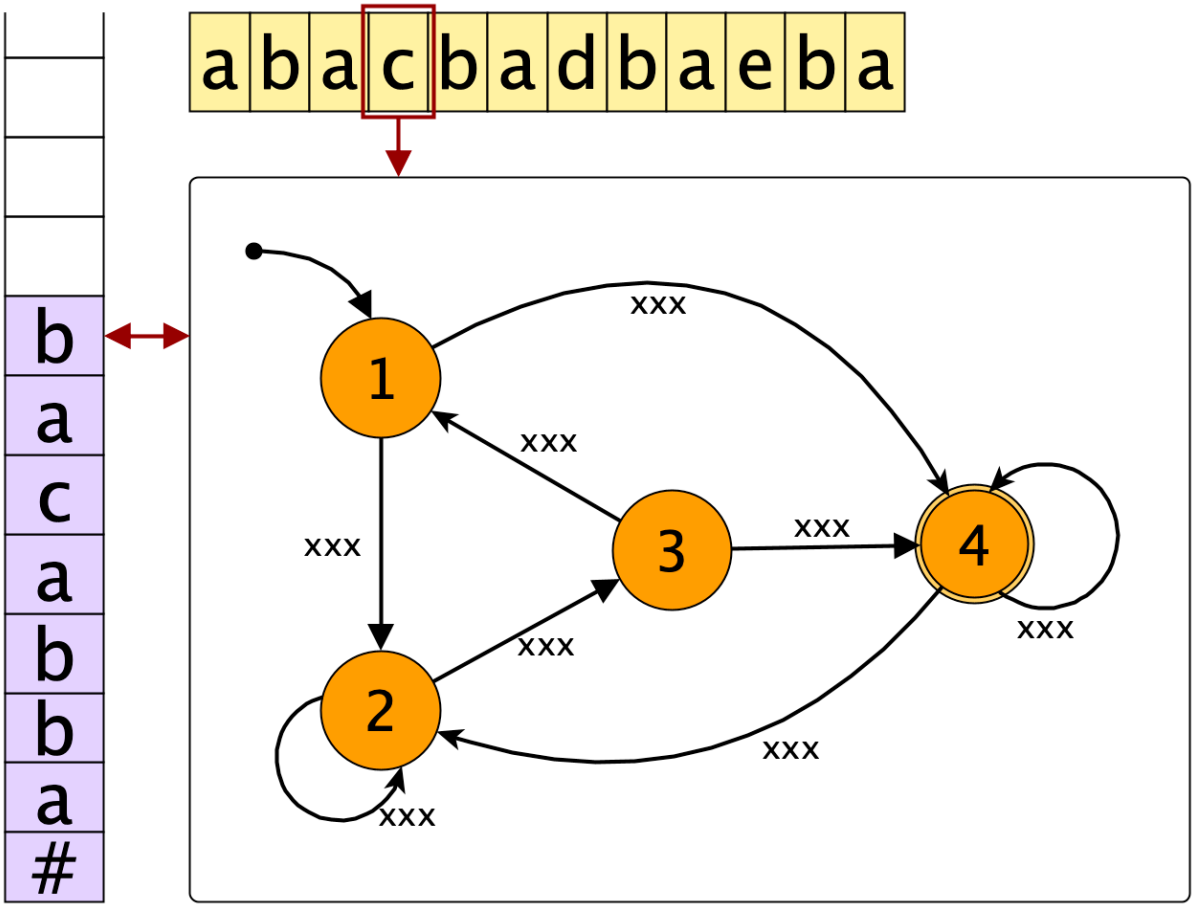
One sign in the stack alphabet is a special sign. It is the stack bottom sign. In the initial state of the stack, that is, before the very first sign is read, the entire content of the stack consist only of the stack bottom sign. So only the lowest position of the stack is occupied by a sign, all other places are free.

The stack bottom sign may occur only once in the stack, therefore the stack bottom sign is also the only sign that the automaton cannot push into the stack. To be more precise: Every time the automaton reads a signs from the stack, this sign will be removed from the stack. This is true for the stack bottom sign too. Then, at the end of the automatons "reaction" the automaton can push zero, one or more signs onto the stack. But if it did read the stack bottom sign before, it must push it back into stack as the first sign. And

after that the automat is not allowed to push another copy of the stack bottom sign onto the stack again.

All other signs from the stack alphabet can be stored there in any amount and order.

The uppermost sign of the stack is the active stack sign. It has the same influence on the automaton as the tape character on which the read head is currently positioned. Unlike the tape, however, the stack does not require a read head. The active sign is always the topmost sign in the stack.



A pushdown automaton with all components

4.2.5 Structure of a non-deterministic pushdown automaton (NPDA)

Since every pushdown automaton contains a state machine inside it, and since it is possible to arrange the arrows in the machine in such a way that they cannot be described as a function but as a relation, it is not a big surprise that pushdown automatons can also be deterministic and nondeterministic. The more general type is the non-deterministic variant, therefore it is described here first:

A is a non-deterministic pushdown automaton (NPDA), and each NPDA is a set with exactly 6 elements:

$$A = \{Z, Z_0, \Sigma, K, \delta, \#\}$$

- Z is the set of all states (represented as nodes in the state graph)
- Z_0 is the initial state $Z_0 \in Z$ (In the state graph marked by an arrow coming from a small dot. This dot is not a state, it only marks the initial state).
- Σ is the tape alphabet. There are only signs from this alphabet on the tape. It is the same alphabet that contains the terminal signs in a grammar. So it is the alphabet from which the words in the language consist.
- K is the stack alphabet. It contains all the signs that can be found in the stack.
- δ is a relation: $\delta \subseteq (Z \times \Sigma \times K) \times (Z \times K)$ It takes as "input" an element from the Cartesian product $Z \times \Sigma \times K$ (a combination of state, tape sign, and stack sign) and produces as "output" a subset of the Cartesian product $Z \times K$ (none, one or more combinations of state and stack sign). In the state graph, this relation is the set of all arrows together with their labels.
- $\#$ is the stack bottom sign. It is a sign from the stack alphabet

4.2.6 Acceptance

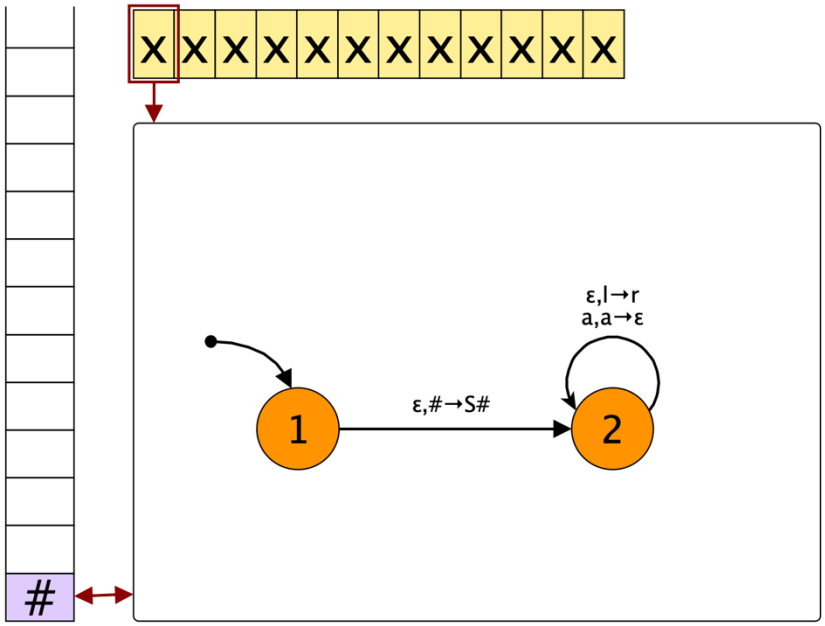
It is noticeable that the pushdown automaton has no accepting states. It does not need them, because the acceptance of a word is handled differently:

A pushdown automat accepts a word exactly if it can read it to the end and if then, at the end, the stack is empty again (only the stack bottom sign is still in the stack).

There is also the possibility to define accepting states, but it is always possible to convert any pushdown automaton into a pushdown automaton that has exactly one accepting state, which can only be reached if the input word has been read to the end and the stack is empty at the same time. But this is completely equivalent to an automaton that accepts a word when exactly these conditions are met. Therefore it is not necessary at all to define accepting states.

4.2.7 Create a pushdown automaton from a context-free grammar

There is a very simple algorithm to build a pushdown automaton from any context-free grammar. The result is a non-deterministic pushdown automaton that gets by with only 2 states. It is based on a very simple framework, which looks like this:



Let's look at this with an example. For this purpose, we start from this grammar, which generates palindromes with an even number of signs:

- $G = \{V, \Sigma, P, S\}$
- $\Sigma = \{0, 1\}$
- $V = \{\mathrm{S}\}$
- $\mathrm{P} = \{\mathrm{S} \rightarrow 0\mathrm{S}0 \mid 1\mathrm{S}1 \mid \varepsilon\}$
- $S = S$

$\mathcal{L}(G) = \{\varepsilon, 00, 11, 0000, 0110, 1001, 1111, 000000, 001100, 010010, \cdots\} = ww^R$

From this we want to construct a pushdown automaton that recognizes exactly this language:

$$A = \{Z, Z_0, \Sigma, K, \delta, \#\}$$

As said, the resulting automaton always has exactly 2 states, which we can call as we like:

$$Z = \{①, ②\}$$

Of these, one is the start state:

$$Z_0 = ①$$

The tape alphabet Σ of the automaton is the terminal alphabet Σ of the grammar

$$\Sigma_G = \{0, 1\} \quad \rightarrow \quad \Sigma_A = \{0, 1\}$$

The stack alphabet is the union of the stack bottom character # and the two alphabets of the grammar.

$$K = \{\#\} \cup \Sigma \cup V = \{0, 1, S, \#\}$$

The stack bottom character is #:

$$\# = \#$$

So the only thing missing is the transition relation δ. This relation contains several elements which can be assigned to 3 different classes:

Class 1 (one fixed rule)

The rule that leads from state ① to state ② from the template in the diagram is labeled "$\varepsilon, \# \rightarrow S\#$" in the state graph above the arrow, and would be written as an element of P like this:

$$(①, \varepsilon, \#) \rightarrow (②, S\#)$$

The three symbols in the left parenthesis indicate the condition under which the rule can be applied:

- ① The state the automaton must be in to apply this rule.
- ε The sign that is read from the tape. If, as here, there is ε, it means that the empty word is to be read. This is just a shortcut to indicate: No sign is read (the read head does not move to the right, and the automaton ignores the sign that is there).
- $\#$ The third symbol is the sign that must be read from the top of the stack. Here it is the cellar bottom character. The read sign is always removed from the stack, so by reading it the stack shrinks by 1 sign.

The two symbols in the right parenthesis tell what happens when you apply this rule:

- ② This is the state the automaton must change to, i.e. the next state.
- $S\#$ This is the word that must be written on the stack instead of the sign just read. Here, this word has two signs, and they are written on the stack in reverse order. So you have to move the # onto the stack first, which just means that as an overall effect, this sign just stays where it was. Then you also put the S sign on top of it.

When labeling an arrow, it is not necessary to indicate the source and target states, because this is clear anyway from the starting point and the end point of the arrow, and the parentheses can also be omitted without causing any misunderstanding.

This single rule of class 1 means:

If the automaton is in the start state ①, and you can read the empty word ε from the tape (i.e., you read nothing which is always possible), while the top sign of the stack is the stack bottom sign, then you can put the sign S on the stack bottom. Thereby with S is meant exactly the start sign of the grammar. Since this is the only rule that can be applied when the automaton is in the initial state, this rule must be executed first. After that, the automaton is in state ② and can never leave it again, because there is no rule that could return the automaton to state ①. The whole effect of this rule is only that afterwards the start sign S is on the stack.

Class 2 (one rule per terminal sign)

The rules from this class are shown in the template as follows:

$$a, a \rightarrow \varepsilon$$

Since they are a part of the label of that arrow leading from state ② to state ②, the general pattern of elements in relation δ looks like this:

$$(②, a, a) \rightarrow (②, \varepsilon)$$

Here a is a placeholder for all elements from the set Σ. This set contains the two elements 0 and 1, so the real elements of δ look are these:

$$(②, 0,0) \rightarrow (②, \varepsilon)$$

$$(②, 1,1) \rightarrow (②, \varepsilon)$$

Any rule that matches this type works as follows:

If the read head is on a tape sign that matches the top sign in the stack, then that sign is removed from the stack and replaced with the empty word (i.e., nothing). The net effect is the removal of the sign from the stack.

When there is such a match, there is always exactly one rule that can be applied, which means nothing more than that matching signs are always removed.

Class 3 (one transition rule in the automaton for each production rule of the grammar)

The last class of rules is represented in the template above the arrow from ② to ② by "$\varepsilon, l \rightarrow r$", which as an element of P would look like this: $(②, \varepsilon, l) \rightarrow (②, r)$. However, l and r are placeholders for what is the real magic of this automaton.

What really needs to be there can be derived directly from the grammar and is shown in the following table:

Element of the production rule P from the grammar G	Element of the transition relation δ of the automaton A
$S \rightarrow 0S0$	$(②, \varepsilon, S) \rightarrow (②, 0S0)$
$S \rightarrow 1S1$	$(②, \varepsilon, S) \rightarrow (②, 1S1)$
$S \rightarrow \varepsilon$	$(②, \varepsilon, S) \rightarrow (②, \varepsilon)$

So the l in the general formula $(②, \varepsilon, l) \rightarrow (②, r)$ always takes the value of the left side of the production rule, and r always gets the right value of the rule. Thus r always takes the value of a nonterminal sign, and in this way the rules from class 3 differ from those from class 2, because in class 2 there are always terminal signs at this point.

Let us summarize:

- $A = \{Z, Z_0, \Sigma, K, \delta, \#\}$
- $Z = \{①, ②\}$
- $Z_0 = ①$
- $\Sigma = \{0, 1\}$
- $K = \{0, 1, S, \#\}$

- $\delta = \left\{ \begin{array}{l} \left(①, \varepsilon, \#\right) \rightarrow \left(②, S\#\right), \\ \left(②, 0{,}0\right) \rightarrow \left(②, \varepsilon\right), \\ \left(②, 1{,}1\right) \rightarrow \left(②, \varepsilon\right), \\ \left(②, \varepsilon, S\right) \rightarrow \left(②, 0S0\right), \\ \left(②, \varepsilon, S\right) \rightarrow \left(②, 1S1\right), \\ \left(②, \varepsilon, S\right) \rightarrow \left(②, \varepsilon\right) \end{array} \right\}$
- $\# = \#$

So, this algorithms proves, that it is possible to transform any given context-free grammar into a nondeterministic PDA. This means, that the set of languages produced by context-free grammars is a subset of the set of languages that NPDAs can recognize.

But there is also an algorithm that allows you to transform any given NPDS into a context-free grammar. This algorithm is complicated, so it is not shown here, but it exists. And its existence proves, that also the set of languages that NPDAs can recognize is a subset of the set of languages produced by context-free grammars.

So, again we have the situation, that two sets are the subset of each other, which means that both sets are equal.

Context-free grammars perfectly correspond with non-deterministic PDAs.

4.2.8 Some example runs

This automaton was derived from a grammar that generates palindromes over the alphabet $\{0,1\}$. So, it should be able to recognize such palindromes.

Let's assign numbers to the rules:

1	$\left(①, \varepsilon, \#\right) \rightarrow \left(②, S\#\right)$
2	$\left(②, 0{,}0\right) \rightarrow \left(②, \varepsilon\right)$
3	$\left(②, 1{,}1\right) \rightarrow \left(②, \varepsilon\right)$
4	$\left(②, \varepsilon, S\right) \rightarrow \left(②, 0S0\right)$
5	$\left(②, \varepsilon, S\right) \rightarrow \left(②, 1S1\right)$
6	$\left(②, \varepsilon, S\right) \rightarrow \left(②, \varepsilon\right)$

001100

We feed 001100 into the pushdown automaton and see what happens:

No.	State	Rest of Input	Stack	rule to be applied	
1	①	001100	$\#$	1	$\left(①, \varepsilon, \#\right) \rightarrow \left(②, S\#\right)$
2	②	001100	$S\#$	4	$\left(②, \varepsilon, S\right) \rightarrow \left(②, 0S0\right)$
3	②	001100	$0S0\#$	2	$\left(②, 0{,}0\right) \rightarrow \left(②, \varepsilon\right)$

4	②	01100	$S0\#$	4	$(②, \varepsilon, S) \rightarrow (②, 0S0)$
5	②	01100	$0S00\#$	2	$(②, 0,0) \rightarrow (②, \varepsilon)$
6	②	1100	$S00\#$	5	$(②, \varepsilon, S) \rightarrow (②, 1S1)$
7	②	1100	$1S100\#$	3	$(②, 1,1) \rightarrow (②, \varepsilon)$
8	②	100	$S100\#$	6	$(②, \varepsilon, S) \rightarrow (②, \varepsilon)$
9	②	100	$100\#$	3	$(②, 1,1) \rightarrow (②, \varepsilon)$
10	②	00	$00\#$	2	$(②, 0,0) \rightarrow (②, \varepsilon)$
11	②	0	$0\#$	2	$(②, 0,0) \rightarrow (②, \varepsilon)$
12	②	ε	$\#$	accepted	

This would happen if the oracle lied and gave a wrong advise:

No.	State	Rest of Input	Stack	rule to be applied	
1	①	001100	$\#$	1	$(①, \varepsilon, \#) \rightarrow (②, S\#)$
2	②	001100	$S\#$	4	$(②, \varepsilon, S) \rightarrow (②, 0S0)$
3	②	001100	$0S0\#$	2	$(②, 0,0) \rightarrow (②, \varepsilon)$
4	②	01100	$S0\#$	4	$(②, \varepsilon, S) \rightarrow (②, 0S0)$
5	②	01100	$0S00\#$	2	$(②, 0,0) \rightarrow (②, \varepsilon)$
6	②	1100	$S00\#$	6	$(②, \varepsilon, S) \rightarrow (②, \varepsilon)$
12	②	1100	$00\#$	no matching rule = not accepted	

However, it does not matter that there are runs where the word is not accepted. As soon as there is even a single run in which the word is accepted, the word is considered recognized, and thus it also belongs to the language that the automaton recognizes.

Let us now try to process a word that is not a palindrome:

No.	State	Rest of Input	Stack	rule to be applied	
1	①	0101	$\#$	1	$(①, \varepsilon, \#) \rightarrow (②, S\#)$
2	②	0101	$S\#$	4	$(②, \varepsilon, S) \rightarrow (②, 0S0)$
3	②	0101	$0S0\#$	2	$(②, 0,0) \rightarrow (②, \varepsilon)$
4	②	101	$S0\#$	5	$(②, \varepsilon, S) \rightarrow (②, 1S1)$
5	②	101	$1S10\#$	3	$(②, 1,1) \rightarrow (②, \varepsilon)$
6	②	01	$S10\#$	6	$(②, \varepsilon, S) \rightarrow (②, \varepsilon)$
7	②	01	$10\#$	no matching rule = not accepted	

Again, there are several different runs possible, but no single possible run will result in a state where the word is read entirely and where the stack is then empty. Therefore the word 0101 is not accepted. It is not a word of the language of palindromes.

4.2.9 Structure of a deterministic pushdown automaton (DPDA)

We talked a lot about nondeterministic PDA, but of course there are also deterministic PDA that work without this nasty oracle:

A is a deterministic pushdown automaton (DPDA), and each DPDA is a set with exactly 6 elements:

$$A = \{Z, Z_0, \Sigma, K, \delta, \#\}$$

Z	is the set of all states (represented as nodes in the state graph)
Z_0	is the initial state $Z_0 \in Z$ (In the state graph marked by an arrow coming from a small dot. This dot is not a state, it only marks the initial state).
Σ	is the tape alphabet. There are only signs from this alphabet on the tape. It is the same alphabet that contains the terminal signs in a grammar. So it is the alphabet from which the words in the language consist.
K	is the stack alphabet. It contains all the signs that can be found in the stack.
δ	is a function: $\delta: (Z \times \Sigma \times K) \rightarrow (Z \times K)$ It takes as "input" an element from the Cartesian product $Z \times \Sigma \times K$ (a combination of state, tape sign, and stack sign) and produces as "output" exactly one element of the Cartesian product $Z \times K$. In the state graph, this function is the set of all arrows together with their labels.
#	is the stack bottom sign. It is a sign from the stack alphabet

So, it's easy to define them, and they surely also recognize some languages, but the big question is this:

Is there an algorithm, that makes it possible to convert every NPDA into a DPDA? If there was such an algorithm, then the set of languages recognized by DPDAs would be identical to the set of languages recognized by NPDAs, like we had it before with NFSM and DFSM.

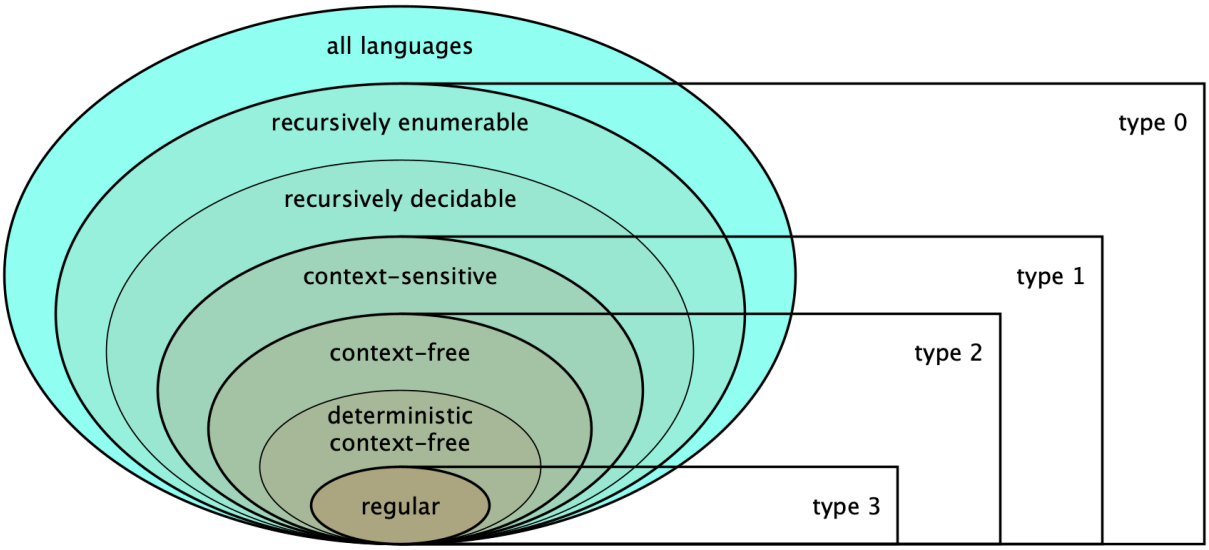
The answer is: NO.

There are languages that can only be recognized by a NPDA. And an example for such a language is just the palindrome-language from our example.

No matter how the automaton is constructed: To accept a palindrome, it needs to know where exactly the middle of the word is. From that point on the string repeats just in reverse order. But how can you know where the middle of the word is without having read it to the end? Imagine you are listening to a new song you never heard before: How can you tell for sure, where exactly the middle of the song is, while you are listening? You have no idea how long the song is while you are listening, and so it is impossible to know the exact point in the middle. And for the very same reason also a DPDA is unable to recognize palindromes.

And this means: The set of languages recognizable by DPDAs is a real subset of the set of languages recognizable by NPDAs.

Remember this image?



Now you understand why there are two bubbles for type-2-languages.

4.2.10 Practical Applications

Programming languages

Almost all programming languages are context-free languages. This is, because almost all parsers for programming languages are PDAs, although most of them have some extras, for practical purposes.

Natural languages

For a long time linguists thought that natural languages like English, Arab, Chinese, Urdu, Swahili etc. were context-free languages. But it has been proven, that Swiss dialects ("Swiss German") contains elements, that can't be created by context-free grammars. So it is in doubt if any natural language is context-free.